Identify minimal failure-causing schemas for multiple faults

XINTAO NIU and CHANGHAI NIE, State Key Laboratory for Novel Software Technology, Nanjing University
HARETON LEUNG, Hong Kong Polytechnic University

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). To extend and make full use of CT, the theory of Minimal Failure-Causing Schema(MFS) has been proposed. The use of MFS helps to isolate the root cause of the failure, which is desired after detecting them by CT. Most existed algorithms based on MFS theory focus on identifying the MFS in SUT with the single fault, however, we argue that multiple faults is the more common testing scenario, and under which masking effects may be triggered so that some expect faults will not be observed normally. Traditional MFS theory as well as its identifying algorithms lack a mechanism to handle such effects, hence will make them incorrectly isolate the MFS in SUT. To address this problem, we proposed a new MFS model with considering multiple faults. We first formally analyse the impact of the multiple faults on existed MFS isolating algorithms, especially when masking effects were triggered between these multiple faults. Based on this, we then give an approach that can assist traditional algorithms to better handle the multiple faults testing scenarios. Empirical studies with several open-source software were conducted, which show that multiple faults with masking effects do negatively affect on traditional MFS identifying approach and our approach can help them to alleviate these effects.

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

General Terms: Reliability, Verification

Additional Key Words and Phrases: Software Testing, Combinatorial Testing, Failure-inducing combinations, Masking effects

ACM Reference Format:
Xintao Niu, Changhai Nie and Hareton Leung, 2014.Identify failure-causing schemas for multiple faults. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 23 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

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 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. In the worst case we need to examine every possible combination of these factors as each such combination can contain unique faults[Song et al. 2012]. While conducting such exhaustive testing is ideal and necessary in theory, it is impractical and not economical in consideration of the limited testing time and computing resource. One remedy for this problem is

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)

DOI:http://dx.doi.org/10.1145/0000000.0000000

Table I. MS word example

id	*Highlight*	*Status bar*	*Bookmarks*	*Smart tags*	**Outcome**
1	On	On	On	Off	PASS
2	On	Off	Off	On	PASS
3	Off	On	Off	Off	Fail
4	Off	Off	On	Off	PASS
5	Off	On	On	On	PASS

combinatorial testing, which systematically sample the interaction space and select a relatively small set 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. Many works in CT aims to construct the smallest set of such efficient testing object [Cohen et al. 1997; Bryce et al. 2005; Cohen et al. 2003], which also called *covering array*.

Once failures are detected by covering array, it is desired to isolate the failure-inducing combinations in these failing test cases. This task is important in CT as it can facilitate the debugging efforts by reducing the code scope that needed to inspected[Ghandehari et al. 2012]. Only with the information stum from the covering array sometimes is far from clear to identify the location and magnitude of the failure-inducing combinations [Colbourn and McClary 2008]. Thus, deeper analysis needed to be conducted. Take an example [Bach and Schroeder 2004], Fig I present a 2-way covering array for testing the MS word application, in which we want to examine various combination of options 'Highlight','Status Bar', 'Bookmarks' and 'Smart tags' of MS word. Assume the third test case failed, then we can get that there are five 2-way suspicious combinations may be responsible for this failure: (Highlight: Off, Bookmarks: Off),(Highlight: Off, Smart tags: Off), (Status Bar: On, Bookmarks: Off), (Status Bar: On, Smart tags: Off),(Bookmarks: Off, Smart tags: Off). (Note that (Highlight: Off, Status Bar: On) excludes in this set as it appeared in the fifth passing test case). Without any more information, we cannot figure out which one or some in this suspicious set caused this failure. In fact, taking account of the higher-strength combination, e.g.,(Highlight: Off,Status Bar: On, Smart tags: Off), the problem will grow more complicated.

To address this problem, prior work [Nie and Leung 2011a] specifically studied the properties of the minimal failure-causing schemas in SUT, based on which, a further diagnosis with generating additional test cases was applied and therefore can identify the MFS in the test case. Other works have been proposed to identify the MFS in SUT, which include approaches such as building tree model [Yilmaz et al. 2006], exploiting the methodology of minimal failure-causing schema [Nie and Leung 2011a], ranking suspicious classification ous combinations based on some rules[Ghandehari et al. 2012], using graphic-based deduction [Martínez et al. 2008] and so on. These approaches can be partitioned into two categories [Colbourn and McClary 2008]: *adaptive*–test cases are chosen based on the outcomes of the executed tests [Nie and Leung 2011a; Ghandehari et al. 2012; Niu et al. 2013; Zhang and Zhang 2011; Shakya et al. 2012; Wang et al. 2010; Li et al. 2012]or *nonadaptive*–test cases are chosen independently and can be executed parallel [Yilmaz et al. 2006; Colbourn and McClary 2008; Martínez et al. 2008; 2009; Fouché et al. 2009].

The MFS methodology as well as other MFS identifying approaches mainly focus on the ideal scenario that SUT only contains one fault, i.e., the test case under testing can either fails or passes the testing. However, in this paper, we argue that SUT with multiple distinguished faults is the more common testing scenario in practice, and moreover, this do have impact on the Failure-inducing Combinations Identifying(FCI) approaches. One main impact of multiple faults on FCI approaches is the masking effects. A masking effect [Dumlu et al. 2011; Yilmaz et al. 2013] 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 an incompatibility between option '-c' and '-o'. When we put this two scenarios into one test case, only one fault information will be observed, which means another fault is masked by the observed one. This effects will prevent test cases normally executing, consequently make approaches make a incorrectly judgements on the correlation between the combinations checked in the test case and the fault that been prevented to be observed.

As known that masking effects negatively affect the performance of FCI approaches, a natural question is how this effect bias the results of FCI approaches. In this paper, we formalized the process of identifying MFS 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 are 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 handles each fault in SUT. For a particular fault under analysis, when applying traditional FCI approaches to identify the failure-inducing combinations, we pick the test cases generated by FCI approaches that trigger unexpected faults and replace them with regenerated newly test cases. These newly test cases should either pass or trigger the same fault under analysis.

Our initial work demonstrated that the extent varies to what different FCI approaches suffered from the masking effects, as a result, follow-on work in this paper was proposed to construct an additional voting system, in which we take comprehensive account of various result from different algorithms. We rank the suspicious MFS according to the frequency it appeared in each algorithm's output, and recommend the MFS which has a high ranking.

To evaluate the performance of our strategy and the voting system, we applied our strategy on three FCI approaches, which are CTA [Yilmaz et al. 2006], OFOT [Nie and Leung 2011a], FIC_BS [Zhang and Zhang 2011] respectively. The subjects we used are several open-source software with the developers' forum in Source-Forge community. 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 augmented versions to identify the failure-inducing combinations in the subjects respectively. The results of our empirical studies shows that the masking effects do impact on the FCI approaches, although to what the extent varies, and the approaches augmented with our strategy can identify failure-inducing combinations more accurately than traditional ones when facing masking effects, and moreover, this performance can get further improvement with using voting system.

The main contributions of this paper are:

— We studied the impact of the masking effects among multiple faults on the isolation of the failure-inducing combinations in SUT.
— We proposed a divide and conquer strategy of selecting test cases to alleviate the impact of this effects.

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

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

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

  return x+y;
  }
```

Fig. 1. A toy program with four input parameters

Table II. test inputs and their corresponding result

id	test inputs	results	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

— We designed a voting system for different FCI approaches which further improve the performance at finding MFS in SUT.
— We conducted several empirical studies and showed that our strategy can assist FCI approaches to get better performance on identifying failure-inducing combinations in SUT with masking effects.

2. MOTIVATING 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: $v_a = \{7, 11\}, v_b = \{2, 4, 5\}, v_c = \{4, 6\}, v_d = \{3, 5\}$. The detail code of the method is listed in Figure 1.

Inspecting the simple code in Figure 1, we can find two potential faults: First, in the step 1 we can get an *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, instead, they apply 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 few 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 III. 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 generate every possible inputs listed in the Column "test inputs" of Table II, and their execution results are listed in Column "result" of Table II. In this Column, *PASS* means that the program runs without any exception under the input in the same row. *Ex 1* indicates that the program triggered an exception corresponding to the step 1 and *Ex 2* indicates the program triggered an exception corresponding to the step 2. According to data listed in table II, we can figure out that (-, 4 , 4, -) must be the failure-inducing combination of Ex 1 as all the inputs triggered Ex 1 contain this combination. 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 their corresponding exception in Table III.

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 has 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 [Nie and Leung 2011b].

Definition 3.1. 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 are triggered by some test cases, what is desired is to figure out the cause of these faults, and hence some subsets of this test case should be analysed.

Definition 3.2. 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 *hits* the *combination*.

Definition 3.3. 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 3.4. 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 then call the combination c the *minimal faulty combination* for F (It is also called Minimal failure-causing schema(MFS) in [Nie and Leung 2011a]).

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 causing the failure and hence facilitate the debugging efforts.

Let c_m be a m-value combination, we denote the set of 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 the set of 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 3.5. *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 $T(c_k) \subset T(c_l)$. □

PROPOSITION 3.6.
For any set T of test cases of a SUT, we can always get a set of minimal combinations $\mathcal{C}(T) = \{c\}$, i.e., $\nexists c', c \in \mathcal{C}(T), s.t. c' \prec c$, 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 $M(S(T)) = \{c|c \in S(T)\ and\ \not\exists c' \prec c, s.t., c' \in S(T)\}$. 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 prove 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 $t \in \bigcup_{c\in M(S(T))} T(c)$. So, $\bigcup_{c\in S(T)} T(c) \subset \bigcup_{c\in M(S(T))} T(c)$.

Hence, $\bigcup_{c\in S(T)} T(c) = \bigcup_{c\in M(S(T))} T(c)$, and $M(S(T))$ is the set of combinations that holds this proposition. □

. Let $\mathcal{C}(T_{F_m})$ denote the set of all the test cases triggering fault F_m , then $\mathcal{C}(T_{F_m})$ actually is the set of failure-inducing combinations of F_m. And obviously for $\forall c_m$ we have $\mathcal{C}(T(c_m)) = \{c_m\}$.

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

PROPOSITION 3.7. *For any $T(c) \subset T$, then it must be that $c \in S(T)$.*

PROOF. We first have $\mathcal{C}(T(c)) = \{c\}$, and $\mathcal{C}(T(c)) \subset S(T(c))$. Then as $T(c) \subset T$, 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 3.8. *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 have $c_l \in S(T_k)$. 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 $\mathcal{C}(T_k)$. □

. Based on this lemma, in fact, the $c_k \in \mathcal{C}(T_k)$ remains the following three possibilities: 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) $\not\exists c_l \in \mathcal{C}(T_l), s.t., c_k \prec c_l\ or\ c_k = c_l,\ or\ c_l \prec c_k$. For the third case, we call c_k is *irrelevant* to $\mathcal{C}(T_l)$

We illustrate these scenarios in Table IV. 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 irrelevant to $\mathcal{C}(T_l)$.

Table IV. Example of the 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, -, -)

3.2. Identify the MFS

According to aforehand analysis, we can get that $\mathcal{C}(T_{F_m})$ actually is the set of failure-inducing combinations of F_m. Then in theory, if we want to accurately figure out the MFS in SUT, we need to exhaustively execute each possible test case, and collect the failing test cases T_{F_m}. This is impossible in practice especially when the testing space is too big.

So for traditional FCI approaches, they need to select part test cases, and then either take some assumption to make prediction of the remaining test cases or just give a suspicious rank. As giving a suspicious rank can also be regard as a special case of making prediction(with computing the possibility), so we next only formally describe the mechanism of FCI approaches belong to the first type.

We refer the observed failing test case to $T_{fail_{observed}}$, and refer the remaining test cases the approach predict to be failed as $T_{fail_{predicted}}$. Then the MFS identified by FCI approaches can be depicted as:

MFS = $\mathcal{C}(T_{fail_{observed}} \bigcup T_{fail_{predicted}})$.

For each FCI approach, the way it predict the $T_{fail_{predicted}}$ according to observed failing test cases varies, further more, as the test cases it generates is different, the failing test cases observed by different test cases, i.e., $T_{fail_{observed}}$ also varies. We take an example using OFOT approach to illustrate this formula.

Suppose SUT has 3 parameters, each can take 2 values. And it find the test case (1, 1, 1) failed. Then we can describe the FCI process as Table V. In this table, test case *t* failed, and OFOT mutate one factor of the *t* one time to generate new test cases: t_1 – t_3, It found the the t_1 passed, which indicates that this test case break the MFS in the original test case *t*. So the (1,-,-) should be one failure-causing factor, and as other mutating process all failed, which means no other failure-inducing factors, therefore, the MFS in *t* is (1,-,-).

Now let us explain this process with our formal model. The $T_{fail_{observed}}$ is {(1,1,1),(1,0,1),(1,1,0)}. And as finding (0,-,-) break the MFS, hence, all the test cases contain (0,-,-) should pass the test cases, so (0,1,1),(0,0,1),(0,1,0),(0,0,0) should pass the testing. Further, as obviously the test case either pass or fail the testing(we label the skip the testing as a special case of failing), so the remaining test case(1,0,0), OFOT predict it as failing, i.e., $T_{fail_{predicted}}$ is {(1,0,0)}. Taking together, the MFS using OFOT strategy can be described as: $\mathcal{C}(T_{fail_{observed}} \bigcup T_{fail_{predicted}})$ = $\mathcal{C}(\{(1,1,1),(1,0,1),(1,1,0),(1,0,0)\})$ = (1,-,-), which is identical to the one it got previously.

Similarly, other FCI approaches can also be modeled into this formal description. It is noted that the test cases FCI predict to be failing is not always identical to the actually failing test cases. Thus, besides FCI predicted completely accurate,

Table V. OFOT with our strategy

original test case				**Outcome**
t	1	1	1	Fail
	observed			
t_1	0	1	1	Pass
t_2	1	0	1	Fail
t_3	1	1	0	Fail
	predicted			
t_4	0	0	1	Pass
t_5	0	1	0	Pass
t_6	1	0	0	Fail
t_7	0	0	0	Pass

$T_{fail_{observed}} \bigcup T_{fail_{predicted}}$ can either contain T_{fail} or be contained by T_{fail} , or neither of the two circumstances. For the third circumstance, we have the following proposition:

PROPOSITION 3.9. *If neither $T_m \subset T_n$ nor $T_n \subset T_m$, then it must be that $c \in \mathcal{C}(T_n)$ is irrelevant to $\mathcal{C}(T_m)$ and must be some $c' \in \mathcal{C}(T_m)$ is irrelevant to $\mathcal{C}(T_n)$.*

PROOF. We just need to prove the first one, as the latter is equivalent to the first one. The first one can be proven by contradiction. □

So either of this three circumstance can make FCI identifying inaccurately as we discussed earlier, and each FCI should avoid these circumstances as much as possible.

3.3. Masking effect

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

Definition 3.10. 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.

3.3.1. 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 failure-inducing combinations of Ex 2 and (- - 4 5) is irrelevant to the failure-inducing combinations of Ex 1.

3.3.2. Distinguish faults. Distinguishing the faults by the exception traces or error code can help make FCI approaches focus on particular fault. Yilmaz in [Yilmaz et al. 2013] 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 combination(-,-,4,5).

3.3.3. masking effects for FCI approaches. Previously we just discussed how failure-inducing combinations under exhaustive testing. Next we see what traditional FCI will face, i.e., $T_{fail_{observed}} \bigcup T_{fail_{predicted}}$ will grow into what.

In fact, the masking effects firstly change the $T_{fail_{observed}}$, as masking effects, FCI either will identify $T_{fail_{observed}}$ less or more than it expected to be, as a consequence, regardless of $T_{fail_{predicted}}$ can be, masking effects makes FCI approaches never reach a perfect result, i.e., accurately and completely find each MFS.

Secondly, as $T_{fail_{observed}}$ changed, $T_{fail_{predicted}}$ consequently changed. As a result, if $T_{fail_{predicted}}$ it can accurate predict without masking effects, then masking effects obviously makes the FCI approaches worse. However, if it originally cannot predicted inaccurately, then this changing will have a possibility make the $T_{fail_{observed}}$ changing better, and of course, it also can have a possibility make the $T_{fail_{predicted}}$ changing worse.

3.4. 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 latter may get the parent combinations of the failure-inducing combinations or may ignore some of them.

Note that our discussion is based on that 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 $T(mask_{F_m}) \subset \bigcup_{i=1\&j\neq m}^{L} T_{F_i}$, in order to weaken 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, there is no room left to improve accuracy. However, when just

ALGORITHM 1: replace test cases triggering unexpected faults

Input: the original test case $t_{original}$,fault type F_m,fixed part s_{fixed}, values set that each option can take $Param$
Output: t_{new} the regenerate test case,The frequency number
$index$ = 0; $FreNum_\alpha$ = -1;
while *not* *MeetEndCriteria()* **do**
  $s_{mutant} \leftarrow t_{original} - s_{fixed}$;
  forall the $opt \in s_{mutant}$ **do**
    $i = getIndex(Param, opt)$;
    $opt \leftarrow opt'$ $s.t.$ $opt' \in Param[i]$ and $opt'! = opt$;
  end
  $t_{new} \leftarrow s_{fixed} \bigcup s_{mutant}$;
  $result \leftarrow execute(t_{new})$;
  if $result == PASS$ or $result == F_m$ **then**
    return t_{new};
  else
    continue;
  end
end
return *null*

needing to select part of the whole test cases to identify failure-inducing combinations, which is practical and sometimes the only solution for large-scale SUT, we can adjust the test cases we need to use by selecting proper ones so that we can limit the size of $T(mask_{F_m})$ to be as small 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 newly test cases to replace them. These regenerated test cases should either pass in the execution or trigger F_m. 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 triggers 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 mutate other part with different values from the original one. For example, if a test case (1,1,1,1) triggered an unexpected fault, and the fixed part is (-,-,1,1), then we can replace it with a test case (0,0,1,1) which may either pass or trigger expected fault.

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

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

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_m, 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_m or passes.

This algorithm is a loop(line 1 - 14) containing two parts:

The first part(line 2 - 7) generates a new test case which is different from the original one. This test case will keep the fixed part (line 7), and just mutate the factors which are not in that part(line 2). The mutation for each factor works by selecting one legal value(by random) which is different from the original one(line 3 - 6). The generated

newly test case must be different from each iteration(we implemented it by hashing method).

Second part is to check whether the newly generated test case is as expected(line 8 - 13). We first execute the SUT under the newly generated test case(line 8) and then check the executed result. Either the test case passes or triggers the same fault – F_m meets the requirement(line 9), and if so we will directly return this test case(line 10). Otherwise, we will repeat the process, i.e., generate newly test case and check again(line 11 -12).

It is noted that the loop has another exit besides we find an expected test case(line 10), which is when the function *MeetEndCriteria()* returns *true*(line 1). We didn't explicitly show what the function *MeetEndCriteria()* is like, because this is depending on the computing resource and how accurate you want to the identifying result to be. In detail, if you want to get a high quality result and you have enough computing resource, you can try much times to get the expected test case, otherwise, a relatively small number of attempts is recommended.

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

4.2. A case study with the replacing strategy

Suppose we have to test a system with four parameters, each has three options. And when we execute the test case (0 0 0 0), a failure–$Err1$ is triggered. Next we will use the FCI approach – OFOT with replacing strategy to identify the failure-inducing combinations for the $Err1$. The process is listed in Table VI. 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 table VI, we can find the algorithm mutates one factor to take the different value from the original test case one time. Originally if the test case encounter the result different from the 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 it 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 remaining test cases triggered either $Err1$ or $Err2$(regard as one fault). Additionally when we take the *distinguish faults* strategy, the failure-inducing combinations obtained is (- 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

We conducted several empirical studies to address the following questions:

Q1: Do masking effects exist 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?

Table VI. OFOT with our strategy

original test case					**fault info**
t	0	0	0	0	Err 1
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 VII. 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

Q4: Does voting system that consists of different approaches can make improvements?

5.1. The existence of masking effects

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 VII 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 picked some bugs which are caused by the options combination. 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 results 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, so that we can reproduce different faults through controlling the inputs to that merged test file. This merging manipulation varies with the pair of bugs we selected, and for each pair of bugs, the source code of the merging file as well as other detailed experiment information is available at– 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 in Table VIII and IX for HSQLDB and JFLex respectively. Each table is organised into four groups: 1)"common options", which lists the options as well as their values under

which every version of this software can be tested. 2)“common boolean options”, which lists additional common options whose values type is boolean. 3)“specific options”, under which only the specific version of that software can be tested. 4)“configure space”, which depicts the input model for each version of the software.

We then generated the exhaustive test suite consist of all the possible combinations of these options and under each of them 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.

Table VIII. Input model of HSQLDB

common options		**values**
Server Type		server, webserver, inprocess
existed form		mem, file
resultSetTypes		forwad, insensitive, sensitive
resultSetConcurrencys		read_only, updatable
resultSetHoldabilitys		hold, close
StatementType		statement, prepared
common boolean options		
sql.enforce_strict_size, sql.enforce_names, sql.enforce_refs		
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 IX. Input model of JFlex

common options		**values**
generation		switch, table, pack
charset		default, 7bit, 8bit, 16bit
common boolean options		
public, apiprivate,cup,caseless,char,line,column,notunix, yyeof		
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	$2^{11} \times 3^2 \times 4^1$	

5.1.2. Result and discusssion. Table X 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 X, the test cases with masking effects do exist, i.e., test cases containing failure inducing combinations did not trigger the corresponding bug. In effect, there is about

Table X. 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

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 remaining software versions, 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 the traditional algorithms

In the second study, our aim is to learn the degree that 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 for the prepared software in Table VII and compare them with the prior failure-inducing combinations.

5.2.1. Study setup. The traditional approaches we selected are: OFOT[Nie and Leung 2011a], FIC_BS [Zhang and Zhang 2011] and CTA[Yilmaz et al. 2006], in which CTA is a integrated failure characterization part of FDA-CIT[Yilmaz et al. 2013]. As CTA is a post-analysis technique applied on given test cases, different test cases will influence the result of characterization process. So to avoid randomness and to 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 [Hall et al. 2009].

For each test case in the exhaustive set for a particular software, we applied O-FOT, 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 adopted 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 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 that 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 the common combinations appeared in both identified combinations and prior failure-inducing combinations. We denote 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 referred 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, high *accurate number* value indicates FCI approaches performs effectively, while *ignored number* and *irrelevant number* indicate the degree of deviation for the FCI approaches. For *parent number* and *sub number*, they indicate FCI approaches that, although with additional noisy information, can determine part parameter values about the failure-inducing combinations.

This case study was 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 illustrated them together in Figure 2 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*.

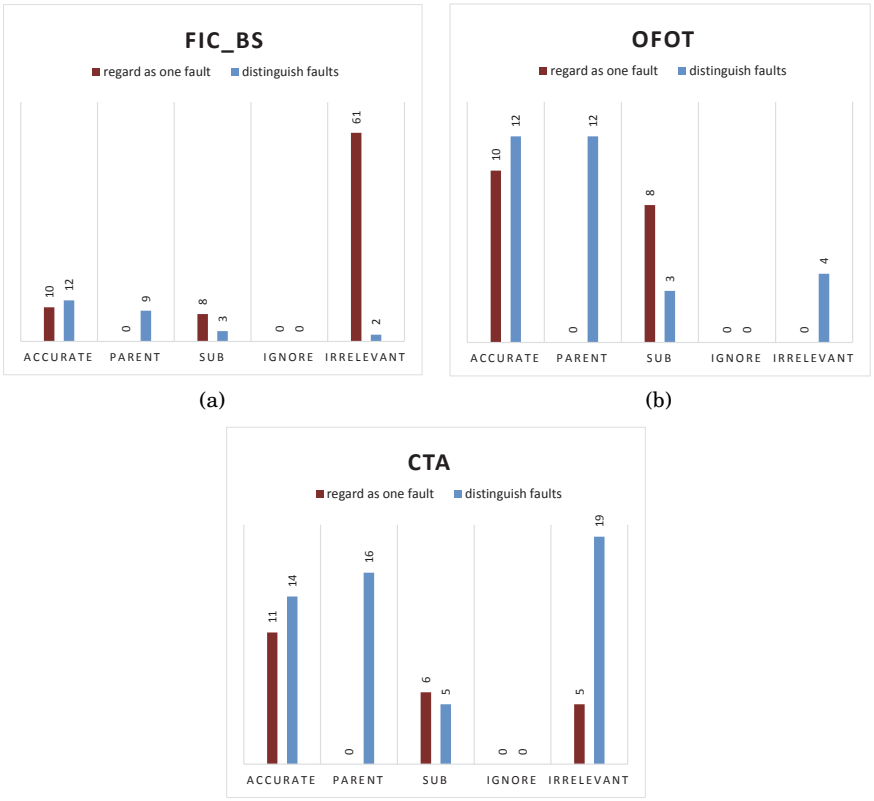

Fig. 2. Results of two strategies for traditional approaches: FIC_BS, OFOT and CTA

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, among which *parent number*, *sub number*,

irrelevant number are 9,3,2 and 0,8,61 respectively. Similar results can be observed 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 latter, while *distinguish faults* strategy get more *parent combinations* than *regard as one* strategy. 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 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* obtained more *irrelevant combinations* (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, 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 they chose are the same, the difference lies at the way they characterizing the failure-inducing combinations in the test cases: OFOT directly identify the parameter according to the passed test cases while CTA used classified tree analysis.

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

5.3. Performance of our approach

The third empirical study aims to observe the performance of our approach and compare it with the result got by the traditional approaches. Our approach augments the three traditional FCI approaches with replacing test cases strategy described in Section 4.

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 augmented ones.

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 of the augmented approaches, which is 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 augmented approaches.

Comparing our approach with two traditional strategies in Figure 3, we observed that there is significant improvement for augmented approaches 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 got 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 augmented approaches also get a good performance at limiting the number of sub combinations and parent combination. In effect, compared with *distinguish faults* which is good at limiting sub combinations while producing more parent combinations and *regard as one fault* which is the other way around, the augmented ones get a more balanced result. Specifically, for instance, in Figure 3(a) for approach FIC_BS,

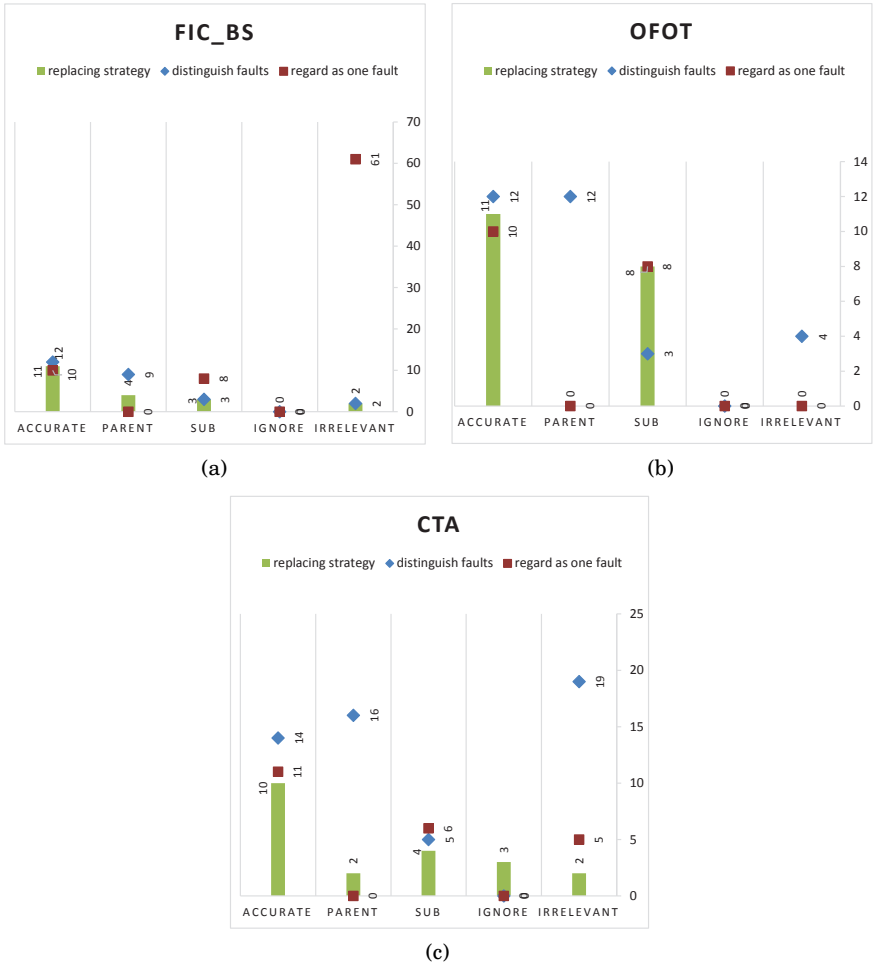

Fig. 3. Three approaches augmented with the replacing strategy

distinguish faults strategy obtained 9 parent combinations while got 3 sub combinations. And for *regard as one fault* strategy, it got none parent combination but attained 8 sub combinations. For the replacing strategy, it only identified 4 parent combinations, which smaller than *distinguish faults* strategy, and obtained 3 sub combinations which is smaller than *regard as one fault* strategy and equal to *distinguish faults* strategy.

Apart from these improvement, there is some slight decline for the augmented approaches in some specific situation. For example, 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 make the FCI approaches, to some extent, get better better performance at identifying failure-inducing combinations when facing masking effect between multiple faults.

5.4. Voting System

The last empirical study aims to observe the performance of our approach and compare it with the result got by the traditional approaches. Our approach augments the three traditional FCI approaches with replacing test cases strategy described in Section 4.

5.4.1. 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 augmented ones.

5.4.2. Result and discussion.

5.5. 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 failure-inducing combinations identifying algorithms, further works needed to examine more algorithms in this filed to get a more general result.

6. RELATED WORKS

Shi and Nie presented a further testing strategy for fault revealing and failure diagnosis[Shi et al. 2005], 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 identification of the specific 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 proposed an AIFL approach which extended the SOFOT process by mutating the changing strength in each iteration of characterizing failure-inducing combinations[Wang et al. 2010].

Nie et al. introduced the notion of Minimal Failure-causing Schema(MFS) and proposed the OFOT approach which extended from SOFOT that can isolate the MFS in SUT[Nie and Leung 2011a]. The approach mutates 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 [Zeller and Hildebrandt 2002] 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. also proposed a similar approach that can identify the failure-inducing combinations that has no overlapped part efficiently [Zhang and Zhang 2011]. Later Li improved the delta-debugging based failure-inducing combination by exploiting the useful information in the executed covering array[Li et al. 2012].

Colbourn and McClary proposed a non-adaptive method [Colbourn and McClary 2008]. Their approach extends the covering array to the locating array to detect and locate interaction faults. C. Martinez 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[Martínez et al. 2008; 2009]. Their algorithms focus on identifying the faulty tuples that have no more than 2 parameters.

Ghandehari et al. defined the suspiciousness of tuple and suspiciousness of the environment of a tuple[Ghandehari et al. 2012]. Based on this, they rank the possible tuples and generate the test configurations. They further utilized the test cases generated from the inducing combination to locate the faults inside the source code[Ghandehari et al. 2013].

Yilmaz proposed a machine learning method to identify inducing combinations from a combinatorial testing set [Yilmaz et al. 2006]. They construct a classified tree to analyze the covering arrays and detect potential faulty combinations. Beside this, Fouché [Fouché et al. 2009] and Shakya [Shakya et al. 2012] made some improvements in identifying failure-inducing combinations based on Yilmaz's work.

Our previous work [Niu et al. 2013] 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.

In addition to 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 [Cohen et al. 2007a; 2007b; 2008] 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. Further study was conducted [Petke et al. 2013]to show the fact that with considering constrains, the higher-strength covering arrays with early fault detection is practical. Besides, additional constraints impacts in CT were studied in works like [Garvin et al. 2011; Bryce and Colbourn 2006; Calvagna and Gargantini 2008; Grindal et al. 2006; Yilmaz 2013].

Chen et al. 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[Chen et al. 2010]. 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 Yilmaz proposed a feedback-driven approach to work around the masking effects[Dumlu et al. 2011]. 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 [Yilmaz et al. 2013], 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 formally analysed the impact of masking effects on FCI approaches and showed that both two traditional

strategies are inefficient in handling such impact. We further presented a divide and conquer strategy for FCI approaches to alleviate this impact.

In the empirical studies, we extended three FCI approaches with our strategy. The comparison between this three traditional approaches and their improved variations was conducted on several open-source software. The results indicated that our strategy do assist traditional FCI approaches in getting 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 misjudgement in the failure-inducing combinations identifying process. At last, as the extent to what the FCI suffers from masking effects varies in different algorithms, the combination of different FCI approaches is desired in the future to further improve the performance for identifying failure-inducing combinations for multiple faults.

8. TYPICAL REFERENCES IN NEW ACM REFERENCE FORMAT

APPENDIX

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Maura Turolla of Telecom Italia for providing specifications about the application scenario.

REFERENCES

James Bach and Patrick Schroeder. 2004. Pairwise testing: A best practice that isnt. In *Proceedings of 22nd Pacific Northwest Software Quality Conference*. Citeseer, 180–196.

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

Renée C Bryce, Charles J Colbourn, and Myra B Cohen. 2005. A framework of greedy methods for constructing interaction test suites. In *Proceedings of the 27th international conference on Software engineering*. ACM, 146–155.

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

Baiqiang Chen, Jun Yan, and Jian Zhang. 2010. Combinatorial testing with shielding parameters. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. IEEE, 280–289.

David M. Cohen, Siddhartha R. Dalal, Michael L Fredman, and Gardner C. Patton. 1997. The AETG system: An approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on* 23, 7 (1997), 437–444.

Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2007a. Exploiting constraint solving history to construct interaction test suites. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007*. IEEE, 121–132.

Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2007b. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 129–139.

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

Myra B Cohen, Peter B Gibbons, Warwick B Mugridge, and Charles J Colbourn. 2003. Constructing test suites for interaction testing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 38–48.

Charles J Colbourn and Daniel W McClary. 2008. Locating and detecting arrays for interaction faults. *Journal of combinatorial optimization* 15, 1 (2008), 17–48.

Emine Dumlu, Cemal Yilmaz, Myra B Cohen, and Adam Porter. 2011. Feedback driven adaptive combinatorial testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 243–253.

Sandro Fouché, Myra B Cohen, and Adam Porter. 2009. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 177–188.

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

Laleh Sh Ghandehari, Yu Lei, David Kung, Raghu Kacker, and Richard Kuhn. 2013. Fault localization based on failure-inducing combinations. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 168–177.

Laleh Shikh Gholamhossein Ghandehari, Yu Lei, Tao Xie, Richard Kuhn, and Raghu Kacker. 2012. Identifying failure-inducing combinations in a combinatorial test set. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 370–379.

Mats Grindal, Jeff Offutt, and Jonas Mellin. 2006. Handling constraints in the input space when using combination strategies for software testing. (2006).

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA Data Mining Software: An Update; SIGKDD Explorations, Volume 11, Issue 1. (2009).

Jie Li, Changhai Nie, and Yu Lei. 2012. Improved Delta Debugging Based on Combinatorial Testing. In *Quality Software (QSIC), 2012 12th International Conference on*. IEEE, 102–105.

Conrado Martínez, Lucia Moura, Daniel Panario, and Brett Stevens. 2008. Algorithms to locate errors using covering arrays. In *LATIN 2008: Theoretical Informatics*. Springer, 504–519.

Conrado Martínez, Lucia Moura, Daniel Panario, and Brett Stevens. 2009. Locating errors using ELAs, covering arrays, and adaptive testing algorithms. *SIAM Journal on Discrete Mathematics* 23, 4 (2009), 1776–1799.

Changhai Nie and Hareton Leung. 2011a. The minimal failure-causing schema of combinatorial testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 4 (2011), 15.

Changhai Nie and Hareton Leung. 2011b. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 11.

Xintao Niu, Changhai Nie, Yu Lei, and Alvin TS Chan. 2013. Identifying Failure-Inducing Combinations Using Tuple Relationship. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 271–280.

Justyna Petke, Shin Yoo, Myra B Cohen, and Mark Harman. 2013. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 26–36.

Kiran Shakya, Tao Xie, Nuo Li, Yu Lei, Raghu Kacker, and Richard Kuhn. 2012. 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*. IEEE, 620–623.

Liang Shi, Changhai Nie, and Baowen Xu. 2005. A software debugging method based on pairwise testing. In *Computational Science–ICCS 2005*. Springer, 1088–1091.

Charles Song, Adam Porter, and Jeffrey S Foster. 2012. iTree: Efficiently discovering high-coverage configurations using interaction trees. In *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 903–913.

Ziyuan Wang, Baowen Xu, Lin Chen, and Lei Xu. 2010. Adaptive interaction fault location based on combinatorial testing. In *Quality Software (QSIC), 2010 10th International Conference on*. IEEE, 495–502.

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

Cemal Yilmaz, Myra B Cohen, and Adam A Porter. 2006. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on* 32, 1 (2006), 20–34.

Cemal Yilmaz, Emine Dumlu, M Cohen, and Adam Porter. 2013. Reducing Masking Effects in Combinatorial Interaction Testing: A Feedback Driven Adaptive Approach. (2013).

Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on* 28, 2 (2002), 183–200.

Zhiqiang Zhang and Jian Zhang. 2011. Characterizing failure-causing parameter interactions by adaptive testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 331–341.

Received February 2007; revised March 2009; accepted June 2009

Online Appendix to: Identify minimal failure-causing schemas for multiple faults

XINTAO NIU and CHANGHAI NIE, State Key Laboratory for Novel Software Technology, Nanjing University
HARETON LEUNG, Hong Kong Polytechnic University

A. THIS IS AN EXAMPLE OF APPENDIX SECTION HEAD

B. APPENDIX SECTION HEAD

The primary consumer of energy in WSNs is idle listening. The key to reduce idle listening is executing low duty-cycle on nodes. Two primary approaches are considered in controlling duty-cycles in the MAC layer.

DOI: http://dx.doi.org/10.1145/0000000.0000000