Mutant Quality Indicators

Mike Papadakis
University of Luxembourg
michail.papadakis@uni.lu

Thierry Titcheu Chekam
University of Luxembourg
Thierry.Titcheu-Chekam@uni.lu

Yves Le Traon
University of Luxembourg
Yves.LeTraon@uni.lu

***Abstract*—The question of which are the valuable mutants has received little attention in mutation testing literature. Naturally, the choice of mutants impacts the quality of the performed analysis and has the potential of changing the conclusions of empirical studies. To this end, we collect definitions related to mutant quality indicators and analyses their relations. We identify two classes of indicators, related to individual mutants (Fault Revealing, Subsuming, Hard-to-kill and Stubborn) and to mutant sets (disjoint/dominator and distinguished). We analyse a large set of mutants from 3,902 (real) faulty program versions, belonging to 40 fault classes, collected from an on-line programming contest. Our analysis categorises mutants as valuable, according to the studied quality indicators, profiles their types and examines the relations between them. Our results suggest that there is a large disagreement between the indicators and that the connection between mutant type, its quality and its ability to reveal faults is weak. Additionally, our paper reveal that the ability of mutants to uncover faults differs significantly across the different fault classes and that some mutant types are well linked (or completely disconnected) to specific fault classes.**

I. Introduction

The question of what constitutes a good mutant remains controversial and unknown. Naturally, in mutation testing the 'quality' of mutants plays a central role and can have major implications on the performed analysis. For instance, empirical studies may come to biased conclusions if they use all available mutants [1]. Similarly, the use of restrictive mutant sets may result in a much lower strength testing [2].

Previous research has investigated this problem by examining the types of mutants. Mutants of specific types are considered as more important than others as they encode test requirements not captured by other mutant types [3]. Apart from the types of mutants, some early studies set specific 'quality' criteria to judge mutants' quality. Thus, they suggested that mutants quality should be measured through the 'easiness', ratio of valid program inputs that kill the mutant, of killing them. The underlying idea is that easy to kill mutants (killed by most of the test inputs) are not of a particular value.

Other studies suggested that quality mutants are those that are stubborn, i.e., resistant to killing by the test cases that execute them, [4]. Thus, mutants that are hard to infect or propagate, i.e., killed by few test cases that execute them are valuable. The reason is that these mutants go beyond coverage, i.e., the mutants are not killed by coverage-based test cases [5].

Another way to define mutants' quality is based their diversity wrt to the program input domain. This way good mutants are a subset that is defined wrt to a reference set of all mutants. Thus, good mutants are those that are killed by different test cases than other mutants. This means that the selected set of mutants is as much disjoint, in terms of their killing condition, as possible [6]. In other words, disjoint mutants have a minimum overlap between the mutants' killings. The underlying idea is that a disjoint mutant set should be representative of all mutants, i.e., their killing must result in the killing of all mutants, and at the same time they are the set of the harder to kill than any other alternative set of mutants.

A related, newly suggested indicator is that quality mutants are those that guide testers towards revealing real faults [7]. The underlying idea is that good mutants should lead to test cases that reveal frequent real faults. Thus, instead of covering the whole spectrum of mutants one should cover the mutants that are most likely to be linked with faults.

Given the plethora of the mutant quality indicators, a natural question to ask is whether there are important differences between them and what are the links with fault revelation. In other words, we are interested to see if the indicators agree on which are the valuable mutants and whether these mutants are linked with fault revelation. Answering these questions is important in order to direct future research and increase the understanding of the mutation testing foundations.

In this paper we study the relatively differences between the quality indicators. We investigate the types of mutants that they involve and explore the link between different mutant types and different fault classes. We find that all quality indicators identify only a few,less than 10%, mutants as good. We also find that there is a large disagreement, between the indicators, on which are the good mutants. In particular we find that 39%, 42% and 6% of the fault revealing mutants are also subsuming, hard-to-kill and hard-to-propagate, while 17%, 60% and 4% of the subsuming mutants are fault revealing, hard-to-kill and hard-to-propagate.

Perhaps more importantly, we find a weak connection between fault revelation and quality indicators, suggesting the need for specialized approaches targeting the particular class of fault revealing mutants. We also show that the link between mutants and faults differs significantly across fault classes. We demonstrate that almost half of the faults related to missing code are weakly linked with mutants, while 90% of the faults related to OAAN category are strongly linked with *SCALAR.BINARY* mutants. These results suggest that future studies should consider the particular classes of faults targeted by the proposed approaches. Overall, our study increases the understanding on what contributes to the mutants' quality and opens several directions for future research.

II. Mutant Quality Indicators

We performed an expert literature review by considering the papers collected in the recent survey of mutation testing [8]. Our analysis revealed the following two classes of indicators:

A. Unit-based MQIs

Fault Revealing (F.R.) are the mutants that are killed only by test cases that reveal a fault. These mutants are the ones that are linked with fault revelation. Thus, one should cover only the mutants that are most likely to be linked with frequently occuring (real) faults [7].

We consider two classes of fault revealing the mutants, denoted as "F.R.-1.0" and "F.R.-0.9". The first class has the mutants that are killed only by fault revealing test cases, while the second class involves the mutants that are killed by at least 90% of the fault revealing tests.

Subsumming mutants are defined based on the subsumption relation and the indistinguished mutants. According to Ammann et al. [9] "one mutant subsumes another if at least one test kills the first and every test that kills the first also kills the second". Indistinguished are two mutants that are always killed by the same tests.

Subsuming are the mutants that are subsumed only by indistinguished mutants. We consider as subsuming, all mutants that are in the leaf nodes of the mutant subsumption graphs [10], built based on the employed test suites.

Hard-to-kill (Hard) are the mutants that are killed only by a small fraction of test cases. We consider two classes of hard-to-kill mutants. Those that are killed by at most 5% and 2.5% of the available test suites. We denote these as "Hard-0.050" and "Hard-0.025".

Another way to define the hardness to kill is based on the RIP model [11]. Thus, hardness can be defined as hardness to reach, infect and propagate. Here, we only consider mutants that are hard to propagate.

Hard-to-propagate (HardP) are the mutants that are killed only by a small fraction of test cases that infect them. We consider two classes of hard-to-propagate, those that are killed by at most 25% and 10% of the test cases that infect them. We denote these as "HardP-0.25" and "HardP-0.10".

B. Set-based MQIs

Non-duplicated is the set of mutants that has no indistinguished mutants. We approximate mutant duplication based on the available test suites.

Disjoint/Surface - Minimal/Dominator. is the subset of mutants with the minimum number of subsuming mutants. Conceptually, there are no difference between disjoint/surface and minimal/dominator mutants. The actual differences are thin and are due to the selection procedure. Disjoint mutants are a subset with minimum joint killings, approximated through a greedy heuristic [1], [6]. Surface mutants [12] are also approximated by a similar heuristic. The minimal/dominator mutants form the actual minimal subset, selected bases on a systematic procedure [10].

TABLE I: Fault Classes

AST Type	Fault Class	Example
Higher Order	Expression HEXP	$\ominus\ if(C) \quad \oplus\ if(C \Vert \mathbf{D})$
	Non-branch Stmt HDMS	$\ominus\ printf(s); \quad \ominus\ x = y + 3;$
	Combination HCOM	$\ominus\ rep(i, n) \quad \oplus\ for(...)$
	Non-branch Stmt HIMS	$\oplus\ printf(s); \quad \oplus\ x = y + 3;$
	Branch Stmt HBRN	$\oplus\ if(C)\ \{printf(s);\}$
	Others HOTH	$\ominus\ g(0); \quad \oplus\ for(...)\{f(i);\}$
Statement	Function Call SISF	$\oplus\ scanf(``\%d'', \&n);$
	Type STYP	$\ominus\ \mathbf{int}\ a \quad \oplus\ \mathbf{long}\ a$
	Control Flow SRIF	$\ominus\ if(a > b) \quad \oplus\ if(\mathbf{g}(a) > b)$
	Data Flow SISA	$\oplus\ t = 0$
	Move SMOV	$\ominus\ f(x); \quad \mathrm{g(x);} \quad \oplus\ f(x);$
Operand	Variable DRWV	$\ominus\ \mathbf{b} = 0; \quad \oplus\ \mathbf{a} = 0;$
	Array DCCA	$\ominus\ int\ x[\mathbf{2}]; \quad \oplus\ int\ x[\mathbf{20}];$
	Variable DRVA	$\ominus\ if(\mathbf{i} > 0) \quad \oplus\ if(\mathbf{k} > 0)$
	Constant DCCR	$\ominus\ if(x > \mathbf{4}) \quad \oplus\ if(x > \mathbf{3})$
Operator	Control Flow ORRN	$\ominus\ if(a > 0) \quad \oplus\ if(a >= 0)$
	Arithmetic OAAN	$\ominus\ v2\ \mathbf{-=}\ 2 \quad \oplus\ v2\ \mathbf{+=}\ 2$
	Function Call OFPF	$\ominus\ f(``\%\mathbf{d}'', i); \quad \oplus\ f(``\%\mathbf{ld}'', i);$
	Control Flow OILN	$\ominus\ if(x) \quad \oplus\ if(x\mathbf{\&\&f(x)})$
	Arithmetic OAIS	$\ominus\ x\ \mathrm{+=}\ y \quad \oplus\ x\ \mathrm{+=}\ y\ \mathbf{/\ 2}$

TABLE II: Mutant Types

Mutated Instruction	Original Instruction Type	Mutated Instruction Type
STATEMENT	WHOLE STMT	TRAPSTMT
	WHOLE STMT	DELSTMT
	CALL STATEMENT	SHUFFLEARGS
	SWITCH STATEMENT	SHUFFLECASESDESTS
	SWITCH STATEMENT	REMOVECASES
EXPRESSION	SCALAR.ATOM	UNARY
	SCALAR.BINARY	BINARY
	SCALAR.BINARY	UNARY
	SCALAR.ATOM	BINARY
	SCALAR.BINARY	TRAPSTMT
	POINTER.BINARY	BINARY
	SCALAR.BINARY	DELSTMT
	DEREFERENCE.BINARY	BINARY
	SCALAR.UNARY	UNARY
	POINTER.BINARY	UNARY
	DEREFERENCE.BINARY	UNARY
	POINTER.ATOM	UNARY
	POINTER.UNARY	UNARY

The set-based indicators depend on the individual choice of the individual mutants and cannot be compared with the unit-based ones. Their relations are also well understood and thus, in the rest of the paper we mainly focus on the unit-based.

III. Experiment Setup

A. Programs and Faults

We used the Codeflaws benchmark [13] that involves programs selected from an on-line programming contests[1]. In Codeflaws, every faulty program version is unique and has two instances, the 'faulty' and the 'fixed' one. The former regards the rejected, while the later the accepted submission. In total, Codeflaws contains 3,902 faults of 40 defect classes. These programs are of 1 to 322 lines of code and are accompanied by a test suite that was used to test and judge the programs as faulty and fixed. We choose Codeflaws because it contains many, diverse, relatively hard to expose faults.

[1]http://codeforces.com/

To conduct a valid experimentaion, we augment the available test suites using KLEE [14], a state-of-the-art test generation tool. Although, these test suites greatly increased the cost of our experiment, we considered their use of vital importance as otherwise our results could be subject to “noise effects” [15]. Overall, our experiment involved 122,261 test cases, 3,213,543 mutants, whose execution required a total of 8,009 CPU days of computation.

We aim at investigating the link between mutants and faults. Thus, we consider important to focus at hard faults, i.e., fault not revealed by every test case. In total, approximately half of our faults are trivial ones (revealed by a large fraction of test case). Thus, we restrict our analysis on the 1,629 faults that are revealed by less than 25% of the test cases involved.

Table I records the main fault classes in the dataset. It is noted that these 20 classes involve more than 10 fault instances that are revealed by less than 25% of the tests in our test suites. Following the classification scheme of Codeflaws the faults fall into 4 categories. The fault classes are related to faults in operators of expressions, faults in operands of expressions, faults in control-flow related statements (e.g., missing if conditional) and faults in function call statements or other statements. The last column of Table I records examples of the fault classes (taken from [13]). These demonstrate the way the example faults were patched, i.e., ”−” denotes the statement(s) deleted/modified and ”+” the statements added in order to fix the fault.

B. Automated Tools

We used KLEE to perform test augmentation with the following settings: a two hours time limit per program, a Random Path search strategy, Randomize Fork Enabled, Max Memory 2048, Symbolic Array Size 4096, Symbolic Standard input size 20 and Max Instruction Time of 30 seconds. This resulted in 26,229 test cases. Since the automatically generated test cases do not include any test oracle, we used the programs' fixed version as oracle. Thus, we considered as failing, every test case that resulted in different observable output when executed in the ‘faulty’ than in the ‘fixed’ program. Similarly, we identified the killed mutants using the program output.

We built a mutation testing tool that operates on LLVM bitcode. Actually all our metrics and analysis were performed on the LLVM bitcode. Our tool implements 18 operators, composed of 816 transformation rules. These include all those that are supported by modern mutation testing tools [3]. We thus, used the 18 operators recorded on Table II.

To reduce the influence of redundant and equivalent mutants, we applied TCE [16]. Since we operate on LLVM bitcode we compared the mutated optimized llvm codes using the llvm-diff utility. llvm-diff is a tool like the known Unix diff utility but for llvm bitcode. TCE Detected 523,097 and 934,415 equivalent and redundant mutants.

C. Experimental Procedure

We start our analysis by forming a pool of all availale test cases. We then constructed a mutation-fault matrix that records the mutants killed and faults revealed by each one of the available test cases. We then applied mutation on the faulty program versions so that we are faithful to real settings and avoid making the Clean Program Assumption [15]. We used this matrix to categorize the mutants.

In summary, we form the population of all mutants, identify the mutants' categores and analyse their relations. We consider the relations between the indicators wrt to all killable mutants and to mutants of the same type. For the different fault classes, we follow the taxonomy adopted by Codeflaws [13]. Details about the considered fault classes and mutant types are recorded on Tables I and II.

TABLE III: Prevalence of mutant categories.

Category	No. Mutants	Ratio	Program Av.
Fault Revealing (FR-1.0)	44,221	3%	27
Fault Revealing (FR-0.9)	65,809	4%	40
Subsuming	98,709	6%	61
Hard-to-kill (Hard-0.050)	111,442	7%	68
Hard-to-kill (Hard-0.025)	45,286	3%	28
Hard-to-propagate (HardP-0.25)	325,158	21%	200
Hard-to-propagate (HardP-0.10)	137,864	9%	85
Non-Duplicated	321,822	21%	80
Disjoint/Dominator	20,182	1%	12

IV. Results

A. Prevalence of mutant quality indicator categories

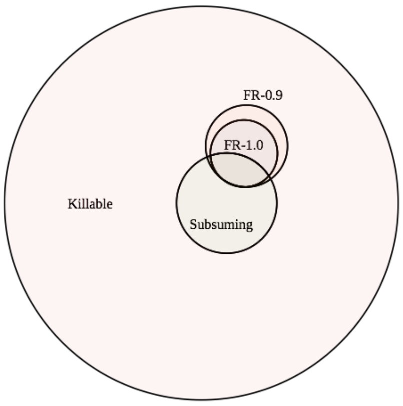
We start our analysis by measuring the prevalence of the mutants that are characterized as good by the studied quality indicators. Table III records the total number of mutants involved, the ratio and average (per program) number of them, per considered indicators. Interestingly, we can observe that only a small fraction of all mutants (less than 10%) is characterized as good, according to all categories (the only exception is the HardP-0.25).

This finding suggests that the great majority of the mutants are not good and may have undesirable effects on the interpretation of the mutation score. Thus, it is likely that one can achieve a good mutation score by simply killing bad mutants and not the good ones. Unfortunately, this fact can have serious implications on the confidence inspired by mutation testing [1]. Therefore, a first finding is that the majority of the mutants are bad ones according to every quality indicator.

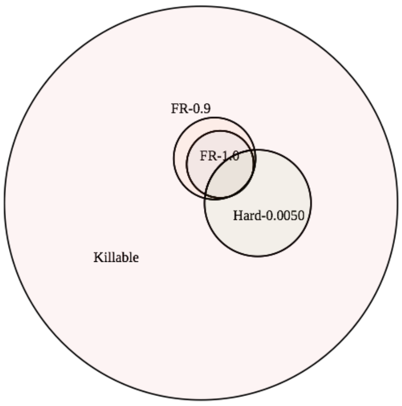
B. Relations between mutant quality indicators

Up to this point, our analysis has shown that few mutants are characterized as good by every quality indicator. However, we have seen nothing about the relations between the different categories of the good mutants. In other words we would like to see whether the indicators agree between themselves on which are the good mutants and which are not.

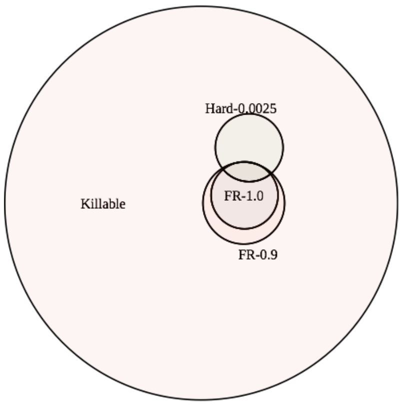
To investigate this issue we explore the geography of the mutants' population. Thus, we characterize every mutant according to the studied indicators and measure the number of them that belong on the same and different categories. We present these results in a pairwise manner in Figure 1. In these diagrams the surface represent the number of mutants that



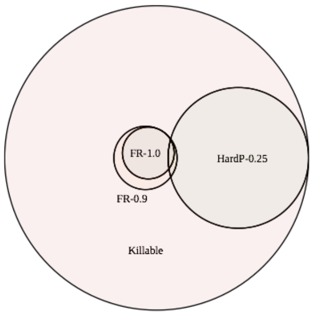
(a) Fault Revealing VS. Subsuming



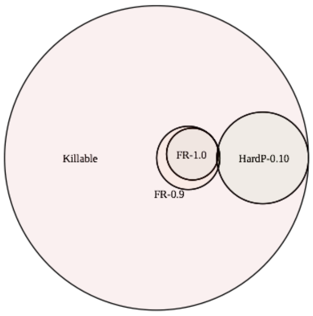
(b) Fault Revealing VS. Hard-to-kill (5%)



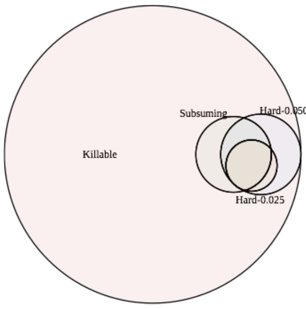
(c) Fault Revealing VS. Hard-to-killl (2.5%)



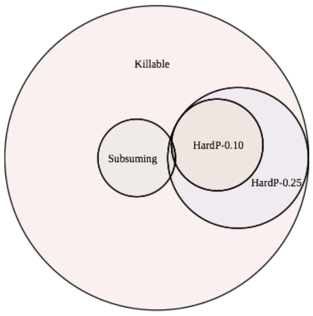
(d) Fault Revealing VS. Hard-to-prop. (25%)



(e) Fault Revealing VS. Hard-to-prop. (10%)

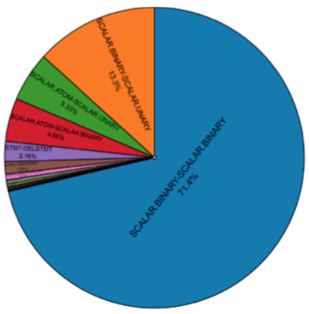


(f) Hard-to-kill VS. Subsuming

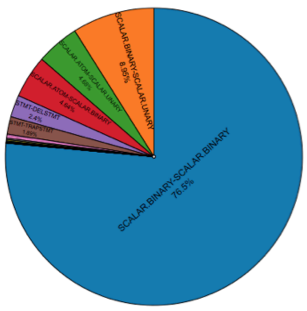


(g) Hard-to-prop. VS. Subsuming

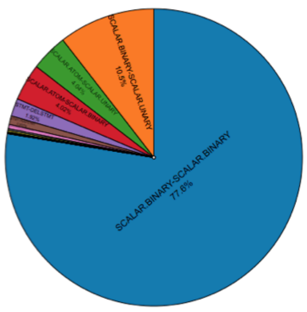
Fig. 1: Relations between different mutant quality indicators



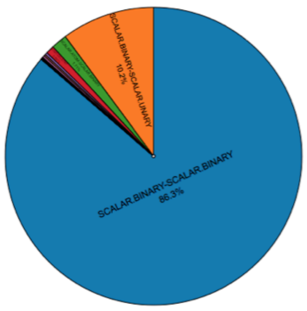
(a) Fault Revealing mutants



(b) Subsuming mutants



(c) Hard-to-kill mutants (5%)

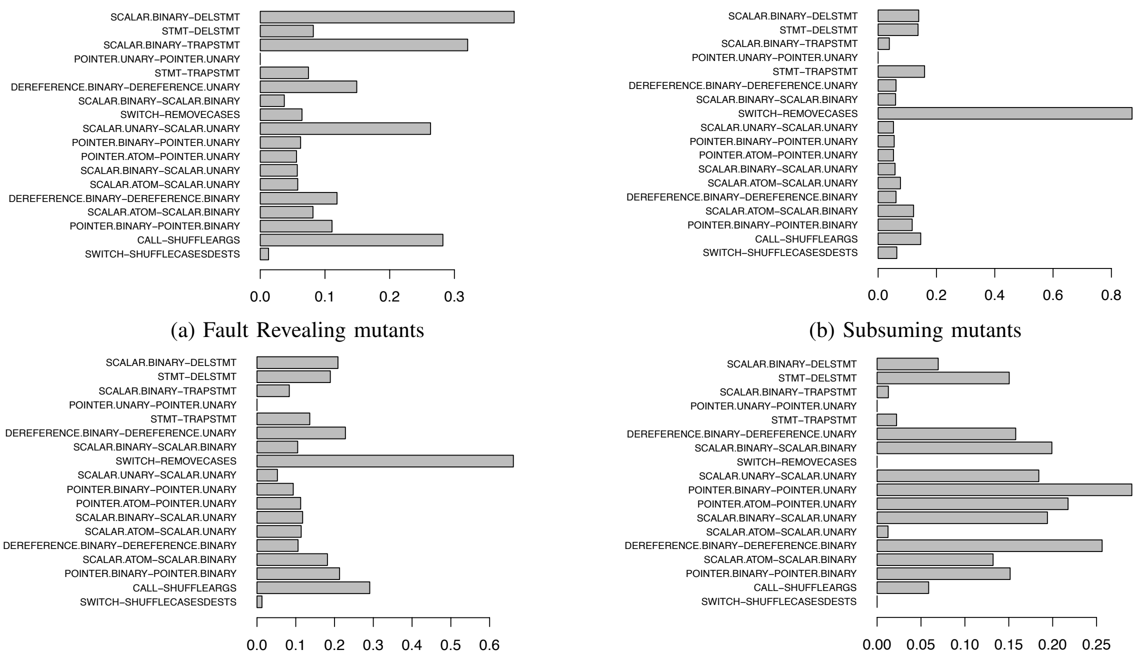


(d) Hard-to-prop. mutants (10%)

Fig. 2: Types of mutants involved in the mutant quality indicator categories

belong to each category. The surfaces have been scaled so that they reflect the actual size relation between the different categories. Thus, we can see that FR-1.0 are less (in number) than the subsuming mutants.

A first observation from Figure 1 is that there is a large disagreement, between the indicators, on which are good mutants. In particular we observe that hard-to-propagate mutants is a distinct category, i.e., it has a very small overlap with every other category. We also observe a medium to small overlap of fault revealing with the subsuming and hard-to-kill mutants. Interestingly when relaxing the fault revealing probability to 90% (FR-0.9) results in a movement away from the subsuming or hard-to-kill mutants. These results suggests that not all the mutants are linked to faults. As subsuming mutants represent the whole spectrum of mutants they include many that are not linked with faults. On the contrary fault revealing ones belong to those parts of the spectrum that are linked with the faults and overall these two categories are not the same. Thus, future research should devise techniques to specialize mutants to the targeted domain or faults.



(a) Fault Revealing mutants

(b) Subsuming mutants

(c) Hard-to-kill mutants (5%)

(d) Hard-to-prop. mutants (10%)

Fig. 3: Ratio of mutants involved in quality indicator categories per mutant type

Another interesting result is that hard-to-kill mutants have a large overlap with subsuming mutants. Still they are not the same, but a large proportion of them is included. We continue our analysis by presenting the exact relations in terms of percentages, measured wrt to each category.

Fault Revealing VS. Subsuming: Our results suggests that 39% and 27% of the fault revealing mutants, with probability equal to 1.0 and 0.9, are also subsuming. Interestingly, 17% and 18% of the subsuming mutants are also fault revealing (with probability equal to 1.0 and 0.9). This results suggest that only a few of the subsuming mutants are linked with the faults and that more than half of all the mutants that are linked with the faults are subsumed (not subsuming!).

Fault Revealing VS. Hard-to-kill: We find that 42% and 29% of the fault revealing mutants, with probability equal to 1.0 and 0.9, are also Hard-to-kill (killed with less than 5% of the tests). On the other side, 16% and 17% of the Hard-to-kill mutants (killed with less than 5% of the tests) are also fault revealing (with probability equal to 1.0 and 0.9). This result suggests that there is a (slightly) stronger link between faults and hard-to-kill than faults and subsuming mutants.

When consider stronger mutants, killed by less than 2.5% of the tests, we find that 18% and 12% of the fault revealing mutants, with probability equal to 1.0 and 0.9, are also Hard-to-kill (killed with less than 2.5% of the tests). Considering the inverse relation we find that 18% and 18% of the Hard-to-kill mutants (killed with less than 2.5% of the tests) are also fault revealing (with probability equal to 1.0 and 0.9). This suggests that stronger mutants have a weaken link with the faults than less strong ones.

Fault Revealing VS. Hard-to-propagate: 6% and 7% of the fault revealing mutants, with probability equal to 1.0 and 0.9, are also Hard-to-propagate (killed by less than 25% of the tests that infect the mutant) and 1% and 1% of the Hard-to-propagate mutants (killed by less than 25% of the tests that infect the mutant)are also fault revealing (with probability equal to 1.0 and 0.9). When consider stronger mutants, 1% and 1% of the fault revealing mutants, with probability equal to 1.0 and 0.9, are also Hard-to-propagate (killed by less than 10% of the tests that infect the mutant). The inverse relation shows that $\approx 0\%$ and 1% of the Hard-to-propagate mutants (killed by less than 10% of the tests that infect the mutant) are also fault revealing (with probability equal to 1.0 and 0.9). These show that a very weak link exists between the faults and hard-to-propagate mutants.

Subsuming VS. Hard-to-kill: 60% and 38% of the subsuming mutants are also Hard-to-kill (killed with less than 5% and 2.5% of the tests) and 53% and 83% of the Hard-to-kill mutants (killed with less than 5% and 2.5% of the tests) are also subsuming. This relation shows the relatively strong link between the subsuming and hard-to-kill mutants.

Subsuming VS. Hard-to-propagate: 4% and 1% of the subsuming mutants are also Hard-to-propagate (killed by less than 25% and 10% of the tests that infect the mutant) and 1% and 1% of the Hard-to-propagate mutants (killed by less than 25% and 10% of the tests that infect the mutant) are also subsuming. This relation shows the weak link between hard-to-propagate mutants and other categories.

Hard-to-kill VS. Hard-to-propagate: 1% and $\approx 0\%$ of the Hard-to-kill mutants, with probability equal to 1.0 and 0.9, are also Hard-to-propagate (killed by less than 25% of the tests that infect the mutant) and $\approx 0\%$ and $\approx 0\%$ of the Hard-to-propagate mutants (killed by less than 25% of the tests that infect the mutant) are also Hard-to-kill (with probability equal to 1.0 and 0.9). When consider stronger mutants, 0% and 0% of the Hard-to-kill mutants, with probability equal to 1.0 and 0.9, are also Hard-to-propagate (killed by less than 10% of the tests that infect the mutant). 0% and 0% of the Hard-to-propagate mutants (killed by less than 10% of the tests that infect the mutant) are also Hard-to-kill (with probability equal to 1.0 and 0.9). This relation also shows the weak link between hard-to-propagate mutants and other categories.

C. Mutant types and quality indicators

We investigate the link between mutant type (characterized by its syntactic transformation) and quality indicators by checking whether there are types of mutants that are more likely to generate good mutants. We thus, check the types of mutants involved in every category, i.e., the ratio of the good mutants that are of each type. We also measure the ratio of the good mutants among those generated per considered mutant type. The former case shows the types of mutants composing the good ones, while the later shows whether the type of mutants relates to the good ones.

Figure 2 presents the types of mutants involved in the studied categories. For simplicity we have omitted the results of Hard-to-kill-0.025 and Hard-to-propagate-0.10, as they are quite similar to those of Hard-to-kill-0.050 and Hard-to-propagate-0.25. Interestingly, the majority of the mutants are of the same types (the top 4 most prevalent types are the same). Thus, the types of *SCALAR.BINARY-SCALAR.BINARY*, *SCALAR.BINARY-SCALAR.UNARY*, *SCALAR.ATOM-SCALAR.UNARY* and *SCALAR.ATOM-SCALAR.BINARY* cover more than 80% of all the good mutants. However, this is due to the number of mutants that are generated by these operators.

The graphs of Figure **??** record the ratios of mutants (of the same type) involved in the studied categories for every consider mutant type. Interestingly, the profiles of the four categories differ significantly. This shows (again) that the indicators disagree between them and characterize different mutants (and different types) as good ones. Interestingly, all types of mutants (with one exception) contributes to all the categories, indicating that all of them are of a value.

We also observe that with a few exceptions the mutant type does not seem to mater much on any category. Regarding the fault revealing mutants, 5 types seems to generate larger proportions of good mutants than the other 12, but overall all types have a similar ratio. Subsuming and hard-to-kill mutants have one type, the *SWITCHREMOVECASES*, which generates a significantly higher ratio of good mutants. However, beside this type all other mutant types generate similar ratios. The case of hard-to-propagate mutants is a bit different as it involves 6 types with a rather low contribution, while the rest 12 types have similar ratios. Overall, by comparing the results of Figures 2 and **??** we see that some (few) mutant types are more important than others but overall, all mutant types are important.

D. Fault classes with no fault revealing mutants

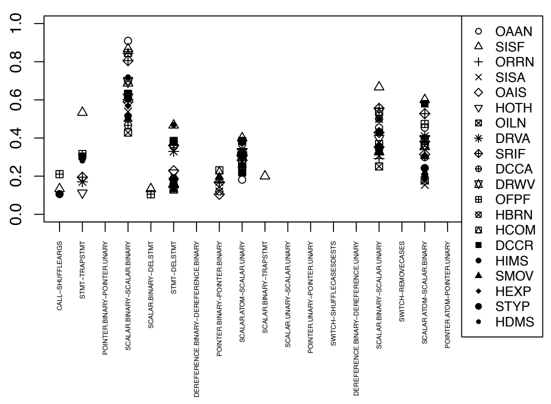
Having investigated the link between mutant type and quality indicators, we turn our analysis on the different fault classes. We thus, investigate which types of faults have no fault revealing mutants. This is important as these cases are faults that are likely to be missed by mutation testing. Overall, in our dataset, we have 462 faulty program versions without any fault revealing mutant. This number accounts for 12% of the faults in our dataset, which is in line with the 13% reported by the literature [15]. To investigate whether there is any link between fault class and absence of fault revealing mutants we report the percentage of faults (per class of faults) without fault revealing mutants. In order to avoid coincidental results we removed from our dataset every faulty class that includes less than 10 fault instances. This resulted in 20 faulty classes, out of the 40 faulty classes included in the dataset.

Figure 5 reports the percentages of the faulty program versions (of the same fault class) with no fault revealing mutants. From these results we observe that 5 classes have a relatively low ratio (with less than 15%) of cases with no fault revealing mutants. 12 classes have ratios between 15%-35%, while 3 classes have a relatively large number of faults (without fault revealing mutants). Thus, we can conclude that mutation is not particularly good at detecting faults of these three classes, (*Operator-ControlFlow-OILN*, *Operand-Array-DCCA* and *HigherOrder-Expression-HEXP*). Interestingly, the OILN and HEXP classes are faults belonging to the general category of omission faults, i.e., faults due to missing code. Omission faults form a known weaknes of code-based techniques [17] and thus, having a strong link with more than half of them is important. The other problematic category is the DCCA class that regards the size of arrays indicating the need for mutation operators related to these faults.

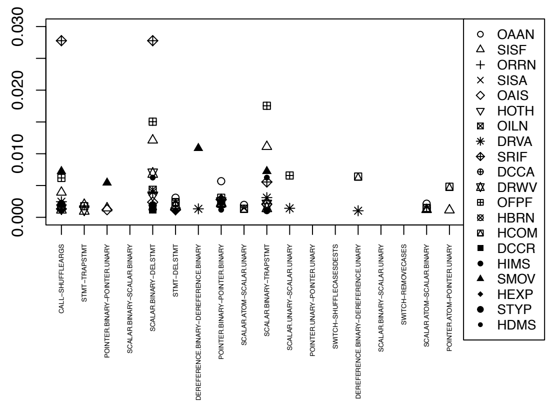
E. Links between mutant types and fault classes

To investigate the link between mutant types and fault classes, we measure, for every considered class, the ratio of faulty programs with fault revealing mutants. Thus, we expect a high ratio when there is a strong link, and a low ratio when there is a weak link. Since every type of mutants involve different number of mutant instances, we also normalize our results with respect to the number of mutants involved. This way we can see whether there are significant differences between the pairs of mutant types and fault classes.

Figure 4 presents the ratios of faulty versions with fault revealing mutants for all pairs of mutant type and fault classes. From these plots we can see that some mutant types (*SCALAR.BINARY SCALAR.BINARY*) are linked with specific fault classes (OAAN), while some mutant types (*POINTER.ATOM POINTER.UNARY*) are linked with many fault classes. This suggests that for specific cases there is a strong link between mutant type and revealed class of fault. When normalizing with respect to the number of mutants the link is less clear, but strong for specific pairs, such as the mutant types *CALL SHUFFLEARGS* and *SCALAR.BINARY DELSTMT* with fault class OAAN.



(a) Ratio of faults for every fault class



(b) Normalized ratio of faults for every fault class

Fig. 4: Ratio of faulty versions with fault revealing mutants (among all faults of the same type) per fault class and mutant type

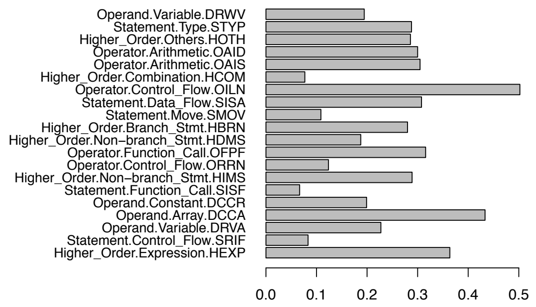


Fig. 5: Faulty versions without fault revealing mutants (ratios)

In conclusion, our results suggests that considering fault classes is important as every class is linked with different mutant types. The differences between mutant types and fault classes provide further evidence that all types of mutants are needed and that there is no dominant type of mutants. Perhaps more importantly, our results reveal that experimental results need to be validated with a diverse class of faults, each one of which should be separately be considered.

V. Conclusion

In this paper we studied the relatively differences between the mutant quality indicators. We found that all indicators identify only a few (less than 10%) mutants as good and that there is no consensus on which mutants should be considered as good. We also found that all mutant types generate valuable mutants, fact indicating that all types of mutant should be used. Overall, we find that some isolated mutant types contribute more on the good mutants, the general trend is that the discriminative power of the mutant type is limited. Perhaps more importantly, we find a weak connection between the fault revelation and quality indicators, suggesting the need for specialized approaches targeting the particular class of fault revealing mutants. Finally, our results demonstrate that the fault revelation ability of mutants differs significantly across the studied classes of faults, indicating that future studies should consider the particular fault classes they target and are involved in the experimental datasets.

References

[1] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. L. Traon, "Threats to the validity of mutation-based test assessment," in *ISSTA 2016*, 2016, pp. 354–365.

[2] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque, "Assessing and improving the mutation testing practice of PIT," in *ICST 2017*, pp. 430–435.

[3] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. on Soft. Eng. & Meth.*, vol. 5, no. 2, pp. 99–118, April 1996.

[4] R. M. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Softw. Test., Verif. Reliab.*, vol. 9, no. 4, pp. 233–262, 1999.

[5] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *ICSE 2014*, pp. 919–930.

[6] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *APSEC*, 2010, pp. 300–309.

[7] T. T. Chekam, M. Papadakis, T. F. Bissyandé, and Y. L. Traon, "Selecting the fault revealing mutants," in *ICSE 2018*.

[8] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation testing advances: An analysis and survey," *Advances in Computers*, accepted for publication.

[9] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *ICST 2014*, pp. 21–30.

[10] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *Mutation*, 2014, pp. 176–185.

[11] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[12] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "Measuring effectiveness of mutant sets," in *Mutation 2016*, pp. 132–141.

[13] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *ICSE 2017*, pp. 180–182.

[14] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI 2008*, pp. 209–224.

[15] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *ICSE 2017*, pp. 597–608.

[16] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *ICSE 2015*, pp. 936–946.

[17] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, 1997.