2018 IEEE International Conference on Software Testing, Verification and Validation Workshops

An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions

Goran Petrović Marko Ivanković
Google Switzerland GmbH
Zurich, Switzerland
{goranpetrovic, markoi}@google.com

Bob Kurtz Paul Ammann
George Mason University
Fairfax, VA, USA
{rkurtz2, pammann}@gmu.edu

René Just
University of Massachusetts
Amherst, MA, USA
rjust@cs.umass.edu

***Abstract*—Mutation analysis evaluates a testing or debugging technique by measuring how well it detects mutants, which are systematically seeded, artificial faults. Mutation analysis is inherently expensive due to the large number of mutants it generates and due to the fact that many of these generated mutants are not effective; they are redundant, equivalent, or simply uninteresting and waste computational resources. A large body of research has focused on improving the scalability of mutation analysis and proposed numerous optimizations to, e.g., select effective mutants or efficiently execute a large number of tests against a large number of mutants. However, comparatively little research has focused on the costs and benefits of mutation testing, in which mutants are presented as testing goals to a developer, in the context of an industrial-scale software development process.**

This paper draws on an industrial application of mutation testing, involving 30,000+ developers and 1.9 million change sets, written in 4 programming languages. It shows that mutation testing with productive mutants does not add a significant overhead to the software development process and reports on mutation testing benefits perceived by developers. This paper also quantifies the costs of unproductive mutants, and the results suggest that achieving mutation adequacy is neither practical nor desirable. Finally, this paper describes lessons learned from these studies, highlights the current challenges of efficiently and effectively applying mutation testing in an industrial-scale software development process, and outlines research directions.

I. Introduction

Mutation testing offers a rigorous test efficacy criterion, which subsumes many code coverage criteria [1], and there is strong empirical evidence that mutants are a valid proxy for real faults [2], [3]. Despite its effectiveness, there are no indications that mutation testing is widely adopted as a test efficacy criterion in practice, even after three decades of research [4]. We conjecture that this is partially due to its heavy computational footprint [5], but to a larger extent due to disconnects between research and practice. For example, even assuming that all generated mutants can be efficiently evaluated, presenting the results to a developer in a meaningful and actionable manner is virtually impossible given the large number of mutants. A developer will not use a technique when the time investment is large and the perceived gain is not reciprocal, especially if non-standard tooling is required that is not integrated into the existing developer's workflow [6].

This paper draws three lessons from these disconnects, and we hypothesize that resolving the disconnects will significantly advance the adoption of mutation testing in practice.

Lesson 1: Some mutants shouldn't be killed. The research community has focused on identifying and eliminating equivalent mutants, which are considered a major burden for developers, and redundant mutants, which inflate the mutation score. Non-equivalent and non-redundant mutants are considered useful. However, the case studies presented in this paper suggest that there are a significant number of non-equivalent and non-redundant mutants that are not useful in practice. These are simply *unproductive* mutants, which represent futile testing goals—and developer frustration.

Lesson 2: The unit of work matters. The research community typically considers mutation adequacy at the method or file level in the context of a single language. However, the standard unit of work for a developer is the *commit*, which amounts to lines of code added, deleted, or modified, often distributed across multiple files and multiple languages. Mutation adequacy at the method or file level would require a developer to analyze mutants and write tests outside the scope of a commit, an obligation inevitably viewed as an unwelcome diversion.

Lesson 3: Mutation adequacy is too expensive. The research community has focused on mutation adequacy. However, a rational goal for a developer is to just *make a test suite better, but not mutation adequate*. Philosophically, this returns to the roots of mutation testing: providing hints for the practicing programmer [7].

This paper reports on an industry-scale application of mutation testing, identifies challenges to be solved, and proposes research directions to ease future adoption. Specifically, the paper's contributions and organization are as follows:

- The development of the notion of *productive* mutants (Lesson 1: Section II).
- A case study on using mutation testing in a commit-oriented code-review process (Lesson 2: Section III).
- A case study on the costs of unproductive mutants and mutation adequacy (Lesson 3: Section IV).
- A summary of lessons, current challenges, and future research avenues (Section V).

II. Productive Mutants

The mutation testing literature generally considers killable mutants desirable (since these mutants lead to tests) and equivalent mutants undesirable (since these mutants don't).

978-1-5386-6352-3/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSTW.2018.00027

One of the inescapable observations from the studies presented in this paper is that this classification is unworkable in practice. Specifically, there exist killable mutants for which developers justifiably should not and, in practice, will not write tests. Conversely, equivalent mutants sometimes reveal issues in the program and lead developers to make useful changes such as refactoring the code or removing redundancy[1].

We argue that the two equivalence classes of killable mutants and equivalent mutants are insufficient to capture the notion of developer productivity and propose a new definition:

> A mutant is *productive* if 1) the mutant is killable and elicits an effective test, or 2) the mutant is equivalent but its analysis advances knowledge and code quality.

It is important to emphasize that the notion of productive vs. unproductive mutants is inherently qualitative: different developers may sometimes reach different conclusions as to whether a test is effective or whether inspecting a mutant advances the developer's understanding of the code. Nonetheless, a developer, who just analyzed a mutant as part of a development task, is well-placed to make this judgement call. The industrial study in section III specifically collects developers' judgement on each surfaced mutant, and a negative assessment from a developer maps directly to the notion of an unproductive mutant.

Unproductive killable mutants For killable mutants, there are many instances for which developers rarely, if ever, write a unit test—nor should they. As an example, consider the string message associated with an exception or logging output. The string message provides potentially useful diagnostic information, and replacing this message with an empty string or a null reference usually results in a killable mutant. Yet, adding a test that detects such a mutant arguably does not improve the effectiveness of the test suite. Rather, it bloats the test suite with a meaningless and hard-to-maintain test. Such a mutant, while killable, is unproductive, and hence should not be surfaced to the developer.

Figure 1 provides additional examples for unproductive mutants. These examples are derived from our case studies in Section III and IV:

- Figure 1a: Comparing floating points is a tricky business, especially when not testing for zero. For most floating-point comparisons, replacing > with => is meaningless and results in an unproductive mutant—testing for equality of floating points is discouraged and bad practice.
- Figure 1b: In Python, mutating a `del` statement results in an unproductive mutant. While this mutant is killable, writing a test for it is a very involved task because that test needs to access and inspect the Python runtime.
- Figure 1c: The preferred Google code style for Python requires that if some code path of a function returns a value, then all code paths in that function must explicitly return instead of relying on default behavior (i.e., `return None`. Removing a `return None` produces an equivalent mutant or breaks the code style, which is enforced via static analysis. The latter implies an unproductive mutant.
- Figure 1d: Constructors of many Java collections such as `HashMap` or `ArrayList` take an initial capacity as a parameter. Changing the initial capacity produces a functionally equivalent mutant, which potentially impairs program performance—a problem unit testing is not concerned with. This mutant is technically killable by asserting on the collection capacity or tracking memory allocations, but it is unproductive to do so.

[1]Coles made a similar argument in his 2017 Mutation keynote talk, based on developers' feedback regarding the usefulness of some equivalent mutants.

```
1 −  return resolved_confidence_value >  0.95
2 +  return resolved_confidence_value >= 0.95
```

(a) Testing for equality of non-zero floating point numbers is discouraged and mostly meaningless, and hence unproductive.

```
1     x = 5
2     ...
3 −   del x
```

(b) While it is possible to test for a missing `del` statement by inspecting Python's runtime, it is unproductive to do so.

```
1  def returnIsLeaf():
2    if False:
3      return 15
4    else:
5      x = 4
6 −  return None
```

(c) The deletion of a `return None` statement in Python is an unproductive mutant as it breaks the code style guidelines.

```
1 − new ArrayList<>(x * 2);
2 + new ArrayList<>(x + 2);
```

(d) Changing the initial capacity of a collection only affects performance, which is out of scope for unit testing.

Fig. 1: Examples for unproductive mutants.

There are many other types and of language-agnostic unproductive mutants. For example, changing the duration of time variables (e.g., `sleep(1)`) results in unproductive mutants. Such time-related code constructs are commonly used for RPC layers, which are usually mocked in testing, and otherwise don't change the functional behavior of the program under normal operation. Likewise, mutants that remove calls to `Start` and `Stop` functions on plentiful RPC servers and corresponding `Wait` functions are unproductive because they do not affect the behavior of the servers under unit tests.

The exhaustive list of identified unproductive mutants is exorbitant, but these examples provide a good overview.

Productive equivalent mutants For equivalent mutants, the distinction between productive and unproductive is murkier; while productive killable mutants lead to immediate tangible outcome in terms of effective tests, productive equivalent mutants lead to a better understanding of the code and, in turn, may lead to code improvements.

Examples of productive yet equivalent mutants tend to be long and involved—the reason why they advance knowledge about the code—so we describe some examples without providing source code. A mutant that is equivalent due to an

unexpectedly short propagation can pinpoint error masking or redundancy in the code, where a result is discarded and recomputed. Similarly, deleting an optimization results in an equivalent mutant, but if the optimization is premature, the equivalent mutant can draw the developer's attention to that fact. An equivalent mutant which indicates ambiguity in the code can lead the developer to refactor that code so that the mutant becomes, in fact, killable.

Section IV quantifies the number and costs of unproductive mutants; simple-to-analyze equivalent mutants were generally considered unproductive, but complex-to-analyze ones sometimes were actionable, and hence productive.

Unproductive redundant mutants Redundant mutants can be unproductive if they are unrealistic or generally hard to understand—compared to a simpler alternative mutant. While a complex redundant mutant still forces the same valuable test as the simpler alternative, we argue that in this case, the developer would waste valuable time understanding the mutant before writing that test. As an example, consider the conditional statement `if (x < a || x > b)`, which tests whether a value `x` lies outside the (non-empty) range from `a` to `b`. Two mutually redundant, killable mutants for this code are `if (x < a && x > b` and `if (false)`. Both mutants are killable and elicit the same test, but the first mutant is harder to understand, and hence less productive than the second.

III. Mutation Testing in Practice

This section reports on a large-scale case study of applying mutation testing at Google. Specifically, it first describes the implemented mutation testing approach and then evaluates the costs and benefits of this approach.

A. Mutation Testing at Google

Mutation testing at Google is integrated in the code review process, which requires approval from reviewers with ownership and expertise in the language to submit the change. Aside from the comments from the human reviewers, Google's code review tool Critique surfaces analysis data on various aspects of the change both to the author and reviewers. For example, Google's static analysis system Tricorder provides data on code coverage, test results, and code formatting [8]. An author can only submit the change to the source tree when all required approvals are granted by the reviewers.

Critique surfaces live mutants, pointing the author and reviewers to a potential issue with the test suite or source code itself. The author may choose to kill a mutant by adding a test case, to change the code, or to ignore that mutant. The reviewers can instruct the author to kill a mutant, unless the author can provide a satisfying counterargument for why it is not useful to do so. Additionally, the author and the reviewers can indicate in Critique whether a surfaced mutant was useful and provide qualitative feedback for why or why not. The provided feedback informs mutant suppression for future commits. Recall from section II that we consider a mutant *unproductive* if the author and the reviewer indicated that the mutant was not useful.

For any given commit, Critique surfaces at most one mutant per changed or added line for two reasons: 1) it reduces the visual intrusion into the developer's workflow and 2) it is a performance optimization to save computational resources and provide timely analysis feedback. Additionally, the total number of surfaced mutants per commit is capped. The selection of mutants to surface is driven by historical mutant survival rates and the feedback obtained from authors and reviewers—unproductive mutants are suppressed and hard-to-detect mutants are surfaced with a higher probability.

The combination of commit-level, selective mutation and unproductive mutant suppression make mutation testing practical in a large-scale software development environment like Google, where 400,000 mutants are evaluated every month and the reported ratio of productive mutants is 80% and rising [9].

Considering a code repository of 2 billion lines of code and 40,000 commits every day [8], we argue that aiming at mutation adequacy is hopeless. Even driving statement coverage improvements across some Google teams has proven a challenging endeavor, and statement coverage is a much simpler metric to measure, display, and improve. However, commit-level, selective mutation testing with unproductive mutant suppression is a viable approach.

B. Costs of Mutation Testing

We wished to study whether surfacing live mutants adds a significant overhead to the review and revision process—compared to surfacing only code coverage information. To that end, we analyzed the size of commits, the time it took to submit them, and whether or not mutants were surfaced during the review. We identified *relevant commits* as follows:

1) Consider all commits submitted in 2017.
2) Discard types of commits to which code coverage analysis and mutation testing is not applicable. This includes commits that affect only binary files, tests, documentation, or configuration files. This also includes commits that only delete code.
3) Discard commits not written in Java, C++, Python, or Go. While mutation testing at Google is supported for nine languages, these four languages account for almost 85% of all relevant commits.
4) Discard commits written in more than one language. Polyglot commits may introduce a bias due to a longer-than-usual time to submit and uncertainty about the review process. About 90% of the commits were written in a single language.
5) Discard commits from third-party and experimental projects. These commits may introduce a bias due to different review requirements.
6) Discard rollback commits and cleanup commits. Cleanup commits are automated commits for which code metrics are usually ignored.
7) Discard outliers—that is, commits whose time to submit is larger than one month.

Overall, we identified and analyzed 1.9 million commits, written in 4 programming languages by 30,000+ developers.

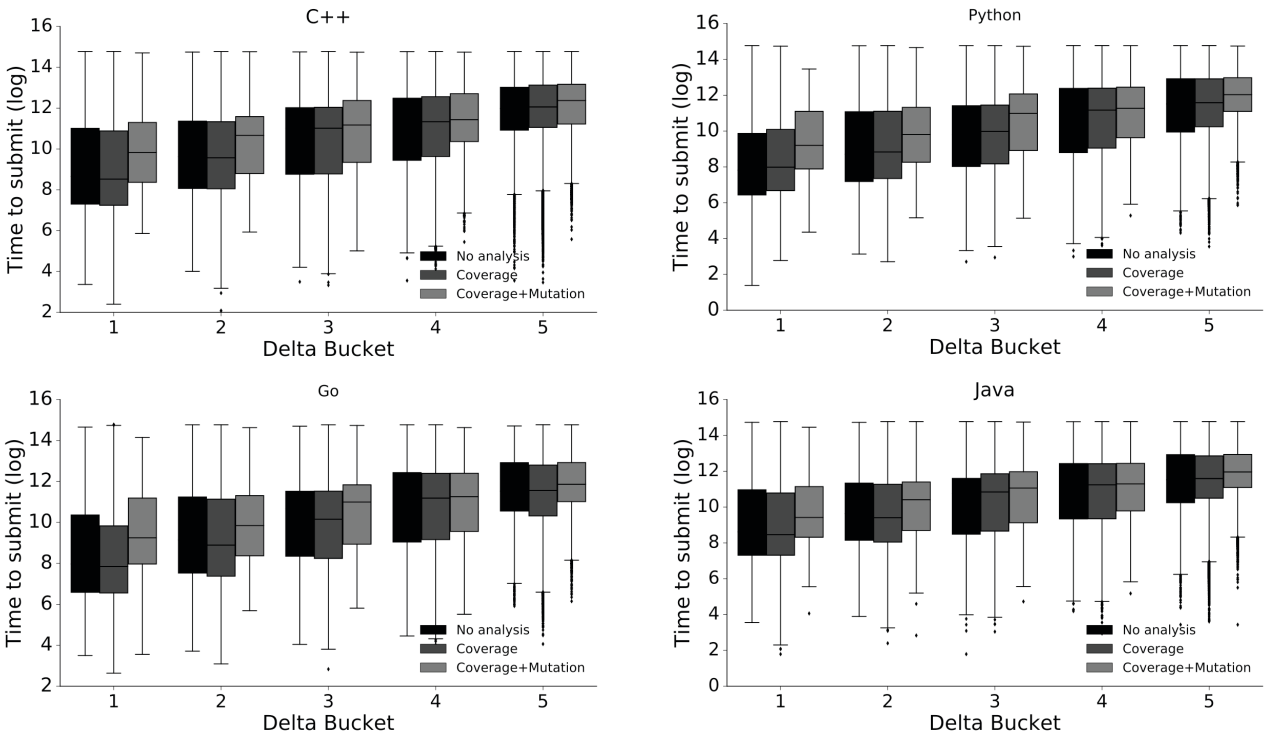

Fig. 2: Distributions of time to submit for each quintile of commit size (*Delta Bucket*) and analysis category. *Time to submit (log)* gives the ln-transformed time to submit, which is measured in seconds.

Both, the average size of a commit and the average time to submit a commit vary significantly between programming languages—some languages are more verbose than others. Considering the set of 1.9 million commits, Python commits tend to be half the size of Java and C++ commits, differences in commit size between Java and C++ are negligible, and the average size of Go commits falls in the middle between Python and Java/C++. Therefore, we separately analyzed the data for each programming language but compared overall trends.

For each programming language, we binned all commits based on delta size (i.e., the total number of added, removed, or changed lines). Specifically, we chose five bins based on the quintiles of all commits for that language. For each bin, we then computed and analyzed the distributions of time to submit in three analysis categories:

1) *No analysis*: Only test results (pass/fail) are available.
2) *Coverage*: Test and code coverage results are available.
3) *Coverage+Mutation*: Test, code coverage, and mutation results are available.

In all delta size bins, there are tens of thousands of commits with only test results, hundreds of thousands of commits with test and code coverage results, and thousands of commits with test, code coverage, and mutation analysis results.

Note that we do not distinguish between commits for which the surfaced mutants were deemed productive vs. unproductive for two reasons. First, an unproductive mutant still requires human analysis to make this judgement call. Second, a developer may also ignore information about an uncovered statement or branch if it is unproductive or impossible to write a test that covers it. Therefore, including all commits for code coverage and mutation analysis enables a realistic and fair evaluation.

Figure 2 shows the distributions of time to submit for each programming language, delta size bin, and analysis category. Overall, the results show little variation between analysis categories for a given language and delta size bin. Additionally, the average time to submit increases with delta size, which is expected. Figure 2 also shows similar trends for all programming languages. Further analyzing the differences, we tested the hypothesis that the difference for time to submit is not significantly (statistically and practically) different between the analysis categories. We chose the non-parametric Mann-Whitney U test and Vargha and Delaney's A12 effect size [10].

The results show that all differences are statistically significant, which is expected given the enormous sample size. However, what matters is the effect size—that is, whether the differences are practically significant. The results show little to no effect, with a notable exception in the two smallest bins. For these two bins, the differences between commits with code coverage results and commits with mutation results show a small to moderate effect (between .58 and .68); time to submit is larger for commits with mutation results. For the three largest bins, all effect sizes are negligible (between .53 and .55). Comparing commits with only tests and commits with additional coverage information, the effect size is negligible (between .48 and .52) across all bins and languages.

For better interpretation of these results, we offer insights into the development and code review processes. While code coverage is an important measure and projects generally have good coverage metrics [9], small changes of only a few lines, or larger automated changes like renames rarely consider coverage or mutation as relevant metrics; in those cases tests ensuring no breakages suffice. In case of features accompanied by tests, coverage and mutation information is much more useful. Many small changes will get submitted without consuming the analysis results, some will even be submitted before the analysis has completed. This is true for coverage and for mutation testing, which requires the completion of the coverage analysis. Larger feature changes requiring thorough review will not be submitted by the time coverage and mutation testing results are surfaced and they are more likely to be picked out by the reviewers. We argue that the non-negligible differences between commits with code coverage results and commits with mutation results in the two smallest bins (figure 2) are, in part, a consequence of this particular development process. Moreover, human expert time is a much scarcer resource—by orders of magnitude—than CPU time spent on performing code coverage or mutation analysis. Our study over-approximates the human time by considering the overall analysis time—the actual human time, in particular for mutation testing, is likely to be much lower.

Although our results stem from 1.9 million commits, 4 programming languages, and 30,000+ developers, they may not be representative of other developers or development processes.

> Commit-level mutation testing does not add a significant overhead compared to using code coverage analysis in a commit-oriented code-review process. This observation holds for all studied programming languages.

C. Benefits of Mutation Testing

Using developer feedback for identifying and suppressing unproductive mutants, the perceived usefulness of surfaced mutants in Critique improved from 20% to 80% [9]. Developer feedback comes in other non-quantifiable ways too, and for mutation testing it has been extensive. Developers have decided to redesign large chunks of code to make them testable just so a mutant could be killed, they have found bugs in complex logical expressions looking at mutants, they have decided to remove code with an equivalent mutant because they deemed it a premature optimization, they've claimed the mutant saved them hours of debugging and even production outages because no test cases were covering the logic under mutation properly. Mutation testing has been called one of the best improvements in the code review verification in years. While this feedback is hardly quantifiable, combined with the sheer number of thousands of developers willing to inspect surfaced mutants on their code changes makes a statement.

> Developers report many perceived benefits of mutation testing, including stronger tests, more effective debugging, prevention of bugs, and improved code quality.

TABLE I: Number of mutants in selected subjects from Lang. *Mutants* gives the total number of mutants, *Live Dev* gives the number of mutants not killed by the developer tests, *Live Dev+Evo* gives the number of mutants not killed by the developer or EvoSuite tests, and *Equivalent* gives the number of equivalent mutants.

Subject	Mutants	Live Dev	Live Dev+Evo	Equivalent
Lang-33	754	131 (17.4%)	108 (14.3%)	80 (10.6%)
Lang-44	1096	284 (25.9%)	207 (18.9%)	118 (10.8%)
Lang-49	1369	246 (18.0%)	229 (16.7%)	142 (10.5%)
Total	3219	661 (20.5%)	544 (16.9%)	340 (10.6%)

TABLE II: Number of examined and unproductive mutants in selected subjects from Lang.

	Lang-33		Lang-44		Lang-49		Total	
	mutants	time	mutants	time	mutants	time	mutants	time
Examined	91	365	158	818	179	782	428	1965
+killed	11	67	40	387	37	322	88	776
+equivalent	80	298	118	431	142	460	340	1189
Unproductive	57	193	40	136	43	128	140	457
+killed	15	58	16	101	18	98	49	257
+equivalent	42	135	24	35	25	30	91	200

IV. The Costs of Unproductive mutants

To understand the amount of effort required to develop a mutation-adequate test set, we used the Defects4J benchmark [11] (v1.1.0), which provides a set of 395 subjects from six open-source projects, each accompanied by a thorough developer-written test suite. For this effort we selected subjects from the *Lang* project where the associated test suite achieved at least 95% statement coverage, then randomly selected three subjects (*Lang-33*, *Lang-44*, and *Lang-49*). All three are of fairly typical size compared to other Lang subjects in Defects4J, and yielded 754, 1096, and 1369 mutants, respectively (see table I). We used the Major mutation framework [12] to generate mutants for the subjects (modified classes [11]) and to perform the mutation analysis. The developer-written test suites of the subjects killed all but 20.5% of the mutants.

In order to more closely approximate the mutants that were actually equivalent, we used the EvoSuite tool [13] to automatically generate additional tests. These tests killed a modest number of additional mutants, reducing the number of live mutants to 16.9% of the total number of mutants.

Finally, we performed mutation testing with the overall goal of developing a mutation-adequate test set for each subject. We manually examined the remaining live mutants to identify which were truly equivalent, and wrote additional tests to kill the others. This significantly reduced the number of live mutants to 10.6%, showing that even with a high-quality test suite with very high statement coverage, approximately half of the remaining live mutants can in fact be killed.

Achieving a mutation-adequate test set, however, was not a trivial task. Manual review of the mutants not killed by developer or generated tests required an average of 4.6 minutes per mutant, for a total time of 32.8 hours. This average time is substantially shorter than the 15 minutes reported in prior

studies [14], [15], but those studies sampled mutants at random from a variety of classes. In contrast, we examined many mutants from a limited number of classes, which allowed for increasing familiarity with the code and increased efficiency. Overall, our results are consistent with prior work: the average time to examine each of the first 10 equivalent mutants from each subject (when familiarity with the code was not yet established) was 11.7 minutes.

We identified 140 unproductive mutants (32.7%) from the 428 mutants that we examined, as shown in table II. Unproductive killed mutants accounted for 35.0% of the 140 unproductive mutants, and required the most developer time with an average of 5.2 minutes per mutant needed to kill each one. This is more than the average of 4.6 minutes for all mutants and nearly 50% more than the 3.5 minutes for equivalent mutants, making these a particular waste of time. Unproductive equivalent mutants were more plentiful, but were easier to review, requiring only 2.2 minutes per mutant.

We also found 17 examples of redundant mutants, where the mutants elicited a useful test, but a simpler mutant exists that elicits the same test while being more easily understood. Redundant mutants are not included in table II since they were not strictly wasted effort. Similarly, many equivalent mutants were not strictly unproductive. In many cases, mutants were only equivalent because of the specific implementation of the code; these mutants provided useful insights and a similar mutation elsewhere in the code or in a different application might well elicit a valuable test. Further, we found several examples where equivalent mutants revealed code redundancy that might lead a developer to refactor the code, improving its clarity. Such equivalent mutants seem in fact quite productive.

> A large ratio of mutants not killed by thorough, industrial-strength test suites are unproductive mutants. The time to write a test to kill an unproductive mutant is on average higher than the time to determine mutant equivalence.

V. Challenges and Research Directions

Based on the case studies reported in this paper, we identified the following challenges for widespread adoption of mutation testing in practice:

- Unproductive mutants represent futile testing goals, and developers are not willing to waste their time on writing pointless tests just to satisfy mutation adequacy. This is similar to code coverage: some lines or branches are more important than others, and 100% code coverage is neither the norm[2] nor considered desirable in many cases [16].
- A developer's workflow revolves around commits. Successfully deploying mutation testing requires a smooth integration into this workflow.
- There are far more mutants than developers can analyze in a reasonable time frame. Since only very few mutants should be surfaced, the challenge is to pick the most productive ones.

[2]https://testing.googleblog.com/2014/07/measuring-coverage-at-google.html

To address these challenges, we propose the following research directions and high-level research questions:

- The notion of a productive mutant is based on whether it improves the test suite, the code base, or knowledge. Can this inherently fuzzy notion be formally characterized, independently of developer judgment?
- Surfacing only developer-preferred mutants may bias the resulting test set with respect to faults in developer blind spots. Can crowdsourcing avoid such a bias?
- Productive mutants cannot be identified in a program-agnostic manner. Can mutation testing be customized to the program under test to be context sensitive and surface mostly productive mutants?
- Coupling between real faults and mutants is still far from being perfect, and selecting fewer mutants decreases fault coupling even further. Is it possible to achieve high fault coupling with a very small set of mutants?
- Currently, mutation testing at Google surfaces at most one mutant per line of code in a commit. Is this the "right" number, and under what circumstances would it be appropriate to surface fewer or more mutants?
- As successive commits revisit the same code over time, does the resulting test set approach mutation adequacy?

VI. Related Work

Mutation testing effectiveness and efficiency Fault-coupling for state-of-the-art mutation systems, while impressive with respect to other coverage criteria, still peaks well short of what is possible. Increasing fault-coupling requires additional mutation operators, and the number of mutants grows very quickly with these mutation operators [17], [18], [19]. To combat cost, the research community has long explored program-agnostic mutant selection strategies [4], [20]—none outperform random selection [21], [22]. Recent work showed that program context matters and explored rule-based and probabilistic approaches to mutant selection [9], [23], [24]. Specifically, Petrović and Ivanković use developer feedback to identify and suppress mutants in uninteresting AST nodes [9]. They summarized heuristics for both generic and language-specific mutant suppression. Just et al. proposed an AST-based program context model, suitable for predicting mutant effectiveness [23]. Fernandez et al. developed 37 rules for useless mutants in Java programs, where the term useless covers both equivalent and redundant mutants [24]. The initial results show promise for developing selection strategies that *do* outperform random selection. Further, Zhang et al. used machine learning to predict mutation scores both on successive versions of a given project and across projects [25].

Mutation testing in practice Reports on large scale application of mutation testing are still rare. Ahmed et al. [26] reported on the experience of applying mutation testing to parts of the Linux kernel. The PIT project aims to make mutation testing usable by practicing developers and sees adoption in industry [27].

Redundant and equivalent mutants A large fraction of the mutation literature has focused on addressing the com-

putational costs and the equivalent mutant problem [4]. Researchers search for techniques that avoid redundant mutants, which not only increase costs but also inflate the mutation score [28], and instead favor hard-to-detect mutants [29], [30] or dominator mutants [31]. Even in the very recent literature [32], effectiveness for mutants is primarily defined in terms of redundancy and equivalence, and does not include the notion that non-redundant mutants might be unproductive or that equivalent mutants can be productive. In our experience, redundant and equivalent mutants have been less problematic than unproductive non-redundant and non-equivalent mutants.

VII. Conclusions

This paper draws lessons from an industry-scale application of mutation testing, identifies challenges that need to be addressed, and proposes research directions to advance the adoption of mutation testing in practice.

Our long-term vision for mutation testing in an industrial setting leverages fine-grained historical data on program context, mutant survival rates, and developer feedback on the usefulness of surfaced mutants. Machine learning classifiers trained on this rich dataset can then predict highly productive mutants to guide the testing effort of a developer.

References

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge, UK: Cambridge University Press, 2017, ISBN 978-1-107-17201-2.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2005, pp. 402–411.

[3] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, November 18–20 2014, pp. 654–665.

[4] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 5, pp. 649–678, 2011.

[5] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2017, pp. 609–620.

[6] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2013, pp. 672–681.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[8] R. Potvin and J. Levenberg, "Why Google stores billions of lines of code in a single repository," *Communications of the ACM*, vol. 59, pp. 78–87, Jul. 2016.

[9] G. Petrović and M. Ivanković, "State of mutation testing at Google," in *Proceedings of the International Conference on Software Engineering—Software Engineering in Practice (ICSE SEIP)*, May 2018.

[10] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability (JSTVR)*, vol. 24, no. 3, pp. 219–250, 2014.

[11] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 23–25 2014, pp. 437–440.

[12] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2014, pp. 433–436.

[13] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, September 2011, pp. 416–419.

[14] B. J. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, April 2009, pp. 192–199.

[15] D. Schuler and A. Zeller, "(un-)covering equivalent mutants," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, April 2010, pp. 45–54.

[16] B. Marick, "How to misuse code coverage," in *Proceedings of the Interational Conference on Testing Computer Software (ICTCS)*, June 1999, pp. 16–18.

[17] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[18] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps, "The care and feeding of wild-caught mutants," in *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, September 2017, pp. 511–522.

[19] M. Allamanis, E. T. Barr, R. Just, and C. Sutton, "Tailored mutants fit bugs better," *arXiv preprint arXiv:1611.02516*, 2016.

[20] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-based and random mutant selection: Better together," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, November 2013, pp. 92–102.

[21] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "On the limits of mutation reduction strategies," in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2016, pp. 511–522.

[22] R. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the validity of selective mutation with dominator mutants," in *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, November 2016, pp. 571–582.

[23] R. Just, R. J. Kurtz, and P. Ammann, "Inferring mutant utility from program context," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2017, pp. 284–294.

[24] L. Fernandes, M. Ribeiro, L. Carvalho, R. Gheyi, M. Mongiovi, A. Santos, A. Cavalcanti, F. Ferrari, and J. C. Maldonado, "Avoiding useless mutants," in *Proceedings of the International Conference on Generative Programming: Concepts and Experiences (GPCE)*, October 2017, pp. 187–198.

[25] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive mutation testing," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2016, pp. 342–353.

[26] I. Ahmed, R. Gopinath, C. Jensen, A. Groce, and P. E. McKenney, "Applying mutation analysis on kernel test suites: An experience report," in *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, March 2017, pp. 110–115.

[27] H. Coles, "Real world mutation testing," http://pitest.org, last accessed January 2018.

[28] R. Just and F. Schweiggert, "Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators," *Software Testing, Verification and Reliability (JSTVR)*, vol. 25, no. 5-7, pp. 490–507, 2015.

[29] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2014, pp. 919–930.

[30] W. Visser, "What makes killing a mutant hard," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, September 2016, pp. 39–44.

[31] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, March 2014, pp. 21–31.

[32] P. McMinn, C. J. Wright, C. J. McCurdy, and G. Kapfhammer, "Automatic detection and removal of ineffective mutants for the mutation analysis of relational database schemas," *IEEE Transactions on Software Engineering (TSE)*, 2017.