A Large-Scale Study of Test Coverage Evolution

Michael Hilton
Carnegie Mellon University
Pittsburgh, PA, USA
mhilton@cmu.edu

Jonathan Bell
George Mason University
Fairfax, VA, USA
bellj@gmu.edu

Darko Marinov
University of Illinois at Urbana-Champaign
Urbana, IL, USA
marinov@illinois.edu

ABSTRACT

Statement coverage is commonly used as a measure of test suite quality. Coverage is often used as a part of a code review process: if a patch decreases overall coverage, or is itself not covered, then the patch is scrutinized more closely. Traditional studies of how coverage changes with code evolution have examined the overall coverage of the entire program, and more recent work directly examines the coverage of patches (changed statements). We present an evaluation much larger than prior studies and moreover consider a new, important kind of change — coverage changes of unchanged statements. We present a large-scale evaluation of code coverage evolution over 7,816 builds of 47 projects written in popular languages including Java, Python, and Scala. We find that in large, mature projects, simply measuring the change to statement coverage does not capture the nuances of code evolution. Going beyond considering statement coverage as a simple ratio, we examine how the set of statements covered evolves between project revisions. We present and study new ways to assess the impact of a patch on a project's test suite quality that both separates coverage of the patch from coverage of the non-patch, and separates changes in coverage from changes in the set of statements covered.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Software testing, code coverage, empirical study, flaky tests

ACM Reference Format:
Michael Hilton, Jonathan Bell, and Darko Marinov. 2018. A Large-Scale Study of Test Coverage Evolution. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages.
https://doi.org/10.1145/3238147.3238183

1 INTRODUCTION

Code coverage metrics are often used by developers to identify how well-tested an application is. There are a wide variety of coverage metrics, including statement, branch, MC/DC, method, file, and path coverage [10]. Statement coverage—the ratio of statements executed by tests divided by total number of statements—is the simplest but most commonly used.

Modern development workflows often use a continuous integration (CI) service to build every push and run their project's tests. As CI services have become widely adopted [25], ancillary services that track additional metrics from CI, such as code coverage, are also becoming more popular. Developers can configure CI services, such as Travis [40], to post code coverage data to a service such as Coveralls [19], which makes the data easily available to developers, for instance when reviewing pull requests. Coveralls maintains a record of a project's coverage over time. Additionally, Coveralls can automatically update coverage badges that display the latest coverage result on a project's homepage [41]. This growing public record of coverage data, organically collected by developers, provides a great opportunity for researchers to study real code coverage data. To the best of our knowledge, we are the first researchers to study coverage data collected and so widely shared by developers.

While the overall statement coverage of a test suite provides some insight into its (in)completeness, it reduces the quality measure to a single ratio, making developers potentially miss valuable information about their test suite and its limitations. For developers of large, stable projects that have a large number of statements, it is often difficult to recognize any noticeable change in this metric from one commit (patch) to another, e.g., for a project with 1 million lines[1] of code, a change in coverage of even 100 lines would only impact coverage by one hundredth of one percentage point. Nonetheless, these changes can add up over time: although a single 100-line patch may not make a noticeable change in coverage, many small patches can make such a change. Even more concerning, coverage of some lines may change non-deterministically due to inherent non-determinism in the tests [22, 28, 29]. Even in smaller projects, where an increase in the overall coverage might be more noticeable, tracking only this simple ratio does not capture *which* statements are covered [32]. In an extreme case, a project with 50% code coverage could maintain that overall coverage while completely flipping the set of statements covered. Coverage can also increase, seemingly indicating a better test suite, even when that is not necessarily the case, e.g., coverage might go up despite a drop in the number of executed statements if code is removed, decreasing the total number of statements even more.

One approach for gaining better insights from statement coverage is to focus not on the coverage of the *entire* system under test (SUT) but, instead, only on the *coverage of each patch* (changed statements) performed on the SUT [4, 19, 29, 32]. Collecting patch coverage can be useful because if a patch is not covered enough,

[1] This paper uses "line(s)" and "statement(s)" interchangeably.

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238183

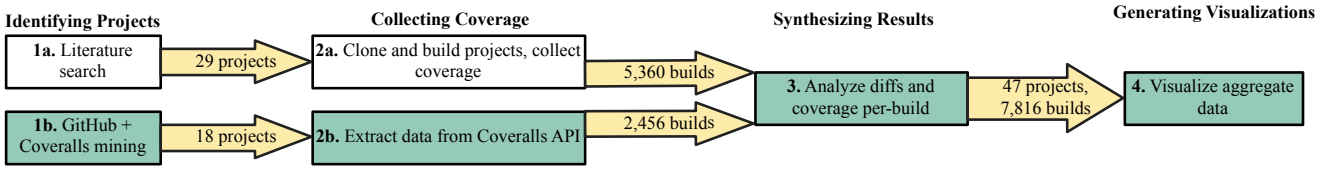

Figure 1: Overview of methodology. Compared to prior studies: green is new; yellow is nearly an order of magnitude larger.

then developers can easily flag this patch in code review and require more tests to be added with the patch. However, even if a patch is well covered, the *impact of the patch* on the non-patch (unchanged statements) part of the SUT is not known a priori.

Prior empirical work has evaluated how coverage changes during code evolution but for a relatively small number of projects and builds. For example, Zaidman et al. [43] reported a high amount of manual labor when studying just 3 projects (and collecting coverage for a total of 30 builds), while Marinescu et al. [29] reported an immense amount of infrastructure needed to collect coverage data from just 6 projects (for a total of 1,222 builds). Further, while prior work has investigated the change in overall coverage, or the coverage of patches, no prior empirical study has examined how the *set of lines covered* changes. Even when the overall SUT coverage appears stable, and the coverage of new patches is high, are the actual lines covered still changing? Or, are there hidden changes to coverage that developers should be aware of, beyond coverage of the entire SUT and coverage of the patch?

To better understand code coverage, how it changes, and how developers can better reason about their code and their tests, we present a large-scale longitudinal study of test coverage evolution. Our study builds on prior empirical work studying code coverage but increases the number of projects (47, written in 7 different languages) and revisions of those projects (7,816) by almost an order of magnitude. Also, ours is the *first* study of code coverage to correlate the coverage of individual lines of coverage throughout project evolution, tracking how a line may become covered and then later become not covered. Further, ours is the first study to utilize code coverage history that developers collect on Coveralls [19] (gathered through the public API of the Coveralls service), in addition to coverage metrics that we collect ourselves.

In this paper, we answer the following questions:

RQ1: What is the distribution of patch coverage across revisions? What fraction of a patch is covered by the regression test suite? Do projects with high patch coverage also have high overall coverage?

RQ2: What impact do patches have on the coverage of non-patch code? Do we see similar behavior across projects? Does high patch coverage imply that a patch increases coverage of non-patch code?

RQ3: Are all changes equally visible? Do projects have changes that are occluded (hidden)?

RQ4: How does the set of covered lines change? Are there hot-spots of coverage change, i.e., lines that flip between being covered and uncovered throughout evolution?

RQ5: What kinds of changes to code drive changes to coverage? Does code coverage change more because old code becomes tested, or because new, tested code is added? Or, do line deletions drive changes to code coverage?

To answer these questions, we prepare an extensive dataset of code coverage, develop a toolset for automated analysis, and perform various analyses, making the following contributions:

- **Dataset:** Our dataset of code coverage information from 7,816 revisions of 47 projects is publicly available: http://www.code-coverage.org
- **Toolset:** Our toolset includes the scripts that gathered our dataset and that can analyze the dataset, allowing other researchers to perform similar experiments and build new tools that analyze coverage change.
- **Novel Coverage:** We highlight the importance of measuring change in coverage of unchanged code.
- **Results:** We perform several analyses and present new findings that answer the listed research questions. One finding is that *which* lines are covered can vary widely in a project, even when the overall coverage appears to remain the same.
- **Implications:** We identify important implications for developers, tool builders, and software engineering researchers working to measure and improve test quality. For instance, we found that changes to non-code files often impact code coverage, and hence, regression test selection tools should track such dependencies to be safe.

2 METHODOLOGY

Researchers have previously studied the evolution of code coverage [21, 29, 43] by downloading some open-source projects and, for several revisions of these projects, compiling the code and running tests while collecting code coverage. Our methodology builds on this approach but significantly expands the *breadth* of the study by including almost an order of magnitude more projects (47) and revisions (7,816). We leverage Coveralls [19], the increasingly popular service for tracking code coverage for open-source projects. We also significantly extend the *depth* of the study by tracking the change to coverage of individual statements across revisions.

Figure 1 shows an overview of our methodology. We first identified candidate projects to include in our study, selecting both projects studied in recent regression testing research and projects that use the Coveralls service. We then collected coverage for these projects, either running the test suites ourselves or collecting data from Coveralls. We next aggregated the data with version-control history to track the coverage of individual code lines throughout project evolution. We finally summarize and visualize the results.

2.1 Identifying Projects

Automatically downloading, compiling, and executing tests for open-source projects is often non-trivial. Some projects fail to compile (e.g. due to missing dependencies), and others require manual configuration or installation of external dependencies. However,

excluding projects that require some degree of manual configuration could bias the projects included in a study. We relied on two complementary approaches to gather a diverse set of projects.

Traditional Evaluation: We identified four recent research papers [12, 15, 23, 36] that had experiments with software evolution (specifically with regression testing). We tabulated the open-source projects studied in these experiments and tried to clone, build, and test each of them. In total, we selected 29 projects: 11 (of the 32) from [23], 5 (of the 10) from [15], 3 (of the 17) from [36], and 10 (of the 26) from [12]. We included all projects which (1) used the Maven build system and (2) successfully compiled (on its most recent commit) with the command `mvn package`. We allowed for up to thirty minutes of troubleshooting per project to potentially install external dependencies required by the project as may have been specified in README files or error messages during this process. These 29 projects make up our *traditional* evaluation set.

***In Vivo* set:** We broadened the scope of our evaluation by including projects which we *did not* compile or test ourselves, instead leveraging coverage data collected and shared by the project developers themselves. Coveralls [19] is a free service that stores coverage data, allowing developers to track the coverage of their projects over time. Many open-source projects from GitHub use Coveralls as part of their continuous integration (CI) pipeline: when developers push their changes to GitHub, a CI service (e.g. TravisCI [40]) automatically fetches these changes, compiles the project, runs the test suite, *and* uploads coverage data to Coveralls. While some prior work [12, 14, 27] has used the output of CI services, like TravisCI, as a dataset for evaluation, we are not aware of any prior work that used code coverage from services like Coveralls as we do. Reusing coverage data has several advantages: (1) we can include projects that are more complex to build, for which developers have provided automated configuration scripts to a CI service; (2) we can include projects written in any language supported by Coveralls, because it abstracts the actual collection of code coverage; and (3) we need not expend resources compiling and running these projects' tests. We refer to the data from Coveralls as our *in vivo* evaluation set, because the coverage results come directly from the field.

To identify projects for our *in vivo* dataset, we started by crawling GitHub to find projects that use both TravisCI and Coveralls services. We searched GitHub by project language (including most popular and more recent languages in our criteria), looking for projects with configuration files that refer to Coveralls, collecting the most-starred projects per language meeting our criteria. For each project that referred to Coveralls, we queried the Coveralls' public API [20] to detect if the project indeed has publicly available coverage data on Coveralls. For each project that had data, we checked the number of builds for which the project shipped coverage data to Coveralls, and the number of lines of code in the most recent version of the project. We then picked arbitrary thresholds to filter out projects with short histories on Coveralls (less than 250 revisions built and tested) or trivially small projects (less than 1,000 lines of code total), leaving 19 projects.

2.2 Collecting Coverage

Traditional Evaluation: We conducted our traditional evaluation by compiling, testing, and collecting coverage on each of the 250 most recent commits of the 29 projects that we had identified, successfully completing a total of 5,382 builds. We ran these builds on a cluster of Ubuntu 14.04 virtual machines, running Apache Maven 3.3.9 and Java 1.8.0_131. We collected coverage using the mature JaCoCo tool [5], configured to collect coverage of all project code files. If a build failed, we did not seek out more builds, hence we may not have 250 successful builds of each project. We considered other coverage tools, Cobertura [18] and Clover [17], but neither fully supports Java 8, and hence, would have limited our study to include only projects that do not use recent Java features.

***In Vivo* set:** Coveralls terms of use request broadly that users of their API do not impose an undue load on the service, and we did not want to abuse the service. Unfortunately, collecting our data required making many requests to the service: one per file, per-revision, per-project. Hence, a project with 1,000 files would require 1,000 requests to collect detailed coverage of each file of a single revision. We self-imposed a rate limit of 5,000 requests per hour and restricted our data collection to only several days. We contacted Coveralls to inquire if they could make the data easier to obtain (e.g., one request for all files in a revision) and to check that our procedure would not place an undue load on their service, but we received no response. Hence, for the 19 projects, we downloaded coverage data for only 2,575 builds—we skipped the remaining builds to not abuse the service.

2.3 Determining Code Changes

The next step in our study required unifying the coverage data with code change information. For each commit of each project, we needed to find which lines were added, modified, or removed from the previous commit. Because our experiments include only projects that use the Git version-control system, it was relatively straightforward to collect code change information using `git diff` . Using the `diff` algorithm, we simplified the potentially rather complex process of tracking lines that may have moved or shifted throughout the codebase [35]. We matched each commit with its parent commit for which we had data, using Git information to track branching and ensuring that each commit was properly matched with a prior commit from which it descended. We compare each commit with its parent, obtaining the list of added and removed lines. For the remaining (unchanged) lines, we built a mapping between the line numbers from the two commits, which is non-trivial when new lines are added or old lines deleted from files, making it difficult to identify where a line from a prior commit is in the next commit. We used `diff` to generate these mappings, invoking it for each changed file to determine the new line number of each line from the previous commit of that file.

2.4 Aggregating Results

Finally, we aggregated code coverage and code change information, and generated visualizations by creating and running a series of R scripts. Table 1 shows a summary of basic statistics for each project. For the remainder of this paper, we refer to projects by their ID (the far left column). For each project, we report the programming language, the prior paper or Coveralls (abbreviated as C.IO), the number of builds studied, average lines of code across commits, and the total commit time window that our coverage data spans. The Coveralls projects are mostly smaller (in LoC) than the rest, but nonetheless are similar in overall coverage. Before addressing our

Table 1: Key statistics describing all of the projects included in this study.

	Project	Lang	Source	Builds	LoC	Time range (months)	Coverage			% of Commits Changing:				Avg. Patch Size (Lines)		
							Start	Sparkline	End	Test	Source	Both	Neither	Source	Test	All
P01	apache/commons-collections	java	[24]	189	12,765	40	84%		84%	6%	34%	24%	36%	89	53	154
P02	apache/commons-dbcp	java	[24]	164	5,662	19	48%		51%	13%	27%	12%	48%	14	14	37
P03	apache/commons-exec	java	[12]	212	971	65	63%		72%	24%	17%	13%	47%	13	20	44
P04	apache/commons-functor	java	[24]	248	2,693	69	83%		97%	11%	47%	8%	34%	116	78	210
P05	apache/commons-io	java	[24]	35	5,021	2	88%		87%	26%	26%	31%	17%	97	79	178
P06	apache/commons-jxpath	java	[24]	199	9,633	94	75%		77%	8%	33%	13%	47%	74	9	125
P07	apache/commons-math	java	[24]	217	45,034	16	90%		90%	19%	39%	32%	10%	314	114	935
P08	apache/commons-net	java	[24]	53	9,210	1	30%		30%	11%	40%	11%	38%	40	4	61
P09	apache/commons-validator	java	[24]	104	2,854	10	77%		78%	23%	36%	21%	20%	44	10	65
P10	apache/empire-db	java	[24]	241	21,258	60	14%		14%	0%	72%	3%	24%	158	1	190
P11	apache/httpcore	java	[12]	223	13,198	18	77%		75%	13%	36%	39%	11%	1,249	387	1,658
P12	ARMmbed/mbed-ls	python	C.IO	60	804	6	75%		78%	2%	65%	30%	3%	27	11	53
P13	bitwalker/timex	elixer	C.IO	128	2,615	16	65%		68%	2%	50%	31%	16%	23	6	79
P14	broadinstitute/firecloud-orchestration	Scala	C.IO	170	2,658	13	64%		68%	4%	16%	64%	16%	93	95	213
P15	containers/virtcontainers	go	C.IO	296	5,332	9	66%		61%	1%	33%	42%	23%	1,276	51	1,664
P16	coreos/alb-ingress-controller	go	C.IO	98	2,041	7	3%		21%	0%	50%	27%	23%	509	34	569
P17	damianszczepanik/cucumber-reporting	java	[15]	248	794	15	88%		99%	9%	8%	48%	34%	60	64	208
P18	dask/dask	python	C.IO	290	15,322	7	94%		92%	5%	21%	54%	20%	46	33	114
P19	doanduyhai/Achilles	java	[12]	111	11,008	9	56%		54%	4%	29%	41%	26%	370	201	626
P20	dropwizard/dropwizard	java	[12, 36]	246	7,700	9	86%		87%	9%	10%	24%	57%	13	21	50
P21	eBay/cors-filter	java	[15]	204	280	45	94%		100%	24%	27%	17%	31%	35	42	83
P22	F5Networks/k8s-bigip-ctlr	go	C.IO	103	5,621	5	76%		83%	7%	24%	33%	36%	68	48	191
P23	fasseg/exp4j	java	[15]	233	640	40	89%		95%	6%	12%	39%	43%	292	80	382
P24	Gillespie59/eslint-plugin-angular	node	C.IO	212	1,213	19	100%		100%	5%	20%	47%	28%	99	30	195
P25	goldmansachs/gs-collections	java	[24]	249	38,242	16	92%		93%	20%	20%	41%	19%	295	292	767
P26	google/jimfs	java	[12]	100	3,401	45	89%		91%	11%	23%	39%	27%	758	292	1,054
P27	HazyResearch/deepdive	Scala	C.IO	111	1,913	5	81%		73%	0%	17%	8%	75%	31	29	271
P28	hector-client/hector	java	[12]	137	8,583	25	35%		39%	4%	55%	32%	9%	114	50	195
P29	ikawaha/kagome	go	C.IO	91	1,099	27	80%		86%	10%	33%	29%	29%	111	29	144
P30	ilovepi/Compiler	dotNet	C.IO	96	1,874	1	87%		90%	7%	57%	31%	4%	209	19	238
P31	jhy/jsoup	java	[15]	246	6,615	28	76%		77%	11%	23%	41%	25%	33	16	70
P32	jknack/handlebars.java	java	[12]	100	3,935	9	83%		84%	11%	30%	40%	19%	192	26	225
P33	JodaOrg/joda-time	java	[24, 36]	248	14,789	35	90%		90%	5%	25%	17%	53%	11	7	90
P34	joel-costigliola/assertj-core	java	[12, 36]	241	11,104	9	90%		90%	15%	22%	42%	22%	234	315	557
P35	mailgun/kafka-pixy	go	C.IO	64	4,387	13	80%		69%	6%	27%	59%	8%	983	233	1,385
P36	MITLibraries/topichub	Scala	C.IO	102	2,466	11	57%		60%	7%	25%	43%	25%	33	40	112
P37	platinumazure/eslint-plugin-qunit	node	C.IO	62	628	22	100%		100%	8%	6%	27%	58%	55	27	102
P38	PragTob/benchee	elixer	C.IO	148	441	6	94%		94%	7%	29%	44%	20%	56	52	119
P39	raml-org/raml-java-parser	java	[15]	248	6,455	16	86%		86%	2%	58%	23%	17%	168	9	605
P40	ShiftForward/apso	Scala	C.IO	94	1,629	9	54%		59%	7%	19%	29%	45%	30	13	58
P41	spatialmodel/inmap	go	C.IO	55	5,983	9	81%		83%	5%	24%	36%	35%	273	98	442
P42	square/okhttp	java	[12]	247	11,854	10	78%		78%	10%	29%	45%	15%	36	33	72
P43	square/retrofit	java	[24]	181	2,479	17	56%		57%	7%	35%	38%	20%	119	102	235
P44	SteamDatabase/ValveResourceFormat	dotNet	C.IO	179	2,794	23	82%		73%	2%	80%	4%	13%	108	1	147
P45	terasolunaorg/terasoluna-gfw	java	C.IO	97	2,561	17	99%		99%	12%	4%	16%	67%	1,481	1,188	2,921
P46	undertow-io/undertow	java	[12]	238	51,388	9	60%		60%	7%	72%	18%	4%	49	12	64
P47	zxing/zxing	java	[12]	198	15,440	21	68%		76%	5%	34%	16%	45%	31	7	712
Total 47 projects, 384,412 LOC			**Average:**	166	8,178	20	74%		76%	9%	32%	30%	29%	224	93	397

five research questions, we present three demographic questions, **DQ**s, that describe the overall composition of our dataset in terms of patches and overall coverage.

DQ1: Do patches touch both code and tests? This question mirrors a question often studied in the context of code coverage and mining software repositories [29, 43]. We examined each patch of each project, looking at the files changed by each patch. We categorized each file as a test code file, source code file, or a non-code file. Code files were defined as ending with the correct suffix given the project language: .java, .scala, .go, .js, .ts, .cs, .ex, .exs, .py. Following prior work [43], code files were then categorized as test or source code files if their path contained "test" or ".spec". Table 1 presents the results. Similar to prior studies [29], we found that few commits modified only test files; far more common were commits that modify both test and non-test files, or only non-test files.

DQ2: What are the sizes of each patch? This question also mirrors one posed previously [29]. If a project has primarily small patches, then these patches are perhaps easier for humans to reason about. However, if those patches are larger—hundreds, or thousands of lines, they may require different approaches to be reasoned about. To answer this question, again, we categorize files as "source code" or "test code," and compute the number of changed lines in each file. This includes all changes to these files and counts each edited line as a change. Table 1 shows the results ("Avg. Patch Size (Lines)"), including *all* changed lines in the "all" column (in source code, test code, or non-code files).

Our results strikingly differ from those of Marinescu et al. [29]; their study of six C/C++ programs reported a median number of patch lines ranging between 4 and 7. While the difference may be partly due to us counting *all* changed lines (not only executable statements), this alone is unlikely to lead to such a significant difference in the extreme cases of projects such as P11, P15, and P45. We believe that this indicates that many of the projects that we studied were under substantially more active development than the mature projects in their study (GNU Binutils, Git, Lighttpd, Memcached, Redis, and ØMQ). This finding also underscores the importance of sampling a diverse set of projects in empirical studies.

DQ3: How does coverage change over time? Finally, we calculated the coverage for each project across all commits we studied. Table 1 reports the average lines of code (LoC) and coverage of each

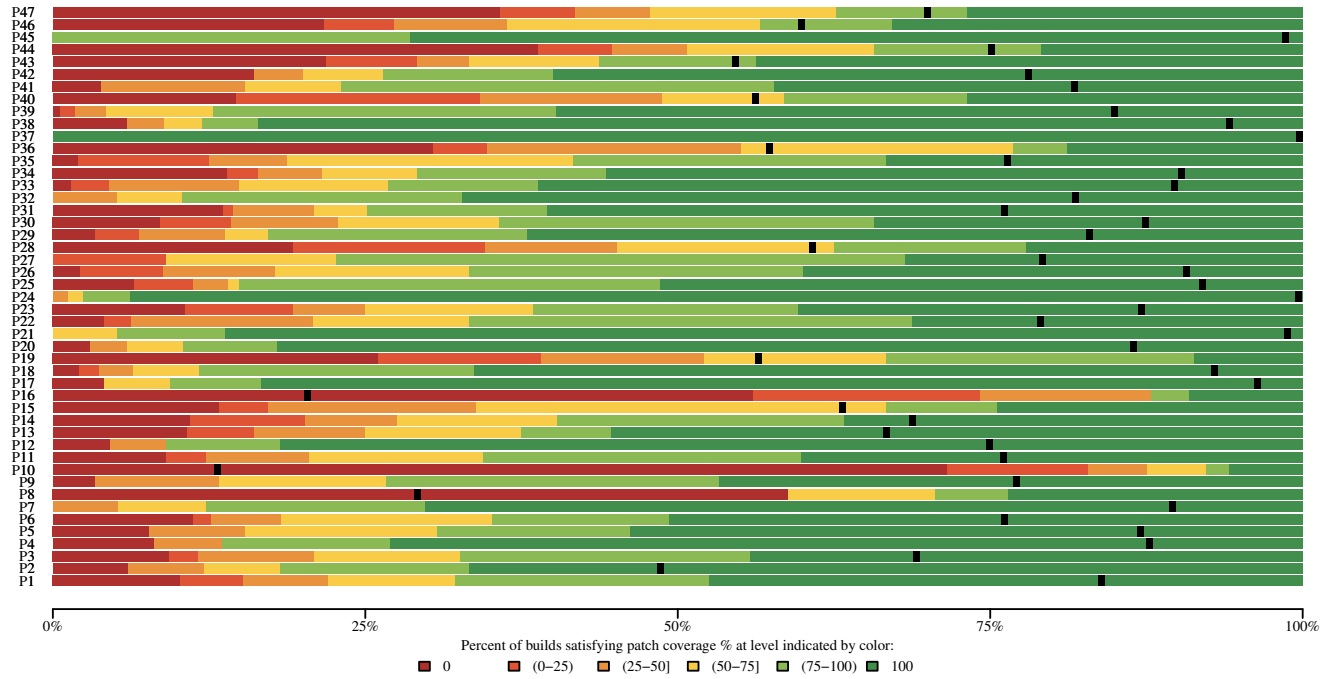

Figure 2: Coverage of new lines in each patch. Each bar represents the proportion of builds with that coverage level (as denoted by the color). The black bar indicates the average *overall* coverage of each build per-project (against the percentage scale)

project at the start and end of our dataset, along with a *sparkline* visualization showing how the coverage changed over time (scaled to 0-100%). We see that our dataset contains a diverse set of projects: some with low coverage, others with high; some with little change, others with more. For most projects, coverage remains relatively flat during the studied window, similar to prior results [29].

However, from the sparklines we see that some projects—in particular `P23`, `P27`, and `P44`—have spikes in their coverage, where coverage drops significantly and then returns to its prior position. After manual inspection, it appears that this is often caused by broken tests: if a test fails early in its execution, then it does not continue to run and cover the statements that it would typically cover. Unfortunately, while we had test result information from our own experiments, we did not have easy access to test results from Coveralls to filter out failed tests.

It is perhaps unsurprising that we do not observe significant changes to coverage, given that our projects are non-trivially large, averaging 8,178 lines of code. The most visible spikes (e.g. `P23`, `P27`) are in projects with the fewest lines of code, while the largest projects (`P07`, `P25`, and `P46`) appear nearly flat. Given that statement coverage is simply the ratio of executed statements to total statements, increasing coverage by even one percentage point may require covering thousands of lines of previously uncovered code. Hence, on a day-to-day basis, developers (especially of large, mature projects) are unlikely to see changes in total coverage.

3 RESULTS

RQ1: What is the distribution of patch coverage across revisions? Since it is difficult to observe changes to overall project coverage on a day-to-day basis, prior work [4, 19, 29, 32] as well as current tools [1–3, 6, 7] have advocated that developers pay particular attention to coverage of patches. We use the term *patch*, commit, and change-set interchangeably: a unit that represents a developer's changes to code (without attempting to understand the nature of the change as a bug fix or new feature). Given that patches are generally much smaller than the overall codebase, patch coverage might be more meaningful to developers reviewing a patch.

To study the coverage of patches in the wild, we calculated the coverage of all changed statements in each patch in our dataset. To visualize these results, we binned each patch by its coverage, choosing bins of 0%, (0%-25%), (25%-50%], (50%-75%], (75%-100%) and 100% coverage of the patch. Figure 2 shows the distribution of patches in each bin, by each project. We also visualize the average coverage across all versions of all of the code in each project with a black bar. For example, for the project `P01`, almost 50% of the code patches have 100% coverage. However, the overall coverage of all code across all builds for this project is 85%.

This visualization is similar to one created by Marinescu et al. in their study of patch coverage of six projects, with two distinctions: (1) we add two more bins ("0" and "100" to segregate patches that are fully covered or not at all covered, rather than simply 0-25, 25-50, 50-75, 75-100), and (2) we superimpose the overall project coverage. Adding these two additional bins allows us to recognize that, in fact, patches are often either entirely covered, or not at all covered: it is far less frequent in the projects that we studied to observe patches that were partially covered.

While it might seem intuitive that higher patch coverage implies higher overall coverage, when we look at our data, we did not see evidence of this. From this chart, we can observe that having more patches with higher coverage does not always indicate higher overall coverage. For example, when comparing `P03` and `P02` we

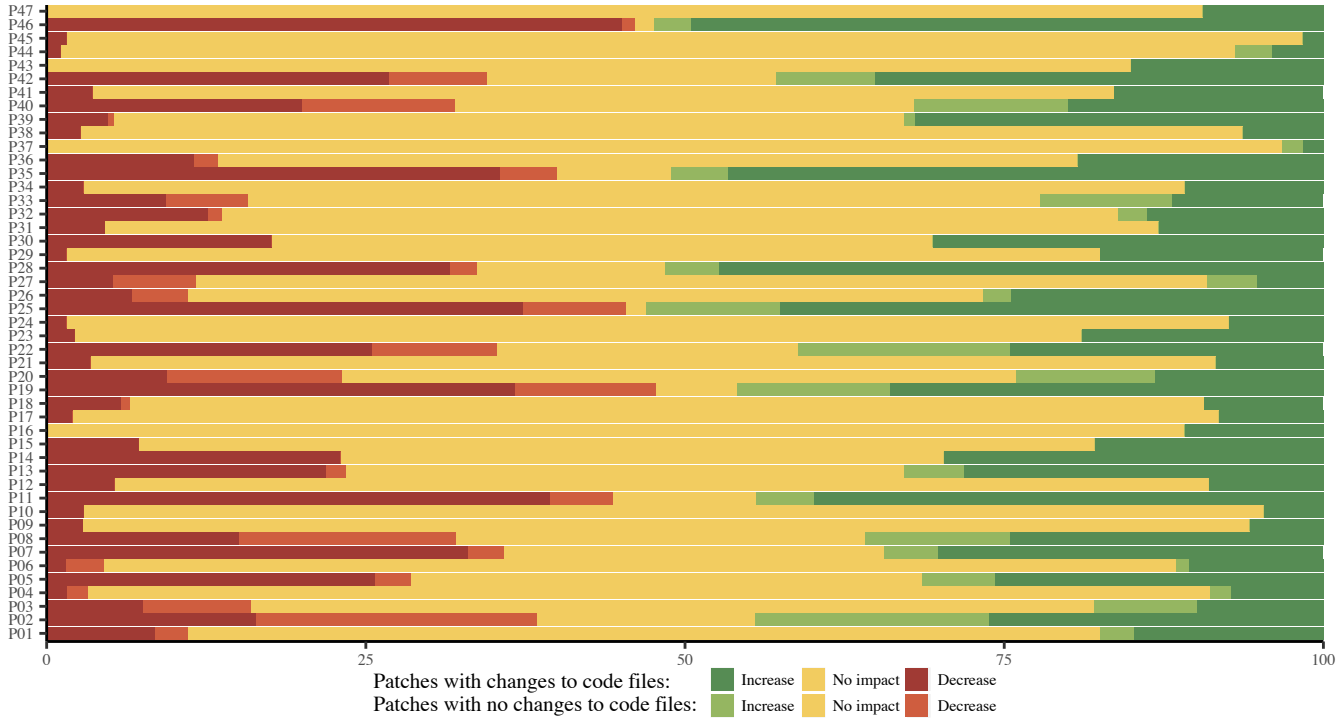

Figure 3: For each patch, we show whether it increases, decreases, or has no impact on coverage of existing (non-patch) code. The size of each bar represents the percent of patches in that bucket.

see that even though P02 has a higher percentage of patches with more coverage, it has lower overall coverage than P03. To better understand the relationship, we tested the correlation between patch coverage and overall coverage. We computed the Kendall Tau coefficient between patch coverage and overall coverage for each patch. We found that there was no correlation between the two variables [r=-0.01,p<0.01].

> **Finding:** Patch coverage varies widely between projects. Patch coverage does not correlate with overall coverage.

RQ2: What impact do patches have on the coverage of existing (non-patch) code? While patch coverage considers how well-tested a patch may (not) be, it surely cannot be the only criteria used to judge the impact of that patch. For instance: a patch might have 100% patch coverage, but applying that patch might reduce the coverage of existing code.

Hence, to better understand the impact of each commit, we also look at the effect that each commit has on the unchanged, existing, *non-patch* code in the project. We categorized the impact of each commit on existing (non-patch) code as a net-increase to the number of existing lines covered, net-decrease, or having no impact. Upon a preliminary investigation, we observed that many of the commits which increased or decreased coverage in non-patch code contained no changes to code themselves. Hence, we further separated each of these groups into patches with and without changes to code files. This statistic is complementary to patch coverage: when reviewing a patch, in addition to seeing that the patch is covered or not, developers can also see if this patch increases or decreases the coverage of the rest of the codebase. Rather than looking at total coverage (of both the new and the existing code), by separating the coverage of a patch from the coverage of existing code, we can observe instances where overall coverage might go up (for instance, because a patch contained a very large number of newly covered lines), but coverage of non-patch code might go down (because that patch removes calls to existing code).

Figure 3 shows the impact of each commit on non-patch code coverage for all commits for each project in our corpus. It is interesting to note that different projects have very different profiles. Some projects, such as P02 have many commits to non-code files which nonetheless have an impact on coverage. Other projects, such as P47 have almost no such commits, where most commits do touch code files and do impact coverage of existing code.

Upon manual inspection, we found that many of these non-code changes involve changing configurations. These changes could be causing changes to coverage due to differences between different versions of APIs or other non code changes. P46 contained many non-code changes; one example commit message describes the change as "Fix build on latest JDK9" and the only changes are to the project's pom.xml file [8]. These changes to coverage could also be due to non-determinism [22], rather than intentional changes.

To determine if there is a relationship between patch coverage and non-patch coverage, we perform a statistical analysis. For each commit in our data, we look at all of the patches which have at least one statement in their diff. We then computed the change to non-patch coverage by calculating the ratio between the number of non-patch lines hit and the total number of non-patch lines. We computed the Pearson's correlation coefficient between patch coverage and non-patch coverage for each patch. We found no

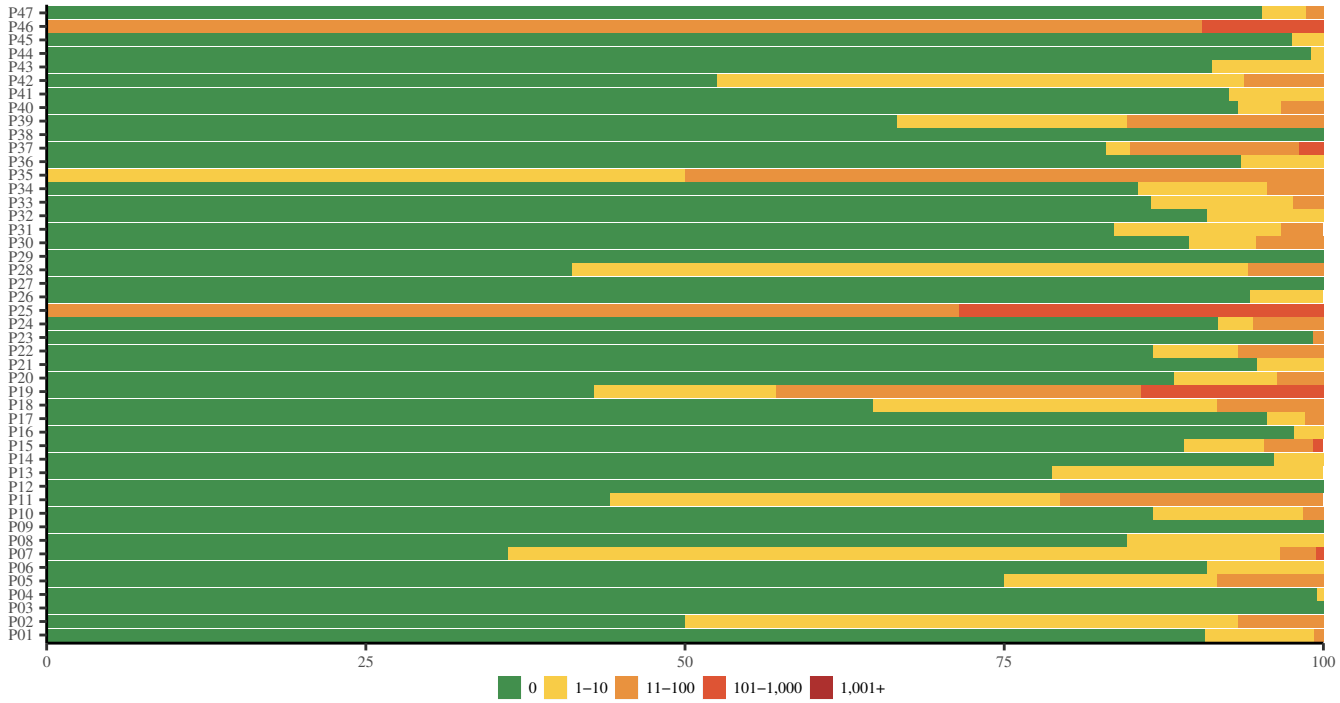

Figure 4: Percentage of commits (size of bar) with statements changing coverage (color of bar) even when the total project coverage did not appear to change.

correlation between the two variables [r=-0.004,p=0.79], concluding that coverage of a patch is not correlated with the patch's impact to coverage of existing, non-patch code.

> **Finding:** Patches often impact coverage of existing (non-patch) code, and high patch coverage does not correlate with increasing non-patch coverage.

RQ3: Are all changes equally visible? When developers observe that overall coverage has not changed between different builds, they might naturally assume that there have not been changes to what their tests execute. However, it is possible that the *lines* covered might change — perhaps drastically — even if the total project coverage appears to be the same. We call these changes *occluded*, and study their prevalence in our dataset. To do so, we filtered our data to examine only commits where there did not appear to be any change to coverage from the prior build (specifically, where the difference in coverage was less than 0.01 percentage points), and then calculated the number of statements changing coverage.

Figure 4 shows how many occluded changes we observed per-project, per-commit: each colored bar represents a range of occluded statements, and the size of the bar represents the percentage of that project's commits at that level. We observe that the number of occluded changes varies widely by project — they are very prevalent in some projects and uncommon in others. However, every project had at least one commit where there were occluded changes, indicating that "steady" coverage does not imply "no change" to code covered. We looked closer at `P25`, `P35` and `P46`, which *always* had occluded changes. We found that `P25` contained a large amount of generated code that often changed between commits, but the size of that generated code remained stable, making coverage appear to be stable. We found that general non-determinism in `P35` and `P46` caused the lines covered to vary.

> **Finding:** Even when patches *appear* to leave coverage unchanged, the set of lines covered can still vary widely. Developers should not trust a seemingly steady coverage metric to indicate that the *same* lines are continuously covered.

RQ4: How does the set of existing lines covered change? In our prior questions, we considered how a patch is covered or how it impacts the coverage of existing code. Here, we study the coverage of individual lines changing over time. Each time that tests are run, the coverage of a line might "flip" from covered to uncovered, or uncovered to covered. For each commit, we examine each flipped line in our dataset. To do so, we used the `diff` tool to identify (1) all lines that exist in *every* version of each project studied, and (2) a mapping for each of those lines in each revision to the equivalent line number in the most recent revision. We used this global ID to track the position of each line over time and then computed the number of times that each of these lines flipped coverage. Figure 5 shows the distribution of the number of flips for each of these lines. We see lines which only flip once, all the way up to a single line which flips coverage 128 times. A line with a high flip count is likely covered non-deterministically, and hence, its coverage might be less important for developers to follow on a day-to-day basis. We note that `P25` and `P46` (which had many occluded changes) also have lines with many flips in coverage.

We randomly select some of the lines with many flips to better understand why lines are flipping coverage. In one, a line in `P20`

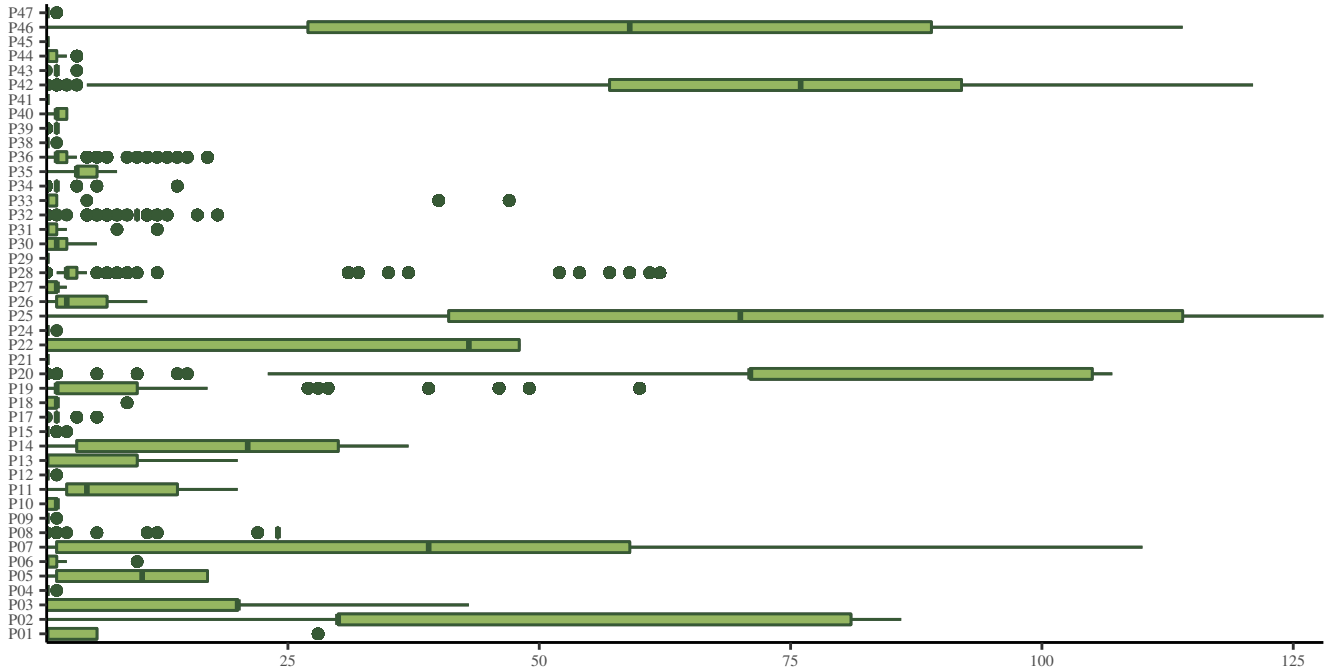

Figure 5: Distribution of how many times coverage flipped for unmodified lines throughout all revisions of each project, showing only lines that changed coverage at least once. The x-axis displays the number of covered/uncovered transitions.

(DBIHealthCheck.java:31), the high-flip statement checks to see if a database connection is still open at a regular time interval. In some cases, tests might complete before this health check is scheduled, causing it to not be executed. In another case, a line in `P08` (TFTPServer.java:643), we found a high-flip statement that was dependent on a network socket's state — and would detect lost packets. The coverage of this line is dependent on whether packets are lost in transmission during the test case — which occurs non-deterministically.

> **Finding:** Lines may often flip between covered and uncovered, suggesting non-determinism in the test suite. Tool builders should consider how to best track and represent this non-deterministic coverage.

RQ5: What kinds of changes to code drive changes to coverage? We conclude our study by returning to project-level coverage, looking at what *kinds of changes* cause coverage to change. A traditional viewpoint might be that coverage increases because existing lines of code become covered, and coverage decreases because those lines are no longer covered. However, of course, looking at a specific revision of a project (compared to the prior), coverage can change for a variety of reasons. For instance, adding new lines will cause coverage to increase or decrease, depending on the coverage of the new lines that are added to the code. Likewise, deleting lines can also impact coverage. If the deleted lines were covered, it can cause coverage to decrease, and if the deleted lines were not covered, it can cause coverage to increase. Coverage can also change due to change in coverage of unchanged lines.

Figure 6 shows all of these impact factors for all revisions of each project. By identifying the different impact factors and what role they play in each projects, we can make observations about how the projects and their test code are changing over time, rather than simply observing "coverage increased" or "coverage decreased". For some projects (e.g., `P38`) we observe that over 75% of the changes to coverage are because of new lines being added. This suggests that most of the code changes are coming from new development. However, for other projects (e.g., `P02`), we observe that they experience many changes when coverage is lost or added to existing lines. This seems to point to higher levels of non-determinism in that project's tests, especially when there are a similar number of changes adding and losing coverage. Comparing to Figure 5, we observe that projects with lines that flip coverage often (`P02`, `P07`, `P25`, `P42`, `P46`) also have significant numbers of coverage changes driven by changes to existing code.

> **Finding:** Many factors have an impact on coverage: newly covering existing statements is not always the primary driver to coverage change.

4 DISCUSSION

In this section we discuss our findings, presenting implications for *Developers*, *Tool Builders* and *Researchers* and discuss several limitations of our study.

4.1 Implications

Developers: Between subsequent revisions, there is often very little observable change in overall project coverage, especially for large projects. Because of this, developers use tools [1–3, 6, 7] that examine patch coverage. However, while knowing if a patch is covered or not clearly has value, developers should *not* use patch coverage as a stand-in metric to evaluate the impact of a patch

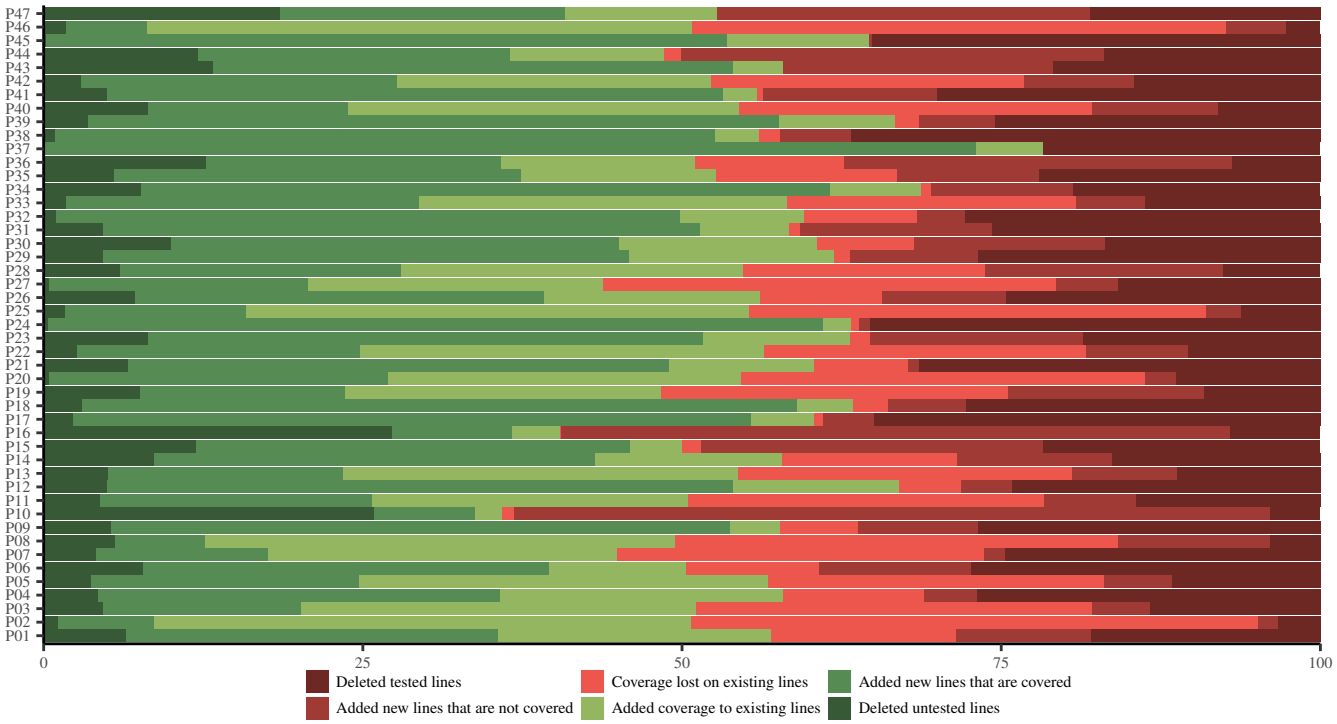

Figure 6: Overview of key factors driving change to coverage of each project during the entire development period measured. Size of bar represents how much that factor contributed to the net change in coverage. Green factors increase coverage, red decrease coverage.

on the overall project coverage. We found that patches often impact the coverage of existing non-patch code, and importantly, that having high patch coverage does *not* correlate with increasing the overall coverage of a project. Developers should adjust their code review processes to consider not only the patch coverage, but also its effect on non-patch code coverage. More generally, developers should consider using more detailed metrics than just the ratio of statements covered to measure their code's testedness.

Developers should be aware that even if the *number* of lines covered remains relatively static, the set of covered lines can greatly vary between runs. If there is concern that a particular part of the project is covered, developers need to specifically track that the relevant lines in that part are covered, because we found that there can be a significant churn in the set of lines covered. We found that in some projects, individual statements might change their coverage very often — perhaps *every single build*. Hence, developers who closely track coverage must be aware of inherent non-determinism in coverage. If developers have a better understanding of how and where their coverage is not deterministic, they may be better prepared to address other impacts of this non-determinism, such as flaky test failures.

Tool Builders We observed that many commits do not involve changes to code files, yet these changes can still change the coverage of existing code, a finding with implications for tool builders whose tools rely on code coverage (for instance, regression test selection tools). Tool builders must be aware of all inputs (e.g. config and data files, as well as non-determinism) that can impact test execution — not only the code itself.

Our study also has implications for tool builders creating code coverage tools. Other researchers [22, 29, 39] have also identified non-deterministic behavior when studying tests. We believe that tool builders should build tools to help developers identify which tests are deterministic, and which are not. Currently, there is no way for developers to identify if their tests are covering code in a deterministic or non-deterministic manner without rerunning tests and comparing very low level coverage data, or waiting for a test to fail in a non-deterministic manner, and having to track down the source of the problem. To help developers identify non-deterministic behavior, tools should show developers which lines have changes in coverage (even unchanged lines), and which of those changes are non-deterministic (or at least potentially non-deterministic). This information would be a valuable tool for developers when debugging flaky test failures [12].

We also found that there are various reasons why coverage can increase or decrease. Current tools show the change in coverage, but they do not show *why* there was a change. If developers are aware of how their coverage is changing, and they are not expecting it to change based on their latest commit, they can then evaluate whether their tests are flaky, or if their latest change impacted the system coverage in ways that they were not expecting. Either way, having this information can help developers better understand their system, and the state of their automated test coverage. Code

coverage tools should fuse code change information with code coverage information, as the Operias tool [32] also proposed.

Researchers Our study was the first to make use of the Coveralls service [19] to gather code coverage data. Code coverage data is notoriously hard to collect, because older project versions are often hard to compile and run, and collecting coverage takes machine time. Fortunately, Coveralls both collects and makes available real data from many different types of projects. This data can be invaluable for researchers who wish to better understand code coverage. By leveraging this data, researchers can use coverage data without having to collect and run historical versions of software, which may be very difficult due to missing dependencies and other infrastucture-related problems. We expect that analysis of data from Coveralls can lead to new insights similar to how analyses of GitHub and TravisCI [13, 16, 33, 37, 38, 42, 44] data enabled researchers to obtain new insights into software development.

4.2 Threats to Validity

1) Construct: Are we asking the right questions? To ensure we are asking the right questions, we base a number of our questions on previous research. For our new research questions, we posed these questions before looking into our data, based on our anecdotal experience from our own development noticing that code coverage can vary greatly.

2) Internal: Did we skew the accuracy of our results with how we collected and analyzed information? We chose projects that were used in previous research, enhanced with a set of large projects with diverse demographics collected from Coveralls. To provide confidence that we have not skewed the results and allow for greater scrutiny, we have made all of the scripts that we wrote and the data we collected available with this paper.

3) External: Do our results generalize? To have our results generalize as much as possible, we selected a large and diverse set of projects in various languages, and from various types of applications. Our dataset is almost an order of magnitude larger (both in terms of the number of projects and the number of builds) than prior related studies, which is encouraging. All of the projects are open source, so we cannot make any claims about how our results might generalize to proprietary projects.

4) Replicability: Can others replicate our results? To support others in replicating our results, we have made our data and the R scripts that we used to process our data publicly available. These can all be found on the project's companion website: http://www.code-coverage.org

5 RELATED WORK

Previous Coverage Studies. We are not the first researchers to study code coverage of software programs. Elbaum et al. [21] study two systems using different types of code coverage metrics. The authors find that the impact of changes on coverage information can be difficult to predict, but calls for further study of the effects of software evolution on coverage information is needed.

Zaidman et al. [43] study three systems and observe changes to coverage. The paper reports that there are periods when the tests and code evolve together, but there also are periods of intense testing. The paper also suggests future work should include analyzing more and larger cases to better understand test coverage evolution.

The most related work to ours is from Marinescu et al. [29]. The authors present both a tool and dataset of code coverage. To evaluate the tool, the paper uses six C/C++ systems. The paper answers nine research questions, three of which are repeated in this paper. Marinescu et al.'s work was also the first to specifically focus on patch coverage, although other researchers [32] have developed tools to help developers visualize patch coverage. In this paper, we examine both patch coverage and also non-patch coverage, and use a significantly larger dataset.

Other Coverage Work. Coverage has often been used as a metric when studying some property of a system. Kochhar et al. [26] find that code coverage has an insignificant correlation with the number of bugs that are found after the release of software at the project level. Mokus et al. [31] find that test effort increases exponentially with test coverage, but the reduction in field problems increases linearly with test coverage. They suggest that the optimal level of coverage for most projects is likely to be well short of 100%. Ahmed et al. [9] study the relationship between statement coverage and mutation score. They find that both metrics have only a weak negative correlation with bug-fixes. Memon et al. [30] describe the challenge when dealing with a codebase the size of Google's, and how it is not possible for them to collect coverage at that scale. Pinto et al. [34] have used coverage to study how tests evolve over time. One category of test evolution they identify is coverage augmentation tests. Gao et al. [22] investigate differences between unit testing, system tests, and invariant detection. They find that when executing system tests, there is often significant non-determinism in the lines that are executed by each test. Our results confirm this finding also.

6 CONCLUSIONS

Statement coverage is often used by developers to evaluate the quality of their test suites. However, by reducing coverage to a single ratio, much valuable information is lost. When working with a large mature project, only very large changes to the number of lines covered will be detectable as a change in the overall coverage, so moderate changes to the test suite may not be observable. Even on smaller projects, viewing coverage as a simple ratio hides potential non-determinism that exists in tests and changes to which statements are covered. Of course, many of these changes to non-patch code may be due to the genuine impact of code changes too; interesting future work may try to identify changes due to non-determinism versus those due to code changes, perhaps using dynamic taint tracking [11]. In this paper, we found that measuring the change in the *set* of statements covered, and the impact of a patch on the coverage of those statements allows developers much more visibility into the impact of their changes. We have released our tools and data so that others can benefit from them and build on our work to obtain new insights that eventually lead to improving quality of testing.

ACKNOWLEDGMENTS

We thank Hayder Al Haddad and Owolabi Legunsen for discussions about this work. Darko Marinov's group is supported by NSF grants CCF-1409423, CCF-1421503, CNS-1646305, and CNS-1740916; and gifts from Google and Qualcomm.

REFERENCES

[1] 2018. coverage-diffe. https://docs.codecov.io/v4.3.6/docs/coverage-diff.
[2] 2018. diff-cover. https://github.com/Bachmann1234/diff-cover.
[3] 2018. diff-coverage. https://github.com/ptone/diff-coverage.
[4] 2018. git-coverage. http://stef.thewalter.net/git-coverage-useful-code-coverage.html.
[5] 2018. JaCoCo Java Code Coverage Library. http://www.eclemma.org/jacoco/.
[6] 2018. jest-diff-coverage. https://github.com/Hylozoic/jest-diff-coverage.
[7] 2018. Patch-Status. https://github.com/codecov/support/wiki/Patch-Status.
[8] 2018. Undertow.io commit a945c17f58cd809558950d858030379179dfdf82. https://github.com/undertow-io/undertow/commit/a945c17f58cd809558950d858030379179dfdf82.
[9] Iftekhar Ahmed, Rahul Gopinath, Caius Brindescu, Alex Groce, and Carlos Jensen. 2016. Can Testedness Be Effectively Measured?. In *FSE*.
[10] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing* (1 ed.). Cambridge University Press, New York, NY, USA.
[11] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms *(OOPSLA)*.
[12] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *ICSE*.
[13] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *MSR*.
[14] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *MSR*.
[15] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie. 2017. How Do Assertions Impact Coverage-Based Test-Suite Reduction?. In *ICST*.
[16] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. 2017. An Empirical Analysis of the Docker Container Ecosystem on GitHub. In *MSR*.
[17] clover. [n. d.]. Clover. https://www.atlassian.com/software/clover.
[18] cobertura. [n. d.]. Cobertura. http://cobertura.github.io/cobertura/.
[19] Coveralls. [n. d.]. Coveralls.io. https://coveralls.io/.
[20] coverallsapi. [n. d.]. coveralls API. https://coveralls.zendesk.com/hc/en-us/articles/201774865-API-Introduction.
[21] Sebastian Elbaum, David Gable, and Gregg Rothermel. 2001. The Impact of Software Evolution on Code Coverage Information. In *ICSM*.
[22] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang. 2015. Making System User Interactive Tests Repeatable: When and What Should We Control?. In *ICSE*.
[23] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *ICSE-DEMO*.
[24] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *ISSTA*.
[25] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *ASE*.
[26] P. S. Kochhar, D. Lo, J. Lawall, and N. Nagappan. 2017. Code Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects. *IEEE Transactions on Reliability* (2017).
[27] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the Cost of Regression Testing in Practice: A Study of Java Projects Using Continuous Integration. In *ESEC/FSE 2017*.
[28] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *FSE*.
[29] Paul Marinescu, Petr Hosek, and Cristian Cadar. 2014. Covrig: A Framework for the Analysis of Code, Test, and Coverage Evolution in Real Software. In *ISSTA*.
[30] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale Continuous Testing. In *ICSE-SEIP '17*.
[31] A. Mockus, N. Nagappan, and T. T. Dinh-Trong. 2009. Test coverage and post-verification defects: A multiple case study. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*.
[32] Sebastiaan Oosterwaal, Arie van Deursen, Roberta Coelho, Anand Ashok Sawant, and Alberto Bacchelli. 2016. Visualizing Code and Coverage Changes for Code Review. In *FSE*.
[33] Klérisson V. R. Paixão, Crícia Z. Felício, Fernanda M. Delfim, and Marcelo de A. Maia. 2017. On the Interplay Between Non-functional Requirements and Builds on Continuous Integration. In *MSR*.
[34] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding Myths and Realities of Test-suite Evolution. In *FSE*.
[35] Steven P. Reiss. 2008. Tracking Source Locations. In *ICSE*.
[36] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and Combining Test-suite Reduction and Regression Test Selection. In *ESEC/FSE*.
[37] Mauricio Soto, Zack Coker, and Claire Le Goues. 2017. Analyzing the Impact of Social Attributes on Commit Integration Success. In *MSR*.
[38] Rodrigo Souza and Bruno Silva. 2017. Sentiment Analysis of Travis CI Builds. In *MSR*.
[39] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-based Fault Locators. In *ISSTA*.
[40] travis. [n. d.]. Travis-ci.com. https://travis-ci.org/.
[41] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. 2018. Adding Sparkle to Social Coding: An Empirical Study of Repository Badges in the npm Ecosystem. In *ICSE*.
[42] Zahy Volf and Edi Shmueli. 2017. Screening Heuristics for Project Gating Systems. In *ESEC/FSE*.
[43] Andy Zaidman, Bart Rompaey, Arie Deursen, and Serge Demeyer. 2011. Studying the Co-evolution of Production and Test Code in Open Source and Industrial Developer Test Processes Through Repository Mining. *Empirical Softw. Eng.* (2011).
[44] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The Impact of Continuous Integration on Other Software Development Practices: A Large-scale Empirical Study. In *ASE*.