Code, Test, and Coverage Evolution in Mature Software Systems: Changes over the Past Decade

Thomas Bailey
Imperial College London
London, United Kingdom
thomas.bailey0@outlook.com

Cristian Cadar
Imperial College London
London, United Kingdom
c.cadar@imperial.ac.uk

***Abstract*—Despite the central role of test suites in the software development process, there is surprisingly limited information on how code and tests co-evolve to exercise different parts of the codebase.**

A decade ago, the Covrig project examined the code, test, and coverage evolution in six mature open-source C/C++ projects, spanning a combined development time of twelve years. In this study, we significantly expand the analysis to nine mature C/C++ projects and a combined period of 78 years of development time. Our focus is on understanding how development practices have changed and how these changes have impacted the way in which software is tested.

We report on the co-evolution of code and tests; the adoption of CI, coverage, and fuzzing services; the changes to the overall code coverage achieved by developer test suites; the distribution of patch coverage across revisions; how code changes drive changes in coverage; and the occurrence and evolution of flaky tests.

Our large-scale study paints a mixed picture in terms of how software development and testing have changed over the past decade. While developers put more emphasis on software testing and the overall code coverage achieved by developer test suites has increased in most projects, coverage and fuzzing services are not widely adopted, many patches are still poorly tested, and the fraction of flaky tests has increased.

***Index Terms*—coverage evolution, patch coverage, CI adoption, fuzzing adoption, flaky tests**

I. Introduction

A good test suite, run continuously as the software evolves, is a key ingredient of a reliable and secure project. While what makes a good test suite is subject to debate, there are at least some characteristics which are generally acknowledged: A good test suite should at the very least achieve high statement and branch coverage (as a test suite cannot find bugs in code which it does not exercise), it should not be flaky (as debugging failures of flaky tests can waste significant time) and it should keep up with the changes in the code (as many bugs are introduced via insufficiently validated patches).

The Covrig project [1], published in ISSTA 2014, is the first to report how software and their test suites evolve over a significant number of revisions. In particular, the 2014 study examines six popular open-source systems written in C and C++, analysing the evolution of their code and test suites over a period of 250 code revisions.

In this work, we are revisiting and significantly extending the study in order to understand how development and testing practices have changed (or not) over the last decade. The study is timely, as the last decade has seen important changes in the open-source testing ecosystem, with wide availability of cloud-based continuous integration (CI) platforms such as TravisCI [2] and GitHub Actions [3], and the emergence of services such as OSS-Fuzz [4] and CIFuzz [5] that take advantage of the latest advances in fuzzing research.

Our study includes the six applications analysed in the Covrig project, to which we add three more. All these applications are widely-used, mature applications, which is why they have continued to be maintained and extended over the last decade. While Covrig looked at 250 code revisions per application, spanning a combined period of 12 years of development time, we are looking at up to 2,500 code revisions per application, spanning a combined period of 78 years.

To understand how the development and testing practices have changed over time, we report how the code and tests have changed in size; how test suite coverage has evolved and what has caused particular inflection points; how well patches[1] are tested; how different kinds of code changes drive changes in overall coverage; the extent to which projects employ CI, coverage and fuzzing services; and the prevalence of flaky tests over time.

A. Research Questions

Our research questions are summarised below. Four of them are taken or adapted from the Covrig paper [1] (RQ1, RQ3, RQ4, RQ6), one of them comes from a 2018 study by Hilton et al. [6] (RQ5), and two are new (RQ2, RQ7).

RQ1: **How do code and tests evolve?** Are they continuously increasing in size? Do they increase at the same rate?

RQ2: **Have projects adopted CI, coverage, and fuzzing services?** When has this taken place and what were the challenges encountered?

RQ3: **How does the overall code coverage evolve?** Does it increase steadily over time, or does it remain constant?

RQ4: **What is the distribution of patch coverage across revisions?** What fraction of a patch is covered by the regression test suite? Do smaller patches have higher coverage?

[1]In this paper, we use the terms “patch”, “commit” and “revision” interchangeably.

RQ5: **How do code changes drive changes in coverage?** Does code coverage change more because old code becomes tested, new tested code is added, or code is deleted?

RQ6: **How prevalent are flaky tests?** Do developers quickly identify and address flaky tests? Does running the test suite multiple times cover different lines of code?

RQ7: **Given the previous research questions, how has software testing changed in mature open-source C/C++ projects over the past decade?**

B. Findings and Contributions

The main findings and contributions of this project are:

1) We investigate the evolution of executable and test code over a combined period of 78 years in nine mature widely-used software projects written in C and C++. We find that all these projects increase in size, both in terms of code and tests. A majority of the projects place more emphasis on testing their code compared to a decade ago and the overall coverage achieved by the developer test suites has increased. However, the overall coverage in these projects is still low in many projects, with a majority of projects at under 50% branch coverage.
2) We find that most projects have adopted CI services in the last decade, with a few also tracking coverage and being fuzzed by the OSS-Fuzz [4] service.
3) We analyse the extent to which patches are tested, by tracking *patch coverage*, the percentage of lines of code in the patch which are covered by the developer test suite. We find that a significant number of patches are poorly tested or not tested at all, and that smaller patches have higher coverage overall.
4) We investigate the extend to which different types of code changes trigger changes in coverage. We find that a lot of code is added without being tested, code changes can cause covered code to become uncovered, and a lot of code which was covered at the beginning of our study is still uncovered at its end.
5) We analyse and quantify the presence of flaky tests and their impact on code coverage. Overall, we find that the number of flaky revisions has substantially increased, despite significant research work in this period on mitigating this problem.
6) We replicate the findings in the Covrig paper [1] and update and extend the Covrig infrastructure. We have contributed our changes to the open-source Covrig project, and make our experiments available as an artifact at https://srg.doc.ic.ac.uk/projects/covrig/.

II. Study Design and Methodology

Our study includes the six projects analysed by Covrig, namely Binutils, Git, Lighttpd2, Memcached, Redis, and ZeroMQ, to which we have added APR, Curl, and Vim. These are all mature, widely-used C/C++ projects, maintained and extended over many years, and used at various times by high-traffic websites such as Facebook, GitHub, StackOverflow, Twitter/X, Wikipedia, YouTube, and many other commercial and open-source software products.

We could have extended the study to contain many more projects written in additional languages, while keeping a relatively small number of revisions, as in similar studies [6]. However, we decided instead to scale up the study in terms of number of revisions studied, managing to study thousands instead of hundreds of revisions, over many years of development time.

A. Systems under Test

The nine projects analysed in our study are:

APR (Apache Portable Runtime, https://apr.apache.org/) provides a performant interface to platform-specific implementations of key software paradigms, such as atomic operations, locks and thread pools. It is used in many popular applications, such as the Apache HTTP web server, to achieve platform independence.

Binutils (https://www.gnu.org/software/binutils/) is a collection of utilities used for inspecting and editing object files, libraries and binary programs. As with Covrig, we analyse the twelve utilities in the `binutils` directory, comprising user-level programs for many UNIX distributions.

Curl (https://curl.se/) is a popular library that enables users to fetch data using URLs. It is portable and supports various protocols, including HTTP2/3, FTP and POP3.

Git (https://git-scm.com/) is the most popular version control system used in open-source development.

Lighttpd2 (https://redmine.lighttpd.net/projects/lighttpd2/) is a new version of the lightweight Lighttpd web server. It was meant as the latest development branch, but has now been “abandoned”, according to the website. There are still commits in the past year, but infrequent when compared to Lighttpd 1.4, their stable version. We have nevertheless decided to analyse Lighttpd2, to compare against the Covrig results, which used this version.

Memcached (https://memcached.org/) is a distributed memory object caching system used to reduce database load in high-performance web platforms.

Redis (https://redis.io/) is an in-memory data store which is most commonly used as a distributed key-value store.

ZeroMQ (https://zeromq.org/) is an asynchronous messaging library for distributed and concurrent applications. Covrig analysed the `4.x` branch of the software, so we are doing the same, although the latest commit to this branch was made in 2020.

Vim (https://www.vim.org/) is a popular open-source text editor, which can be used both on the command line and via a GUI.

Table I shows some basic statistics of the projects under study. ZeroMQ is written in C++, while the rest are C projects. As of the last revision studied, the projects have between 7.5k and 110k executable lines of code (ELOC), with a median

TABLE I: Code and test suite size of the studied projects, as of the last revision analysed.

	Code		Tests	
App	**Lang.**	**ELOC**	**Lang.**	**TLOC**
APR	C	17,890	C	22,578
Binutils	C	33,556	DejaGNU	20,350
Curl	C	31,965	Perl/Python	47,486
Git	C	109,698	C/Perl	159,316
Lighttpd2	C	27,334	Python	3,596
Memcached	C	11,776	C/Perl	10,058
Redis	C	39,287	Tcl	14,036
Vim	C	108,295	Vim Script	32,353
ZeroMQ	C++	7,546	C++	4,018

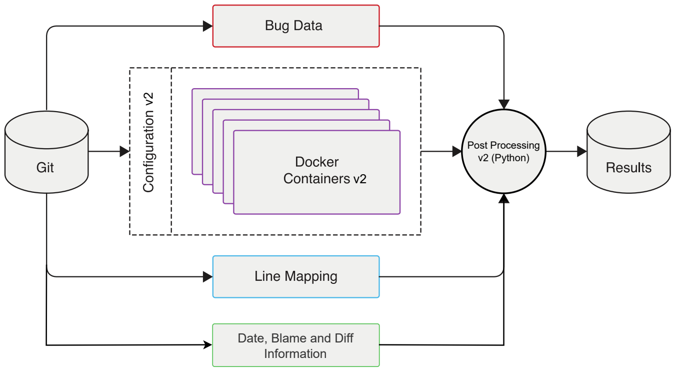

Fig. 1: Extended Covrig architecture (adapted from [1]).

value of 32k ELOC. The test suites of these projects are written in a variety of languages and vary between 3.6k and 159k test lines of code (TLOC), with a median of 20k TLOC. Overall, these projects have large test suites, with three of them (APR, Curl, Git) having more lines of code in their test suites than in the actual code. We retrieved `ELOC` from coverage runs using `lcov` [7], and counted the lines of code present in test files using `cloc` [8].

B. Analysis Framework

Our analysis extends Covrig, which is available as open source. Figure 1 shows the architecture of the extended Covrig infrastructure. Covrig pulls repository data from the projects' version control systems (Git), retrieving bug data and line mappings. It then starts Docker containers to run the test suites of individual revisions in isolation. Static project information and data from test suite runs are then combined to give coverage data.

We have upgraded the configuration of the Docker containers, rewrote the post-processing phase into Python (the original was written in Shell), and enhanced the collected data with additional information needed for the differential coverage analysis of §III-E. We have contributed these changes back to the project.

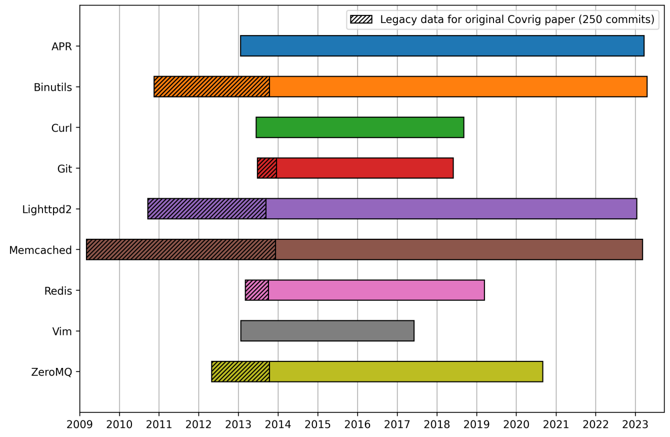

Fig. 2: Timespan for the revisions collected in each project. The hashed parts on the left indicate the revisions considered by Covrig.

C. Experimental Methodology

For the six projects studied in the Covrig paper, we started with the same 250 revisions and selected up to 2,500 revisions (including the original 250) until either we had reached this limit or the most recent revision of that project. For the other three projects, we started in 2013, around the time when the Covrig study was conducted.

The range of commits for each repository can be seen in Figure 2, where the hashed parts on the left indicate the revisions considered by Covrig.

As in Covrig, we only examined commits on the main branch of the repository and restricted ourselves to revisions that modify executable code and/or test code, and compile.

Table II shows the number of revisions collected for each project and the timespan during which they were developed. Overall, we considered development timespans varying between 4.4 and 14 years, with a combined timespan of 78 years.

Four of the projects—Curl, Git, Redis, Vim—reach the maximum number of 2,500 revisions (needing between 4 to 6 years to do so). The other projects reach their latest commits, with the total number of revisions ranging between 287 for ZeroMQ and 889 for Memcached.

The total number of revisions that compile correctly in the timespan considered is 23,569 revisions.[2] However, we only consider revisions that add or modify code or tests, which are the 13,610 revisions shown in Table II.

Most projects have the majority of commits in the `OK` bins as their test suites pass a high proportion of the time (as they should). Redis is an exception with tests failing frequently—we later learned this is due to inherent flakiness in its test suite (see §III-F). Curl also contains a majority of revisions with test failures. In this case, flakiness is less of an issue:

[2]When a sequence of commits that previously were on development branches are merged into the main branch, often only the last commit compiles. This kind of behaviour is common when authors do not squash commits on a merge. We encountered a reasonable number of compile errors in this process, but just ignored these revisions.

TABLE II: Revisions used in this study. `OK`: code compiles and all tests pass, `TF`: some tests fail, `TO`: test suite times out, `Total = OK + TF + TO`, `Span`: timespan for revisions.

App	Span	Total	OK	TF	TO
APR	122mo	435	429	5	1
Binutils	149mo	1,630	1,305	325	0
Curl	63mo	2,500	890	1,527	83
Git	60mo	2,500	1,843	623	34
Lighttpd2	148mo	369	332	37	0
Memcached	168mo	889	615	258	16
Redis	72mo	2,500	483	2,007	10
Vim	53mo	2,500	1,925	575	0
ZeroMQ	100mo	287	202	85	0
Total	78y	13,610	8,024	5,442	144

instead, usually a single test fails occasionally due to changed environmental conditions when running its tests, such as with certificates. For instance, certain sites now require `curl -k` to connect insecurely to certain webpages, whereas the test written at the time would not have required the flag. Git and Curl also have fair numbers of test timeouts—they tend to exist in a series of consecutive revisions, until they are eventually fixed.

Correctness. To enable an accurate comparison between the data obtained ten years ago in the Covrig project and our new data, it was important to make sure that our improvements to the Covrig infrastructure did not introduce any errors. To do so, we checked whether we succeeded in replicating the Covrig results from 2014. This comparison was over the set of 250 revisions which were presented in the original Covrig paper, using the archive of results made available by the project. For numerical results, we used Levene's test [9], [10], which produces a statistic to judge the equality of variances between two groups, using the medians of the groups. This test does not assume a normal distribution (which is instead the case in methods such as the F-test), and so is an appropriate metric to compare our two datasets. As a result of these checks, we discovered and fixed some problems—e.g., some tests were not run in ZeroMQ because the Libsodium dependency was not installed. Once these were fixed, the results were deemed statistically similar. One important exception are the test flakiness results, where we often see somewhat larger changes. This is expected given the nondeterministic nature of these experiments and the impact of different hardware on flakiness [11].

III. Empirical Study

We now present the results of our empirical study, with reference to the RQs introduced earlier in §I-A.

A. Code and Test Evolution

RQ1: How do code and tests evolve? Executable lines of code (ELOC) is a good base metric to outline the development activity within a codebase. Figure 3 shows the ELOC evolution for the nine repositories in our study. All projects have ended up with considerably more ELOC by the end of the study. Two notable drops are the one in ZeroMQ which was already studied in the Covrig paper (a result of refactoring and removal of code duplication), and a large drop in Binutils. On investigation, the latter is triggered by a single commit that removes support for

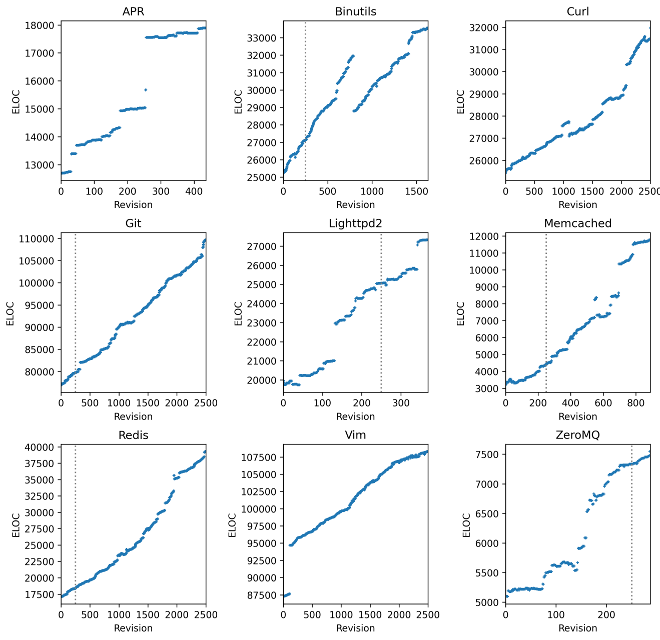

Fig. 3: Evolution of executable lines of code (ELOC). The vertical dotted line represents the extent of the previous study (at 250 revisions).

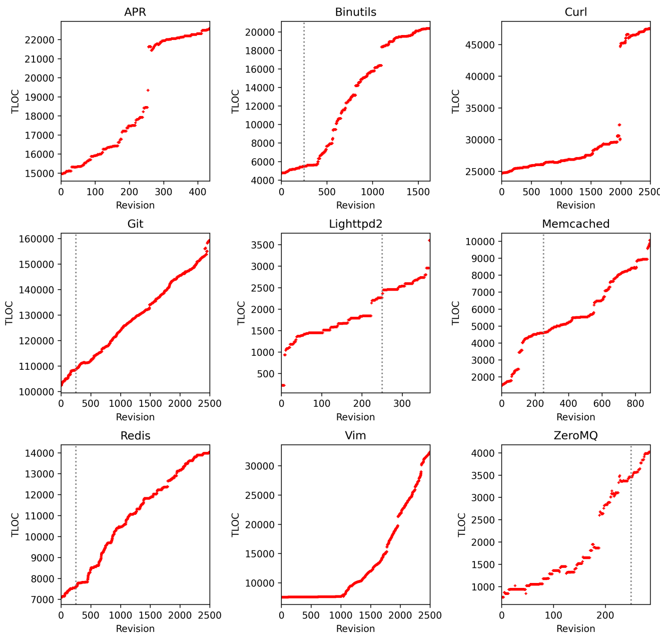

Fig. 4: Evolution of test lines of test code (TLOC). The vertical dotted line represents the extent of the previous study (at 250 revisions).

a now withdrawn standard for object modules (IEEE 695).[3] Apart from these cases, most revisions contribute rather than remove executable code. Some revisions add a large number of ELOC to the project: These typically correspond to the addition of new features, such as support for flash memory in Memcached, a new regular expression engine in Vim, and support for a new protocol in Curl.

Figure 4 shows the evolution of test lines of code (TLOC) in these projects. For many projects, we see a steady increase in TLOC, with fewer bursts than in the case of ELOC.

The biggest jump in TLOC is seen in a Curl revision. This is the result of importing a Python testing dependency into the package directly (`impacket`, as a result of doing a `pip install`),[4] which allows for better testing capabilities. This increases TLOC by around 12k in a single commit, and commits after this one employ these changes in new tests.[5] We will see in §III-C that coverage increases significantly in the same period as a result of this addition.

In both Binutils and Vim, we see a step change in the amount of effort dedicated to testing. While initially there are few test contributions in both projects, this changes dramatically in mid-2015 and beginning of 2016 respectively. For Vim in particular, the developers add over 30k TLOC in a period of around 18 months (starting roughly from revision `da59dd5`), quadrupling the size of their test suite.

One interesting observation is that compared to a decade ago [1], the majority of the projects (APR, Binutils, Curl, Git, Vim and ZeroMQ) add considerably more lines of test code than they do executable lines during the analysed period.

To summarise, the answer to RQ1 is that compared to a decade ago, significantly more effort has been dedicated to testing in many projects, with project such as Vim witnessing an overhaul in their test strategies. This is of course a positive development, and we will examine its impact shortly.

B. Adoption of CI, Coverage and Fuzzing Services

RQ2: Have projects adopted CI, coverage, and fuzzing services? To answer this RQ, we first investigated how projects use CI services to run their tests and keep track of coverage metrics. From inspecting the project repositories, we discovered that seven of the nine projects studied have a CI process in place. Table III provides a summary of the CI systems used, together with the dates and patches which put them in place.

As can be seen in the table, two of these projects (Binutils, Lighttpd2) still do not use a CI as of the time of writing, while the other seven projects adopted CI systems between 2013 (ZeroMQ) and 2021 (APR). While at the time of the Covrig study essentially none of the projects had a CI in place (only ZeroMQ adopted one toward the end of the study), now most of them have one. The most intense use of CI services is in Curl: we noticed over 130 different CI pipelines that are run for each pull request in 2023.

TABLE III: CI systems used by each project, including any legacy CIs: GH (GitHub Actions), CircleCI (Circle), CirrusCI (Cirrus), TravisCI (Travis), AppVeyor, Azure Pipeline (Azure), Zuul.

App	Legacy CI	Current CI	Start
APR	Travis	GH	2021[6]
Binutils	–	–	–
Curl	Travis, Zuul	GH, Circle, Cirrus, AppVeyor, Azure	2013[7]
Git	Travis	GH, Azure	2015[8]
Lighttpd2	–	–	–
Memcached	Travis	GH	2018[9]
Redis	Circle	GH	2019[10]
Vim	Travis	GH, Cirrus, AppVeyor	2015[11]
ZeroMQ	–	Travis	2013[12]

It is also interesting to see that initially most projects adopted TravisCI, but then switched to a different CI, most likely once TravisCI stopped being free for OSS projects [12]. Nowadays the most popular CI is GitHub Actions. This shows how OSS projects are often at the mercy of free offerings from commercial providers (not just for CI, but also code hosting, program analysis tools etc.), which adds extra burden to their maintenance.

Only two of the seven projects seem to explicitly track coverage during the CI runs. One of them is Vim, which subscribes to the Codecov [13] service.[13] The other one is Curl, which checks on each pull request that "code coverage does not shrink drastically"[14] and subscribed to the Coveralls [14] service in the past.

A challenge in keeping track of overall coverage is that some variations are expected over time (see §III-C and §III-E). For instance, Vim had to configure their CI to tolerate coverage decreases of less than 0.05%.[15] More radically, Curl decided to drop support for the Coveralls service in 2018, after observing large coverage fluctuations.[16]

During the past decade, fuzzing has seen tremendous progress [15], with parts of the industry starting to adopt it. In particular, Google's OSS-Fuzz [4] has established itself as a popular platform for fuzzing open-source software. OSS-Fuzz uses several state-of-the-art fuzzers to fuzz open-source projects at scale. To date, OSS-Fuzz has found over 10,000 security vulnerabilities and over 36,000 bugs while fuzzing over 1000 open-source projects [16].

[3]https://github.com/bminor/binutils-gdb/commit/fdef3943
[4]https://github.com/curl/curl/commit/f1609155d54c82b
[5]https://github.com/curl/curl/commit/a6f8d27efc7b0
[6]https://github.com/apache/apr/commit/c50db7ae
[7]https://github.com/curl/curl/commit/be31924f
[8]https://github.com/git/git/commit/522354d7
[9]https://github.com/memcached/memcached/commit/924dade6
[10]https://github.com/redis/redis/commit/a2ac5c38
[11]https://github.com/vim/vim/commit/0600f351
[12]https://github.com/zeromq/zeromq4-x/commit/59a164d2

[13]https://app.codecov.io/gh/vim/vim
[14]https://github.com/curl/curl/blob/master/tests/CI.md
[15]https://github.com/vim/vim/commit/845b7285
[16]https://github.com/curl/curl/commit/83c0e960

TABLE IV: OSS-Fuzz fuzzing statistics as of March 2024, from https://introspector.oss-fuzz.com.

App	Fuzzed	Coverage	Fuzz targets
APR	No	-	-
Binutils	Yes	32.17%	26
Curl	Yes	21.77%	17
Git	Yes	10.76%	10
Lighttpd2	Yes	34.71%	1
Memcached	No	-	-
Redis	No	-	-
Vim	No	-	-
ZeroMQ	No	-	-

However, despite their high popularity, only four of the nine projects analysed in this study have been fuzzed by OSS-Fuzz: Binutils, Curl, Git, and Lighttpd2. The other six projects, APR, Memcached, Redis, Vim and ZeroMQ, are absent. Furthermore, even for the four projects which are fuzzed, the coverage achieved by OSS-Fuzz is low, at under 35%. Table IV shows the OSS-Fuzz fuzzing statistics for these projects.

One likely reason for which many projects are absent and the ones present have low coverage is that fuzzing these projects is in fact not fully automated. Developers need to write *fuzz targets*, essentially test drivers to exercise different parts of the code. The last column of Table IV shows the number of fuzz targets written for each project. Systems for automatically generating fuzz targets would help address this challenge and have started to be created [17], [18], [19], although this topic is still under-explored compared to other aspects of fuzzing.

In conclusion, the last decade has seen most projects adopt CI services, with some also adopting coverage and fuzzing services. However, challenges remain: the availability of free CI services plays a big role in open-source development (as witnessed by the migration from TravisCI), coverage tracking is often imprecise (with Curl, for instance, giving up on coverage tracking because of this), and fuzzing requires significant manual work to be adopted (in the form of fuzz target development).

C. Coverage Evolution

RQ3: How does the overall code coverage evolve? We run each test suite five times and measure line and branch coverage. We follow the same methodology as in Covrig, with a line counted as covered if it is executed in at least one of the runs.

Figure 5 plots the evolution of line and branch coverage in these projects. The first observation is that line and branch coverage closely follow each other. Therefore, as in the Covrig paper, we mostly focus on line coverage from this point on.

Coverage levels vary significantly between projects, but across the revisions studied, almost all stay roughly constant or increase. Two projects with significant increases in coverage are Binutils and Vim, which as discussed in §III-A, have put significant effort into testing activities during this period.

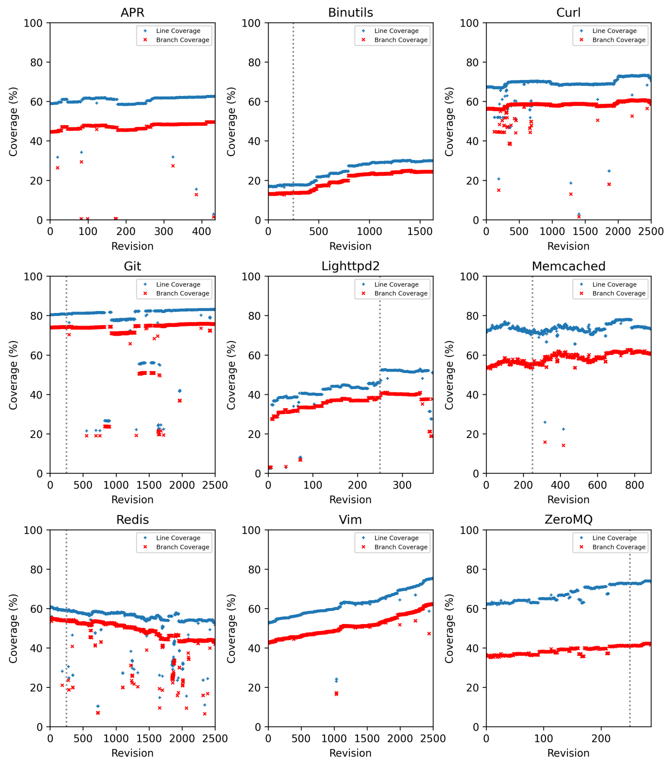

Fig. 5: Evolution of the overall line and branch coverage accumulated over five runs of each system's test suite.

The exception to the rule in terms of coverage evolution is Redis, where we observe a marked decrease in line coverage from 60.7% to 51.5%. We will discuss in more detail in §III-E the kinds of changes that have driven this decrease in coverage.

One noticeable feature in some of the graphs are the points below the coverage lines. These are revisions where due to bugs, tests do not run to completion.

D. Patch Coverage

RQ4: What is the distribution of patch coverage across revisions? Given a patch, the *patch size* is the number of ELOC added or modified by the patch, and the *patch coverage* is the fraction of these ELOC which are covered by the developer test suite. As with Covrig [1], we present in Figure 6 the patch coverage distribution, but extend it to highlight the 0% and 100% bins to match the format in the study by Hilton et al. [6].

As a decade ago, the patch coverage distribution appears to be bimodal, even when extended to more projects and more revisions. With the introduction of 0% and 100% bins, we see similar results to that of the Hilton et al [6], in that a large percentage of patches are either not covered at all, or are fully covered.

A large number of patches have poor coverage, under 25%, including a substantial number with 0% coverage. We find it disconcerting that in such popular software projects, developers submit so many patches without a single test exercising them. Perhaps even a simple system where such patches would be

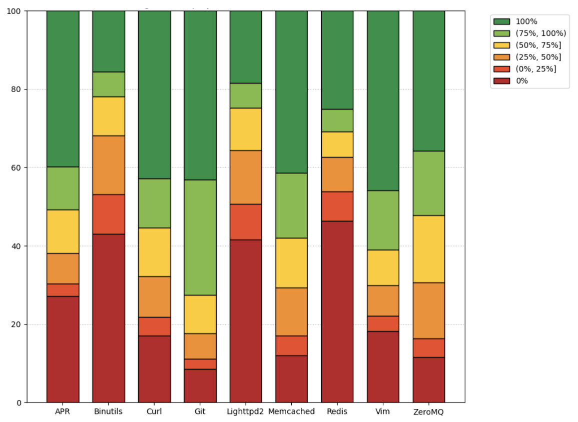

Fig. 6: Distribution of patch coverage for each project. This considers only revisions that contribute ELOC.

flagged, or even rejected by the CI, could partly address this problem.

One caveat of this representation is that it does not take into account the size of the patch (in terms of ELOC added or modified). This means a fully-covered patch with hundreds ELOC is given the same weight as a fully-covered patch with a single ELOC. To account for this, Figure 7 reports the distribution of patch sizes, with each size bin labeled with the overall patch coverage. Most projects have a similar distribution, with most patches having at most 10 ELOC. However, they all have large patches as well, including some which are often 1000 ELOC. The average patch coverage is usually higher for smaller patches, which is expected. However, there are exceptions, most notably in Git and Memcached—these projects have high overall coverage (see Figure 5), which benefits patches across sizes.

E. Explaining Changes to Coverage

RQ5: How do code changes drive changes in coverage? Similar to Hilton et al. [6], we plan to understand the impact of source code changes to coverage.

We consider the differential coverage categories introduced by Cox [20], which are more fine-grained than those used by Hilton et al. [6], but we merge several categories to get a final list of eight categories instead of twelve.[17] Our categories and their descriptions are summarised in Table V.

Figure 8 graphically shows how the changes in the studied projects divide among the different categories. The baseline covered and uncovered categories (CBC and UBC) are shown under the x-axis as they represent the lines of code whose status has not changed.

This graph immediately reveals several interesting observations regarding the evolution for these projects. For instance, there is unfortunately a lot of code which is added without being covered by any of the test cases (UNC), across all projects, and

[17]We merge GIC into GNC and UIC into UNC, as usually the inclusion of code is an addition to the codebase. Similarly, we merge EUB into DUB and ECB into DCB.

TABLE V: Differential coverage categories, taken from [20].

UNC	Uncovered New Code	Newly added code is not covered.
LBC	Lost Baseline Coverage	Code covered at the baseline revision is no longer covered.
UBC	Uncovered Baseline Code	Code uncovered at the baseline revision is still not covered.
GBC	Gained Baseline Coverage	Code not covered at the baseline revision is now covered.
GNC	Gained coverage New Code	Newly added code is covered.
CBC	Covered Baseline Code	Code covered at the baseline revision is still covered.
DUB	Deleted Uncovered Baseline code	Previously uncovered code has now been deleted.
DCB	Deleted Covered Baseline code	Previously covered code has now been deleted.

sometimes changes lead to covered code to become uncovered (LBC). Furthermore, a lot of code which was uncovered at the beginning of our study is still uncovered now (UBC), often more than a decade later.

Some projects do a better job at testing the newly added code, with most new code in Git and Vim in particular being covered by the test suite (GNC). Vim in particular is also noticeable in its efforts to add test cases to cover previously uncovered code (GBC). One aspect that these graphs hide though is the time when this coverage is gained: this could happen when code is committed, or any number of revisions later until the end of the study.

At the other end of the spectrum from Git and Vim, most of the code added to Redis is uncovered, which is one of the main causes of the overall decrease in coverage that we observed in §III-C. Binutils similarly adds a majority of code which is uncovered, but its testing effort (see §III-A) combined with the amount of uncovered code which is deleted (DUB) seems to have compensated and led to an overall gain in coverage (see §III-C). Nevertheless, overall it is worrying to see so much code being added without a single test case exercising it.

F. Flaky Tests

RQ6: How prevalent are flaky tests? In this question, we study the extent of test flakiness and its impact on coverage. To do so, as in Covrig, we run each test suite five times and record for each revision whether we get non-deterministic pass/fail test suite outcomes. Flakiness in these applications usually stems from multi-threaded code, network event ordering, and non-deterministic behaviour in the test harness.

The results are shown in Table VI. The fraction of revisions affected by flaky tests varies between only 0.5% for APR to almost 60% for Redis. Disappointingly, the relative number of revisions with flaky tests has increased compared to a decade ago. This is despite flakiness becoming an important topic in software engineering research since the time the Covrig paper was written [11].

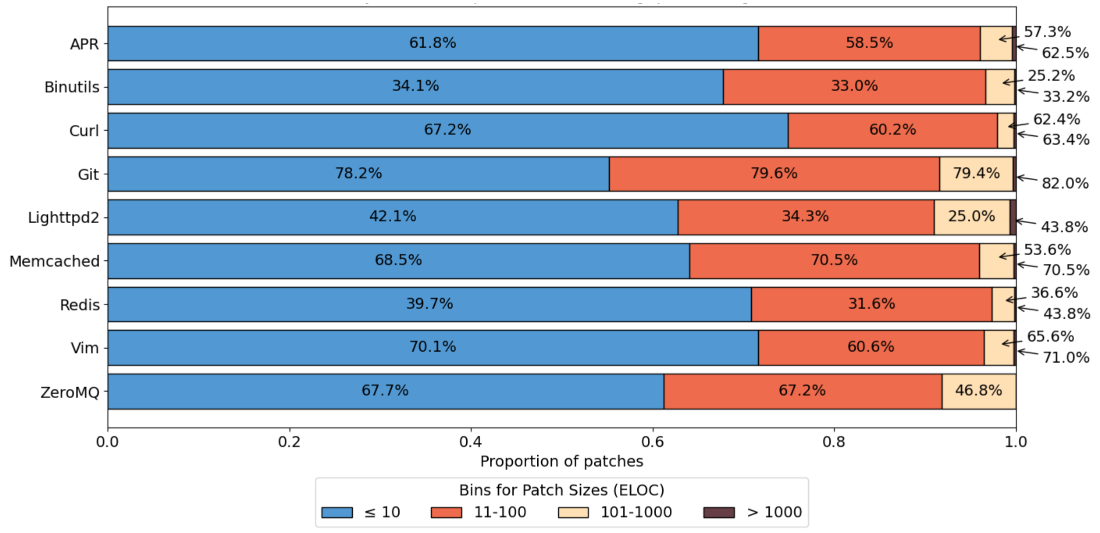

Fig. 7: Patch size distribution. Each bin is annotated with the average patch coverage across all the patches it contains.

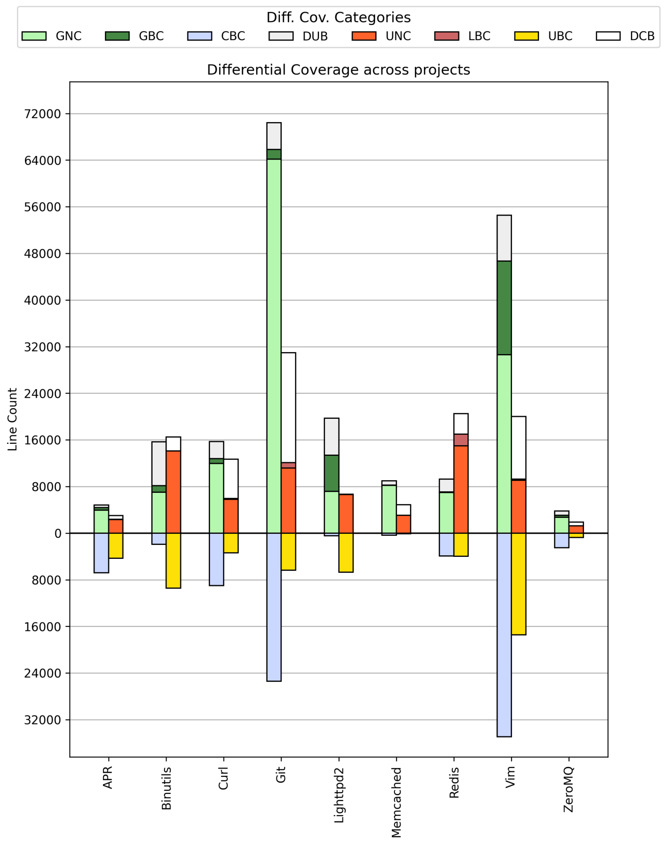

Fig. 8: Differential coverage by category, measured from the first revision in the study.

The largest increases are in Binutils and Redis. For Binutils, no flaky tests were recorded at the time of the Covrig study, while now 18.6% of the revisions (including the 250 ones studied by Covrig, which show no flakiness) have them. This seems to have started with a flaky test for the objdump utility.

TABLE VI: Number of flaky revisions, where the test suite reports mixed results across runs. Statistics on the ELOC non-deterministically executed in these revisions.

	Flaky revisions			**Flaky ELOC**	
	Revs	**%Revs**	**v. Covrig**	**Max**	**Median**
APR	2	0.5%	N/A	35	30.5
Binutils	303	18.6%	+18.6pp	0	0
Curl	99	4.1%	N/A	39	18
Git	43	1.7%	+1.3pp	26	14.5
Lighttpd2	10	2.7%	+2.3pp	28	7
Memcached	95	10.9%	+2.5pp	42	24
Redis	1490	59.8%	+53.4pp	290	50
Vim	49	2.0%	N/A	11	4
ZeroMQ	51	17.8%	+5pp	33	14

For Redis, the number of flaky revisions has increased substantially, with around 60% of the revisions studied now flaky. The number of flaky tests in Redis seems to be substantial, with relatively few fixes. For instance, we found one issue[18] which was first reported in 2013, but is still unresolved as of 2024.

To understand how quickly flaky tests are fixed, Figure 9 shows the distribution of flaky tests over time. One can observe clusters of consecutive revisions which are flaky: Flaky tests are introduced, persist for a number of revisions, and then are eventually fixed. Such clusters are particularly pronounced in Lighttpd2, Memcached and ZeroMQ.

G. Changes over the Last Decade

RQ7: Given the previous RQs, how has software testing changed in mature open-source C/C++ projects over the past decade? Our empirical study has extended the original Covrig study from a combined period of 12 years of development time to

[18]https://github.com/redis/redis/issues/1417

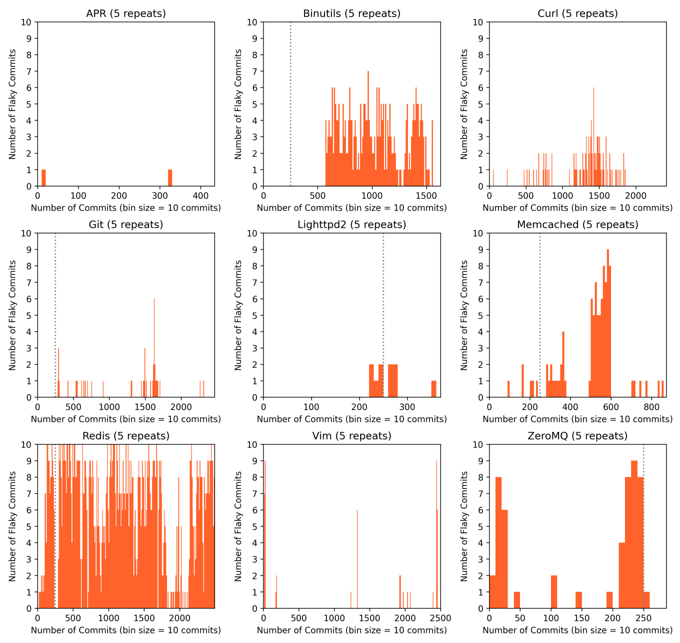

Fig. 9: Histograms for revisions with flaky tests. Each bin represents 10 commits, and the y-axis shows the number of commits in each bin that report mixed test suite results across the five repetitions.

78 years, by including many (and in some cases all) of the commits written in the last decade in nine mature software projects. In addition to some of the questions posed in the Covrig study,[19] we have also examined new research questions, including how projects have adopted CI services and how code changes have driven changes in coverage.

Our large-scale study paints a mixed picture of how the development of mature open-source C/C++ software projects has changed in the last decade.

Some changes are positive: Most projects have now adopted CI services such as GitHub Actions, and a few of them explicitly track changes to coverage and use the OSS-Fuzz fuzzing service. Thus, most projects are expanding more effort on testing, with some projects, such as Vim, seeing significant contributions in terms of new test cases. In fact, compared to a decade ago, we see much more test code added in a majority of the projects compared to executable code. Most projects at least double their TLOC count, and in some cases we see increases in the range of 3-6x. As a result, overall line and branch coverage increase in all projects except one, although different projects start from different baselines.

Other changes are unfortunately either neutral or negative. The code of all projects increases significantly over time, suggesting at least a certain amount of code bloat. Despite the increase in overall coverage, some projects still have unacceptably low coverage overall, with a majority of the projects at under 50% branch coverage at the end of the studied period. In terms of patch coverage, we observed the same bimodal distribution, with most of the patches either well tested or not tested (almost) at all. As expected, smaller patches are more likely to see higher coverage than large patches in most projects. The fact that so many patches continue to be poorly tested, or not tested at all, is particularly worrying, given that many critical bugs and security vulnerabilities are introduced by patches [21], [22] Another negative finding is that the proportion of flaky tests has increased considerably, despite the fact that during the same period we have seen a lot of research in detecting and mitigating flaky tests [11].

[19]We excluded a few RQs from the Covrig paper. For example, we excluded ”How many patches touch only code, only tests, none, or both?” as we found it less important, although we do have this data in the artifact. We also excluded the RQs on the correlation between coverage and bugs, due to the limited bug data available, which also led to inconclusive results in the Covrig paper.

H. *Threats to Validity*

The most important threat to the validity of this study is that it might not generalise to other software projects. To mitigate this threat, we added three more projects in addition to the six considered in the Covrig study. Nevertheless, the study is limited to longstanding, mature C/C++ projects. Furthermore, we did not analyse recent revisions in all projects, so it is possible that we may have missed development and testing changes that were implemented more recently.

Our study involves a huge amount of data and processing scripts, which might contain errors. We mitigated this threat by independently reproducing the Covrig results and applying statistical tests to confirm the correctness of our changes to the infrastructure, and by making our artifact available for inspection.

IV. Actionable Insights

Our findings, summarised in §I-B and discussed in detail throughout the paper, provide actionable insights to the research community and have the potential to influence future research.

Most testing research uses overall coverage as its main optimisation objective, but our study shows that even when overall coverage is high, code changes are often poorly tested. The research community should put more effort into incremental program analysis techniques that target code changes.

Despite the increased adoption of CI and fuzzing frameworks, open-source developers struggle to adopt them. One issue is the changing availability of free services offered by commercial providers. The establishment of standards could make it easier for open-source developers to change providers. Another challenge is related to the difficulty of writing fuzz drivers. While a lot of fuzzing research is focused on improving exploration heuristics, more effort should be placed on the under-explored area of automated fuzz driver generation.

Our study investigates the different ways in which code changes trigger changes in coverage—for example, that code changes can cause covered code to become uncovered. Future research could use this insight to automatically extract such dependencies and ensure they are properly tested.

Our study shows that flaky tests increase over time despite significant research work in this period on mitigating this problem. This raises the challenge of understanding the gap between the state of practice, and research on flaky test detection and mitigation.

V. Related Work

Our work focuses on understanding how code is tested in mature software projects, and how this has changed in the last decade. Compared to a majority of prior work that focuses on static metrics, we focus on dynamic metrics, based on various forms of coverage.

The Covrig paper [1] was the first to report line, branch and patch coverage information for a large number revisions in six mature C/C++ open-source projects. Our paper replicates this work and then significantly extends it to include more projects and up to ten times as many software revisions per project.

Four years after the Covrig study, Hilton et al. [6] analysed the patch coverage of 47 projects written in several different languages. The much larger number of projects considered means that the number of revisions considered in each is relatively small, around 166 on average. By contrast, we are keeping Covrig's focus on mature C/C++ projects and analyse only nine such projects, but look at an order of magnitude more revisions per project. One of the key contributions of the Hilton et al. study is to report the different ways in which code changes can affect coverage. Inspired by their study, we consider this as one of our research questions.

Earlier work on coverage has included much more limited studies. For instance, Elbaum et al. [23] are the first to report on the impact of code evolution on overall coverage, but their observations are limited to the small Space program and seven revisions of Bash. Zaidman et al. [24] investigate the co-evolution of code and tests in two open-source and one industrial Java projects, providing around ten data points for each project.

More broadly, understanding software evolution is an active area of research and prior work provides valuable findings that complement ours, in the areas of code and test evolution [24], [23], [25], [26], CI and fuzzing adoption [27], [28], [29], and flaky tests [30], [11], [31], [32], [33], among others. Compared to most prior work, our study is characterised by its long timespan and its focus on dynamic metrics, and particularly coverage, to understand how software development and testing have changed over the past decade.

VI. Conclusion

Despite the ubiquity of test suites in the software development process, there is surprisingly limited information on how code and tests co-evolve to exercise different parts of the codebase in real software projects.

We have studied nine mature C/C++ software projects over a total of 13,610 revisions spanning a combined period of 78 years of development time. In this time span, we report a continuous increase in both ELOC and TLOC, with test code often exceeding the amount of executable code added. The majority of the projects have adopted CI services, and a few projects also coverage and fuzzing services. By and large, more attention is given to testing, and overall coverage increases in this timespan for all but one project. On the negative side, many of the projects are still poorly tested, and an unacceptable fraction of patches are tested very little or not at all. Our differential coverage analysis shows that in many projects a lot of code is added without being tested and a large fraction of the code which was uncovered at the beginning of our study remains uncovered. Flaky tests have unfortunately increased for all the projects studied in Covrig, although they nevertheless affect a small number of revisions in most of the projects studied.

VII. Data Availability

We have contributed all our changes to the open-source Covrig repository[20] and are making our data and artifact available to the community [34]. Please visit https://srg.doc.ic.ac.uk/projects/covrig/ for detailed information.

Acknowledgements

This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement 819141).

References

[1] P. D. Marinescu, P. Hosek, and C. Cadar, "Covrig: A framework for the analysis of code, test, and coverage evolution in real software," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'14)*, Jul. 2014.
[2] Travis-CI, "Travis-CI," https://www.travis-ci.com/.
[3] GitHub, "GitHub Actions," https://github.com/features/actions.
[4] K. Serebryany, "OSS-Fuzz – Google's continuous fuzzing service for open source software," in *Proc. of the 26th USENIX Security Symposium (USENIX Security'16)*, Aug. 2017, invited talk.
[5] Google, "CIFuzz," https://google.github.io/oss-fuzz/getting-started/continuous-integration/, 2024.
[6] M. Hilton, J. Bell, and D. Marinov, "A large-scale study of test coverage evolution," in *Proc. of the 33rd IEEE International Conference on Automated Software Engineering (ASE'18)*, Sep. 2018.
[7] Linux Test Project, "LCOV," https://github.com/linux-test-project/lcov.
[8] ALDanial, "CLOC: Count lines of code," https://github.com/AlDanial/cloc.
[9] H. Levene, "Robust tests for equality of variances," *Contributions to probability and statistics*, pp. 278–292, 1960.
[10] M. B. Brown and A. B. Forsythe, "Robust tests for the equality of variances," *Journal of the American Statistical Association*, vol. 69, no. 346, pp. 364–367, 1974.
[11] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Transactions on Software Engineering Methodology (TOSEM)*, vol. 31, no. 1, Oct. 2021.
[12] J. Geerling, "Travis CI's new pricing plan threw a wrench in my open source works," https://www.jeffgeerling.com/blog/2020/travis-cis-new-pricing-plan-threw-wrench-my-open-source-works, Nov. 2020.
[13] Codecov Team, "Codecov." [Online]. Available: https://about.codecov.io/
[14] Coveralls Team, "Coveralls." [Online]. Available: https://coveralls.io/
[15] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 03, pp. 79–86, 2021.
[16] Google, "OSS-Fuzz trophies," https://github.com/google/oss-fuzz?tab=readme-ov-file#trophies, 2024.
[17] B. Domagoj, F. I. Stefan Bucur, Yaohui Chen, C. L. Tim King, Markus Kusano, L. Szekeres, and W. Wang, "FUDGE: Fuzz driver generation at scale," in *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, Aug. 2019.

[20]https://github.com/srg-imperial/covrig/

[18] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, “FuzzGen: Automatic fuzzer generation,” in *Proc. of the 29th USENIX Security Symposium (USENIX Security’20)*, Aug. 2020.

[19] D. Liu, J. Metzman, and O. Chang, “AI-powered fuzzing: Breaking the bug hunting barrier,” https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html, Aug. 2023.

[20] H. Cox, “Differential coverage: automating coverage analysis,” in *Proc. of the IEEE International Conference on Software Testing, Verification, and Validation (ICST’21)*, Apr. 2021.

[21] “Heartbleed bug,” http://heartbleed.com/, Apr. 2014.

[22] “Shellshock (software bug),” https://en.wikipedia.org/wiki/Shellshock_(software_bug), 2014.

[23] S. Elbaum, D. Gable, and G. Rothermel, “The impact of software evolution on code coverage information,” in *Proc. of the IEEE International Conference on Software Maintenance (ICSM’01)*, Nov. 2001.

[24] A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer, “Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining,” *Empirical Software Engineering (EMSE)*, vol. 16, no. 3, pp. 325–364, 2011.

[25] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, “Untangling spaghetti of evolutions in software histories to identify code and test co-evolutions,” in *Proc. of the IEEE International Conference on Software Maintenance and Evolution (ICSME’21)*, Sep. 2021.

[26] Z. Lubsen, A. Zaidman, and M. Pinzger, “Using association rules to study the co-evolution of production and test code,” in *Proc. of the 2009 International Working Conference on Mining Software Repositories (MSR’09)*, May 2009.

[27] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, “Usage, costs, and benefits of continuous integration in open-source projects,” in *Proc. of the 31th IEEE International Conference on Automated Software Engineering (ASE’16)*, Sep. 2016.

[28] Y. Gupta, Y. Khan, K. Gallaba, and S. McIntosh, “The impact of the adoption of continuous integration on developer attraction and retention,” in *Proc. of the 2017 International Working Conference on Mining Software Repositories (MSR’17)*, May 2017.

[29] O. Nourry, M. Kondo, M. Alfadel, S. McIntosh, and Y. Kamei, “Exploring the adoption of fuzz testing in open-source software: A case study of the Go community,” in *Proc. of the IEEE International Conference on Software Maintenance and Evolution (ICSME’24)*, Oct. 2024.

[30] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” in *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE’14)*, Nov. 2014.

[31] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, “A study on the lifecycle of flaky tests,” in *Proc. of the 42nd International Conference on Software Engineering (ICSE’20)*, May 2020.

[32] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, “An empirical analysis of UI-based flaky tests,” in *Proc. of the 43rd International Conference on Software Engineering (ICSE’21)*, May 2021.

[33] S. Thorve, C. Sreshtha, and N. Meng, “An empirical study of flaky tests in Android apps,” in *Proc. of the IEEE International Conference on Software Maintenance and Evolution (ICSME’18)*, Sep. 2018.

[34] “Artifact for ’*Code, Test, and Coverage Evolution in Mature Software Systems: Changes over the Past Decade*’.” [Online]. Available: https://zenodo.org/records/10937123