COVRIG: A Framework for the Analysis of Code, Test, and Coverage Evolution in Real Software

Paul Marinescu, Petr Hosek, Cristian Cadar
Department of Computing
Imperial College London, UK
{p.marinescu,p.hosek,c.cadar}@imperial.ac.uk

ABSTRACT

Software repositories provide rich information about the construction and evolution of software systems. While static data that can be mined directly from version control systems has been extensively studied, dynamic metrics concerning the execution of the software have received much less attention, due to the inherent difficulty of running and monitoring a large number of software versions.

In this paper, we present COVRIG, a flexible infrastructure that can be used to run each version of a system in isolation and collect static and dynamic software metrics, using a lightweight virtual machine environment that can be deployed on a cluster of local or cloud machines.

We use COVRIG to conduct an empirical study examining how code and tests co-evolve in six popular open-source systems. We report the main characteristics of software patches, analyse the evolution of program and patch coverage, assess the impact of nondeterminism on the execution of test suites, and investigate whether the coverage of code containing bugs and bug fixes is higher than average.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Monitors;
D.2.7 [**Distribution, Maintenance, and Enhancement**]: Version control

General Terms

Measurement, Reliability

Keywords

Patch characteristics, coverage evolution, latent patch coverage, nondeterministic coverage, bugs and fixes.

1. INTRODUCTION

Software repositories provide detailed information about the design and evolution of software systems. While there is a large body of work on mining software repositories—including a dedicated conference on the topic, the Working Conference on Mining Software Repositories (MSR)—past work has focused almost exclusively on *static metrics* (i.e. requiring no program execution). We suspect that the main reason behind the scarcity of studies focusing on dynamic metrics lies in the difficulty of running multiple software versions,[1] especially since doing so involves evolving dependencies and unstable (including non-compilable) versions. For example, prior work [41] cites the manual effort and the long time needed to run different revisions as the reason for reporting dynamic measurements for only a small number of versions.

While static metrics can provide useful insights into the construction and evolution of software, there are many software engineering aspects which require information about software executions. For example, the research community has invested a lot of effort in designing techniques for improving the testing of software patches, ranging from test suite prioritisation and selection algorithms [11, 30, 35] to program analysis techniques for test case generation and bug finding [1, 2, 20, 21, 27, 28, 36, 40] to methods for surviving errors introduced by patches at runtime [14]. Many of these techniques depend on the existence of a manual test suite, sometimes requiring the availability of a test exercising the patch [24, 37], sometimes making assumptions about the stability of program coverage or external behaviour over time [14, 29], other times using it as a starting point for exploration [10, 16, 22, 39], and often times employing it as a baseline for comparison [3, 6, 9, 26]. However, despite the key role that test suites play in software testing, it is surprising how few empirical studies one can find in the research literature regarding the co-evolution of test suites and code and their impact on the *execution* of real systems.

In this paper, we present COVRIG[2]—an infrastructure for mining software repositories, which makes it easy to extract both static and ***dynamic*** metrics. COVRIG makes use of lightweight virtual machine technology to run each version of a software application in isolation, on a large number of local or cloud machines. We use COVRIG to conduct an empirical study examining how programs evolve in terms of code, tests and coverage. More precisely, we have analysed the evolution of six popular software systems with a rich development history over a combined period of twelve years, with the goal of answering the following research questions:

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
ACM 978-1-4503-2645-2/14/07
http://dx.doi.org/10.1145/2610384.2610419

[1] In this paper, we use the terms *version* and *revision* interchangeably.
[2] The name emphasises one of the core aspects of the framework, its ability to measure coverage. COVRIG also means *bagel* in Romanian.

RQ1: Do executable and test code evolve in sync? Are coding and testing continuous, closely linked activities? Or do periods of intense development alternate with periods of testing?

RQ2: How many patches touch only code, only tests, none, or both? Are most code patches accompanied by a new or modified test case? How many patches modify neither executable code nor tests?

RQ3: What is the distribution of patch sizes? How spread out is each patch through the code? Are most patches small? How many different parts of the code does a patch touch? What is the median number of lines, hunks and files affected by a patch?

RQ4: Is test suite execution deterministic? Do tests fail nondeterministically? Does running the test suite multiple times cover different lines of code?

RQ5: How does the overall code coverage evolve? Is it stable over time? Does the overall coverage increase steadily over time, or does it remain constant? Are there revisions that significantly increase or decrease coverage?

RQ6: What is the distribution of patch coverage across revisions? What fraction of a patch is covered by the regression test suite? Does patch coverage depend on the size of the patch?

RQ7: What fraction of patch code is tested within a few revisions after it is added, i.e. what is the *latent patch coverage*? Are tests exercising recent patches added shortly after the patch was submitted? If so, how significant is this latent patch coverage?

RQ8: Are bug fixes better covered than other types of patches? Are most fixes thoroughly exercised by the regression suite? How many fixes are entirely executed?

RQ9: Is the coverage of buggy code less than average? Is code that contains bugs exercised less than other changes? Is coverage a reasonable indicator of code quality?

Overall, the main contributions of this paper are:

(1) A software repository mining infrastructure called Covrig, which can be used to run software versions in isolation, using a lightweight virtual machine environment that can be deployed on a private or public cloud.

(2) An analysis of the evolution of the execution of code and test suites in six popular open-source software systems over a combined period of twelve years. In particular, we believe this is the first study that reports patch coverage and coverage nondeterminism over a large number of program versions.

(3) A list of both theoretical and practical aspects related to mining dynamic information from software repositories, including revision granularity, non-compilable versions, and nondeterministic execution.

The rest of this paper is structured as follows. We first describe the Covrig infrastructure in Section 2, and then present our empirical study, structured around the research questions introduced above, in Section 3. We discuss related work in Section 4 and conclude in Section 5.

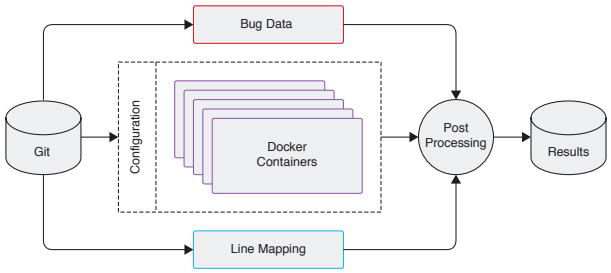

Figure 1: Covrig infrastructure.

2. COVRIG INFRASTRUCTURE

The overall architecture of the Covrig infrastructure is depicted in Figure 1. It contains a generic driver which iterates through all the revisions in a given range and invokes routines specific to each system to compile, run, and collect statistics of interest.

Lightweight software containers. Covrig employs software containers [33], an operating system-level virtualisation mechanism that provides the ability to run multiple isolated virtual Linux systems ("containers") atop a single host OS. When launched, Covrig starts by loading the selected range of revisions from the project's Git repository, and for each revision starts a new software container. The use of containers offers increased isolation and reproducibility guarantees by providing a consistent environment in which to run each software revision and ensuring that different revisions do not interfere with each other, e.g. by inadvertently leaving behind lock files or not properly freeing up resources.

The choice of lightweight OS-level virtualisation rather than more traditional virtual machines (e.g. KVM[3] or Xen[4]) reduces the performance penalty associated with spawning and tearing down VMs, operations performed for each revision analysed. To get a sense of this difference, we compared an LXC[5] container, which required under a second for these operations, with a Xen VM, which needed over a minute.

In our implementation, we use Docker[6] to create and manage the lower-level LXC containers, and deploy them on multiple local or cloud machines. Each container is used to configure, compile and test one program revision, as well as collect the metrics of interest, such as code size and coverage. The containers are remotely controlled through SSH using the Fabric[7] framework.

Configuration file. Covrig has a modular architecture, which makes it possible to analyse new systems with modest effort. A potential user of our infrastructure only needs to provide a Python configuration file describing the system. A minimal file provides the name of the system, its Git repository location, a method to compile the system, e.g. install dependencies and run the appropriate `make` command, and a method to run the regression tests, e.g. run the `make test` command. Finally, the configuration file can also specify an *end revision* and a specific number of revisions to analyse. For accurate test suite size measurements, the files or folders which make up the test suite can also be indicated.

[3] `http://www.linux-kvm.org/`
[4] `http://www.xenproject.org/`
[5] `http://linuxcontainers.org/`
[6] `https://www.docker.io/`
[7] `http://fabfile.org/`

For each revision, COVRIG collects several static and dynamic metrics. The static metrics are obtained either directly from the version control system (e.g. the number of lines of test code) or after compiling each revision (e.g. the number of executable lines of code). The dynamic metrics require running the regression tests (e.g. the overall line coverage or the regression test success status). Further information and graphs—including the ones presented in our empirical study—are automatically derived in the post-processing stage from these primary metrics using a set of scripts.

Bug data. One possible application of COVRIG is finding useful data about software bugs and correlating them with the static and dynamic metrics collected. For our study, we mined bug data from both software repositories and, where available, bug tracking systems. We automatically obtained a list of candidate bug-fixing revisions by iterating through the list of commits and checking the commit message for words such as *fix*, *bug* or *issue*, followed by a number representing the bug identifier. For example, a typical Memcached bug fix commit message looks like *"Issue 224 - check retval of main event loop"*. The regular expression that we used to identify these commits is similar to the ones used in prior work [12]: `(?:bug|issue|fix|resolve|close)\s*\#?\s?(\d+)`

Where possible, we confirmed that the bug identifier is valid by querying the associated bug tracking system. We further manually checked all reported revisions and confirmed that they included no false positives. While it is impossible to quantify the false negative rate without a knowledgeable developer manually checking all the revisions in a repository, we believe that the automatically obtained bug fixes create a representative subset of the fixes in the repository.

Line mapping. The ability to track how lines move and change across revisions is the cornerstone of many high-level software evolution analyses. A line mapping algorithm improves over the traditional `diff` algorithm by tracking the movement of individual lines rather than hunks. Conceptually, line mapping is a function which takes two revisions, *r1* and *r2*, and a program location described by a pair *(file name 1, line number 1)* associated with *r1*. The output is a pair *(file name 2, line number 2)* identifying the corresponding location in *r2*.

Our implementation of the line mapping algorithm is similar to the algorithms described in previous work [4, 17, 32, 38]. It makes use of the *Levenshtein edit distance* [19] to track line edits, and *tf–idf* [34] and *cosine similarity* [31] to track line movements. It also uses the *Hungarian algorithm* [18] to find the optimal matching of lines across versions. Compared to previous work, our implementation can also improve precision by using coverage information to filter non-executable lines.

In our study, we used line mapping to determine whether patches are tested within the next few revisions after they were created (§3.4).

Cloud deployment. To enable large-scale data collection and processing, we deployed COVRIG to our private cloud. We have built our system around a standard set of tools: Packer[8] for building custom Docker-enabled machine images, Vagrant[9] for controlling and provisioning the virtual machines based on these images, a Docker registry for serving COVRIG's Docker containers and a *fabfile* for orchestrating the entire cluster. The same set of tools and scripts can be used to deploy COVRIG to different private or public clouds.

[8] `http://www.packer.io/`
[9] `http://www.vagrantup.com/`

Table 1: Summary of applications used in our study. *ELOC* represents the number of executable lines of code and *TLOC* the number of lines in test files in the last revision analysed.

	Code		Tests	
App	**Lang.**	**ELOC**	**Lang.**	**TLOC**
Binutils	C	27,029	DejaGnu	5,186
Git	C	79,760	C/shell	108,464
Lighttpd	C	23,884	Python	2,440
Memcached	C	4,426	C/Perl	4,605
Redis	C	18,203	Tcl	7,589
ØMQ	C++	7,276	C++	3,460

3. EMPIRICAL STUDY

We used the COVRIG infrastructure to understand the evolution of six popular open-source applications written in C/C++, over a combined period of twelve years. Our empirical study has been successfully validated by the ISSTA artifact evaluation committee, and found to exceed expectations. The six evaluated applications are:

GNU Binutils[10] is a set of utilities for inspecting and modifying object files, libraries and binary programs. We selected for analysis the twelve utilities from the `binutils` folder (`addr2line`, `ar`, `cxxfilt`, `elfedit`, `nm`, `objcopy`, `objdump`, `ranlib`, `readelf`, `size`, `strings` and `strip`), which are standard user-level programs under many UNIX distributions.

Git[11] is one the most popular distributed version control systems used by the open-source developer community.

Lighttpd[12] is a lightweight web server used by several high-traffic websites such as Wikipedia and YouTube. We examined version 2, which is the latest development branch.

Memcached[13] is a general-purpose distributed memory caching system used by several popular sites such as Craigslist, Digg and Twitter.

Redis[14] is a popular key-value data store used by many well-known services such as GitHub and Flickr.

ØMQ[15] is a high-performance asynchronous messaging middleware library used by a number of organisations such as Los Alamos Labs, NASA and CERN.

The six applications are representative for C/C++ open-source code: GNU Binutils are user-level utilities, Git is a version control system, Lighttpd, Memcached and Redis are server applications, while ØMQ is a library. All applications include a regression test suite.

Basic characteristics. Table 1 shows some basic characteristics of these systems: the language in which the code and tests are written, the number of executable lines of code (ELOC) and the number of lines of test code (TLOC) in the last revision analysed. To accurately measure the number of ELOC, we leveraged the information stored by the compiler in `gcov` graph files, while to measure the number of TLOC we did a simple line count of the test files (using `cloc`, or `wc -l` when `cloc` cannot detect the file types).

[10] `http://www.gnu.org/software/binutils/`
[11] `http://git-scm.com/`
[12] `http://redmine.lighttpd.net/projects/lighttpd2/`
[13] `http://memcached.org/`
[14] `http://redis.io/`
[15] `http://zeromq.org/`

The code size for these applications varies from only 4,426 ELOC for Memcached to 79,760 ELOC for Git. The test code is written in a variety of languages and ranges from 2,440 lines of Python code for Lighttpd to 108,464 lines of C and shell code for Git. The test code is 36% larger than the application code in the case of Git, approximately as large as the application code for Memcached, around 40% of the application code for Redis and ØMQ, and only around 10% and 19% of the application code for Lighttpd and Binutils respectively. Running the test suite on the last version takes only a few seconds for Binutils, Lighttpd, and ØMQ, 110 seconds for Memcached, 315 seconds for Redis, and 30 minutes for Git, using a four-core Intel Xeon E3-1280 machine with 16 GB of RAM.

The version control system used by all these applications is Git. Four of these projects—Git, Memcached, Redis, and ØMQ—are hosted on the GitHub[16] online project site. The other two—Binutils and Lighttpd—use their own Git hosting.

Selection of revisions. Our goal was to select a comparable number of revisions across applications. The methodology was to start from the current version at the day of our experiments, and select an equal number of previous revisions for all systems. We only counted revisions which modify executable code, tests or both because this is what our analyses look at. We decided to select 250 such revisions from each system because some systems had non-trivial dependency issues further back than this, which prevented us from properly compiling or running them. We still had to install the correct dependencies where appropriate, e.g. downgrade `libev` for older versions of Lighttpd and `libevent` for Memcached.

Note that not all revisions compile, either due to development errors or portability issues (e.g. system header files differing across OS distributions). Redis has the largest number of such transient compilation errors—38. The prevailing reasons are missing `#include` directives, e.g. `unistd.h` for the `sleep` function, and compiler warnings subsequently treated as errors. The missing `#include` directives most likely slipped past the developers because on some systems other headers cause the missing ones to be indirectly included. The compiler warnings were generated because newer compiler versions, such as the one that we used, are more pedantic. Other reasons include forgotten files and even missing semicolons.

We decided to fix the errors which had not likely been seen at the time a particular revision was created, for example by adding the compile flag `-Wno-error` in Binutils so that warnings do not terminate the build process. In all situations when we could not compile a revision, we rolled over the changes to the next revisions until we found one where compilation was successful. Revisions which do not successfully compile are not counted towards the 250 limit.

Another important decision concerns the granularity of the revisions being considered. Modern decentralised software repositories based on version control systems such as Git do not have a linear structure and the development history is a directed acyclic graph rather than a simple chain. Different development styles generate different development histories; for example, Git, Redis and ØMQ exhibit a large amount of branching and merging while the other three systems have a mostly linear history. Our decision was to focus on the main branch, and treat each merge into it as a single revision. In other words, we considered each feature branch a single indivisible unit. Our motivation for this decision was twofold: first, development branches are often spawned by individual developers in order to work on a certain issue and are often "private" until they are merged into the main branch. As a result, sub-revisions in such branches are often unusable or even non-compilable, reflecting work-in-progress. Second, the main branch is generally the one tracked by most users, therefore analysing revisions at this level is a good match in terms of understanding what problems are seen in the field. This being said, there are certainly development styles and/or research questions that would require tracking additional branches; however, we believe that for our benchmarks and research questions this level of granularity provides meaningful answers.

[16] `https://github.com/`

Table 2: Revisions used in our study. *OK*: code compiles and tests complete successfully, *TF*: some tests fail, *TO*: tests time out, *CF*: compilation fails, *Time*: the number of months analysed.

	OK+TF+TO=250				
App	**OK**	**TF**	**TO**	**CF**	**Time**
Binutils	240	10	0	25	35mo
Git	249	0	1	0	5mo
Lighttpd	145	105	0	13	36mo
Memcached	206	43	1	5	47mo
Redis	211	38	1	38	6mo
ØMQ	171	79	0	11	17mo

Table 2 summarises the revisions that we selected: they are grouped into those that compile and pass all the tests (*OK*), compile but fail some tests (*TF*), and compile but time out while running the test suite (*TO*). The time limit that we enforced was empirically selected for each system such that it is large enough to allow a correct revision to complete all tests. As shown in the table, timeouts were a rare occurrence, with at most one occurrence per application.

Table 2 also shows the development time span considered, which ranges from only 5-6 months for Git and Redis, which had a fast-paced development during this period, to almost 4 years for Memcached. The age of the projects at the first version that we analysed ranges from a little over 2 years for Lighttpd (version 2), to 11 years for Binutils.

Setup. All the programs analysed were compiled to record coverage information. In addition, we disabled compiler optimisations, which generally interact poorly with coverage measurements. For this we used existing build targets and configuration options if available, otherwise we configured the application with the flags `CFLAGS='-O0 -coverage'` and `LDFLAGS=-coverage`. All code from the system headers, i.e. `/usr/include/` was excluded from the results.

Each revision was run in a virtualised environment based on the 64-bit version of Ubuntu 12.10 (12.04.3 for Git) running inside an LXC container. To take advantage of the inherent parallelism of this approach, the containers were spawned in one of 28 long-running Xen VMs, each with a 4 Ghz CPU, 6 GB of RAM, and 20 GB of storage, running a 64-bit version of Ubuntu 12.04.3.

The following subsections present the main findings of our analysis. They first reiterate and then examine in detail our target research questions (RQs).

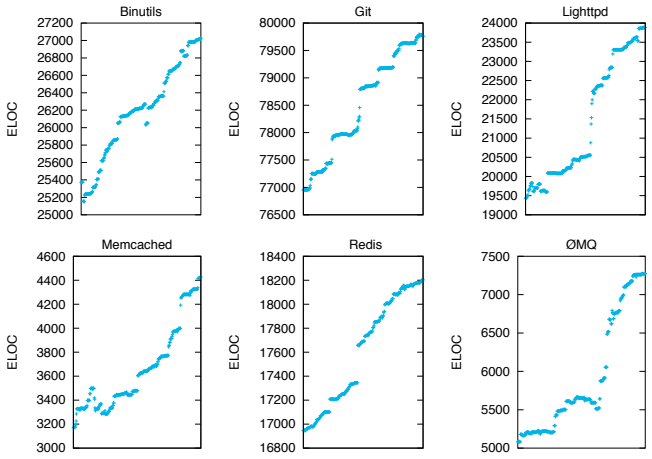

Figure 2: Evolution of executable lines of code.

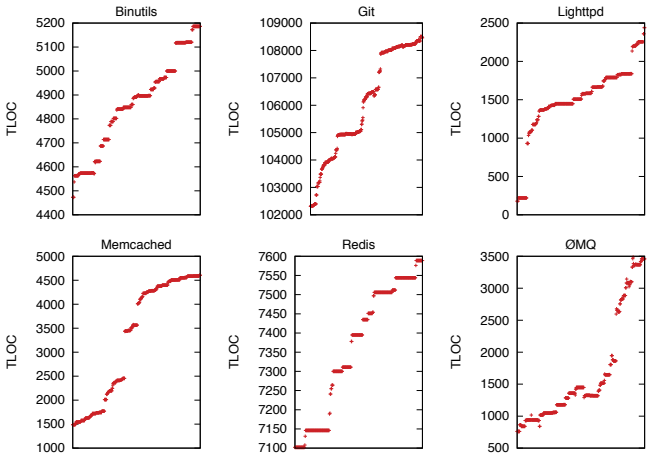

Figure 3: Evolution of textual lines of test code.

3.1 Code and Test Evolution

RQ1: Do executable and test code evolve in sync?

Figure 2 shows the evolution of each system in terms of ELOC. As discussed above, we measured the number of ELOC in each revision by using the information stored in `gcov` graph files. This eliminates all lines which were not compiled, such as those targeting architectures different from our machine. One of the main reasons for which we have decided to measure ELOC rather than other similar metrics is that they can be easily related to the dynamic metrics, such as patch coverage, presented in Sections 3.3 and 3.4.

As evident from Figure 2, all six systems grow over time, with periods of intense development that increase the ELOC significantly, alternating with periods of code tuning and testing, where the code size increases at a slower pace. It is interesting to note that there are also several revisions where the number of ELOC decreases (e.g. in ØMQ): upon manual inspection, we noticed that they relate to refactorings such as using macros or removing duplicate code.

The total number of ELOC added or modified varies between 2,296 for Redis and 10,834 for Lighttpd, while the end-to-end difference in ELOC varies between 1,257 for Memcached and 4,466 for Lighttpd.

Figure 3 presents the evolution of the size of the test suite in each system, measured in textual lines of test code (TLOC). For each system, we manually identified the files responsible for regression testing and recorded the number of lines contained in them at each revision. It can be seen that test evolution is less dynamic than code evolution, developers adding less test code than regular code.

To better understand the co-evolution of executable and test code, we merged the above data and plotted in Figure 4 only whether a revision changes the code (tests) or not: that is, the *Code* and *Test* values increase by one when a change is made to the code, respectively to the tests in a revision, and stay constant otherwise. As it can be seen, while the *Code* line smoothly increases over time, the *Test* line frequently stays constant across revisions, indicating that testing is often a *phased* activity [41], that takes place only at certain times during development. One exception is Git, where code and tests evolve more *synchronously*, with a large number of revisions modifying both code and tests.

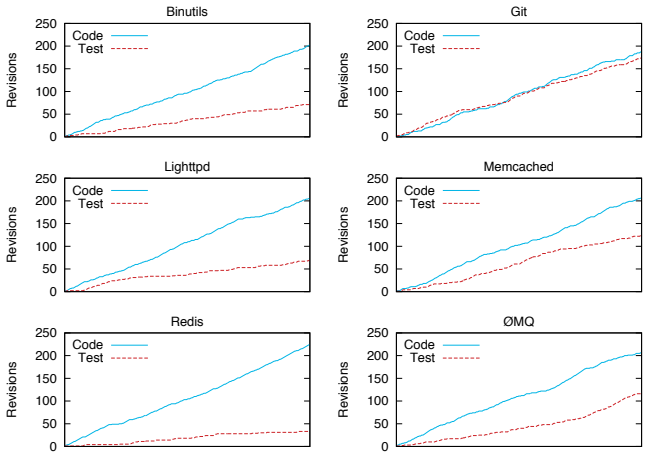

Figure 4: Co-evolution of executable and test code. Each increment represents a change.

3.2 Main Patch Characteristics

RQ2: How many patches touch only code, only tests, none, or both?

Each revision defines a *patch*, which consists of the totality of changes introduced by that revision. Software patches represent the building blocks of software evolution, and can affect code, regression tests, or infrastructure components such as build scripts, and play a variety of roles, including bug fixing, feature addition, and better testing.

Figure 5 classifies patches into those that modify executable application code but not the test code (*Code only*), those that modify both executable application code and test code (*Code+Test*), and those that modify test code but not executable application code (*Test only*). Note that for each application, these three values sum to 250, since we only selected revisions which modify executable code and/or tests, as discussed previously. Figure 5 also shows the number of patches from the time span analysed that modify neither executable program code nor tests (*Other*).

The first observation is that a substantial amount of time is spent in maintenance activities that do not involve code or tests. For example, during the period analysed, in addition

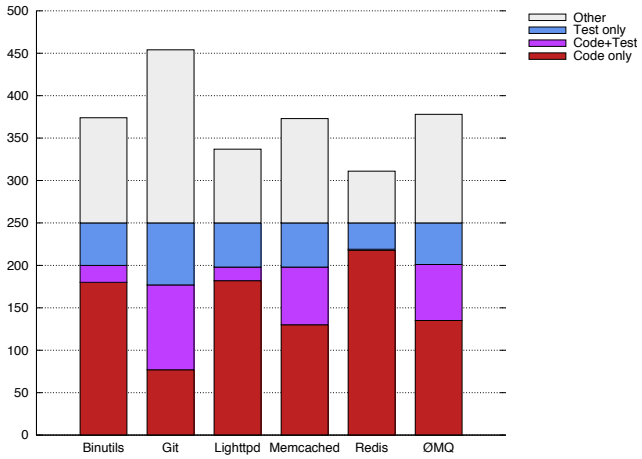

Figure 5: Breakdown of patches by type: affecting executable application code but not test code, affecting both, affecting only test code, and neither.

to the 250 target patches, there were around 120 additional such patches in Binutils, Memcached and ØMQ, and 204 in Git. Note that some of these patches may modify code that is excluded during preprocessing on our machine, but most cases involved changes to build scripts, documentation, and other similar software artefacts.

From the 250 patches selected for each application, the majority only modify code, with a relatively small number of patches (73 in Git, and under 52 for the others) touching only tests. The number of revisions that modify both code and tests can offer some indication of the development style used: at one end of the spectrum there is Redis, with only one such patch, suggesting that coding and testing are quite separate activities; at the other end there is Git, with 100 such patches, suggesting a development discipline in which code changes are frequently accompanied by a test case.

RQ3: What is the distribution of patch sizes? How spread out is each patch through the code?

The size of a patch and the number of locations that it affects can provide useful guidance for longitudinal testing techniques. The *Lines* column in Table 3 provides information about the size of the executable code patches analysed in each system, measured in ELOC. Note that our measurements ignore changes in the amount of whitespace, e.g. whitespace at the end of the line, because our target programming languages, C and C++, are insensitive to such modifications. Most patches are small, with the median number of ELOC ranging from 4 to 7.

To understand the degree to which patches are spread out through the code, we also recorded the number of areas in the code—*hunks* in Git terminology—and the number of files containing executable code which suffered changes. More formally, a hunk groups together all the lines added or modified in a patch which are at a distance smaller than the *context size*. We used the default unified diff format with a context size of three lines when computing the hunks.[17] The *Hunks* column in Table 3 shows that the median number of hunks varies between 2 and 4.

[17]See `http://www.gnu.org/software/diffutils/manual/html_node/` for more details.

Table 3: The median number of executable lines, hunks from executable files, and executable files in a patch. Only data from patches which add or modify executable code is considered.

App	Lines	Hunks	Files
Binutils	5	2	1
Git	7	3	1
Lighttpd	6	3	1
Memcached	6	3	1
Redis	4	2	1
ØMQ	7	4	2

Table 4: Number of revisions where the test suite nondeterministically succeeds/fails, and the maximum, median and average number of lines which are nondeterministically executed in a revision.

	Nondet.	Nondet. ELOC		
App.	Result	Max	Median	Average
Binutils	0	0	0	0
Git	1	23	13	11.80
Lighttpd	1	37	10	13.01
Memcached	21	22	8.5	7.55
Redis	16	71	23	30.98
ØMQ	32	47	27	19.52

Finally, the median number of files modified by a patch is only 1 for all benchmarks with the exception of ØMQ, for which it is 2. The fraction of patches that modify a single file is, in increasing order, 48.7% for ØMQ, 58.7% for Git, 65.1% for Lighttpd, 66.6% for Memcached, 84.9% for Redis, and 88.5% for Binutils.

3.3 Overall Code Coverage

RQ4: Is test suite execution deterministic?

As a large part of our study focuses on coverage metrics, we first investigate whether code coverage is deterministic, i.e. whether the regression test suite in a given revision achieves the same coverage every time it is executed. As we show, nondeterminism has implications in the reproducibility of test results—including the ones that we report—and the fault detection capability of the tests.

We measured the overall coverage achieved by the regression test suite using `gcov`. Interestingly, we found that all the programs from our experiments except Binutils are nondeterministic, obtaining slightly different coverage in each run of the test suite. Therefore, we first quantified this nondeterminism by running the test suite five times for each revision and measuring how many revisions obtained mixed results, i.e. one run reported success while another reported failure. We were surprised to see a fair number of revisions displaying this behaviour, as listed in Table 4 under the column *Nondet Result*.

We further counted for each pair of runs the number of lines whose coverage status differs. We used a 0/1 metric, i.e. we only considered a difference when one of the five runs never executes a line and another one executes it. We only did this for revisions in which the test suite completes successfully to avoid spurious results that would occur if we compare a run which completed with one that was prematurely terminated

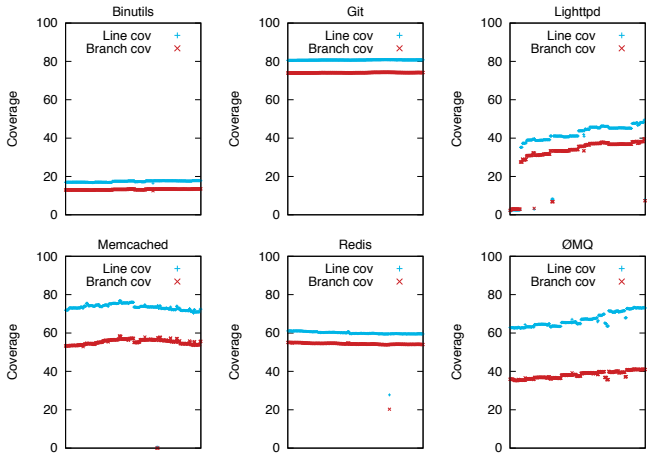

Figure 6: Evolution of the overall line and branch coverage.

due to a failure. As shown in Table 4, Binutils seems to be completely deterministic with respect to its test suite, while Redis, for example, contains on average 30.98 lines that are nondeterministically executed.

We manually investigated the nondeterminism and pinpointed three sources: (1) multi-threaded code, (2) ordering of network events, and (3) nondeterminism in the test harness. As an example from the first category, the test from ØMQ `test_shutdown_stress` creates 100 threads to check the connection shutdown sequence. In a small percentage of runs, this test exposes a race condition.[18] As an example in the third category, some Redis tests generate and store random integers, nondeterministically executing the code implementing the internal database data structures. The Memcached test `expirations.t` is representative of tests that make assumptions based on hardcoded wall-clock time values, which cause failures under certain circumstances. The test timings were previously adjusted[19] in response to failures under Solaris' `dtrace` and we believe that some of the failures that we encountered were influenced by the Docker environment.

The potential drawback of nondeterminism is the inability of coverage comparison across revisions, lack of reproducibility and consequent difficulty in debugging. Developers and researchers relying on test suite executions should take nondeterminism into account, by either quantifying its effects, or by using tools that enforce deterministic execution across versions [14], as appropriate. Tests with nondeterministic executions—such as the ones presented above—are fragile and should be rewritten. For example, tests relying on wall-clock time could be rewritten as event-based tests [15].

RQ5: How does the overall code coverage evolve? Is it stable over time?

When reporting the overall coverage numbers, we accumulated the coverage information across all five runs.[20] Therefore, the results aim to count a line as covered if the test suite *may* execute it. The blue (upper) lines in Figure 6 plot the overall line coverage for all benchmarks. It can be seen that coverage level varies significantly, with Binutils at one end achieving only 17.39% coverage on average, and Git at the other achieving 80.74%, while in-between Lighttpd achieves 39.08%, Redis 59.97%, ØMQ 66.88%, and Memcached 72.98%.

One interesting question is whether coverage stays constant over time. As evident from Figure 6, for Binutils, Git, Memcached, and Redis, the overall coverage remains stable over time, with their coverage changing with less than 2 percentage points within the analysed period. On the other hand, the coverage in Lighttpd and ØMQ increases significantly during the time span considered, with Lighttpd increasing from only 2.02% to 49.37% (ignoring the last two versions for which the regression suite fails), and ØMQ increasing from 62.89% to 73.04%. An interesting observation is that coverage evolution is not strongly correlated to the co-evolution of executable and test code (RQ1). Even when testing is a phased activity, coverage remains constant because the already existing tests execute part of the newly added code.

One may notice that a few revisions from Lighttpd, Memcached and Redis cause a sudden decrease in coverage. This happens because either bugs in the program or in the test suite prevent the regression tests from successfully running to completion. In all cases, these bugs are fixed after just a few revisions.

Figure 6 also shows that branch coverage closely follows line coverage. The difference between line and branch coverage is relatively small, with the exception of Memcached and ØMQ. The larger difference is due to the frequent use of certain code patterns which generate multiple branches on a single line, such as the one shown in Listing 1, which comes from the ØMQ codebase. The `zmq_assert` macro is expanded into a single line resulting in 100% line coverage, but only 50% branch coverage when executed in a typical run of the program (where assertions do not fail).

The fact that line and branch coverage closely follow one another suggests that in many situations only one of these two metrics might be needed. For this reason, in the remainder of the paper, we report only line coverage.

Finally, we have looked at the impact on coverage of revisions that only add or modify tests (*Test only* in Figure 5). An interesting observation is that many of these revisions bring no coverage improvements. For instance, in Lighttpd only 26 out of 52 such revisions improve coverage. The other 26 either do not affect coverage (18 revisions) or decrease it (8 revisions). The revisions which do not affect coverage can be a sign of test-driven development, i.e. tests are added before the code which they are intended to exercise. The revisions which decrease coverage are either a symptom

```
#define zmq_assert(x) \
  do {\
    if (unlikely (!(x))) {\
      fprintf (stderr, "Assertion failed: %s (%s:%d)\n", #x, \
          __FILE__, __LINE__);\
      zmq::zmq_abort (#x);\
    }\
  } while (false)
```

Listing 1: Example of an assertion macro used in ØMQ codebase.

[18] `https://github.com/zeromq/zeromq4-x/commit/de239f3`
[19] `https://github.com/memcached/memcached/commit/890e3cd`
[20] With the exception of Git, where for convenience we considered a single run, as the number of lines affected by nondeterminism represent less than 0.3% of the total codebase.

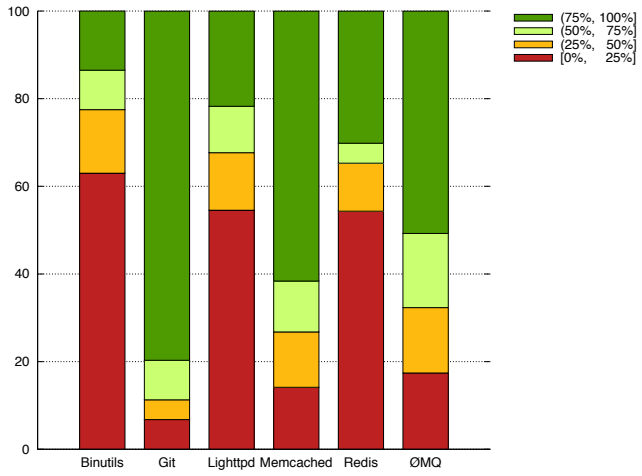

Figure 7: Patch coverage distribution. Each colour represents a range of coverage values with the bar size indicating the percentage of patches whose coverage lies in the respective range.

of nondeterminism—six of them, with small decreases in coverage—or expose bugs or bigger changes in the testing infrastructure (the other two). These two revisions exhibit a drop in coverage of several thousands lines of code. In one case, the tests cause Lighttpd to time out, which leads to a forceful termination and loss of coverage data. This problem is promptly fixed in the next revision. In the other case, the new tests require a specific (new) module to be built into the server, terminating the entire test suite prematurely otherwise.

3.4 Patch Coverage

RQ6: What is the distribution of patch coverage across revisions?

We define *patch coverage* as the ratio between the number of executed lines of code added or modified by a patch and the total number of executable lines in the patch, measured in the revision that adds the patch.

Figure 7 shows the distribution of patch coverage for each system. Each column corresponds to all patches which affect executable code in a system, normalised to 100%. The patches are further grouped into four categories depending on their coverage. As it can be observed, the patch coverage distribution is bimodal across applications: the majority of the patches in Git, Memcached and ØMQ achieve over 75% coverage, while the majority of the patches in Binutils, Lighttpd and Redis achieve under 25%. One interesting aspect is that for all applications, there are relatively few patches with coverage in the middle ranges: most of them are either poorly (≤25%) or thoroughly (>75%) covered.

Table 5 presents the same patch coverage statistics, but with the patches bucketed by their size into three categories: less than or equal to 10 ELOC, between 11 and 100 ELOC, and greater than 100 ELOC. For all benchmarks, patches are distributed similarly across buckets, with the majority of patches having ≤10 ELOC and only a few exceeding 100 ELOC. Across the board, the average coverage of patches

Table 5: Overall patch coverage bucketed by the size of the patch in ELOC. NP is the number of patches in the bucket and C is their overall coverage. Only patches which add or modify executable code are considered.

	≤10		11-100		>100	
App	NP	C	NP	C	NP	C
Binutils	128	19.5%	63	25.0%	9	16.8%
Git	102	87.4%	65	82.4%	10	87.0%
Lighttpd	120	41.9%	58	31.3%	20	30.8%
Memcached	122	73.7%	73	70.8%	3	57.0%
Redis	164	33.8%	51	34.8%	4	21.1%
ØMQ	119	65.5%	64	68.0%	18	48.9%

Table 6: Overall latent patch coverage: the fraction of lines of code in all patches that are only executed by the regression suite in the next 1, 5 or 10 revisions. The overall patch coverage is listed for comparison.

App	Overall	+1	+5	+10
Binutils	21.2%	0.1%	0.3%	0.3%
Git	85.1%	0%	0%	0%
Lighttpd	31.3%	0.9%	5.0%	6.1%
Memcached	68.9%	2.1%	3.4%	3.5%
Redis	30.4%	5.2%	5.5%	6.4%
ØMQ	56.9%	0.4%	3.5%	6.0%

with ≤10 ELOC is higher than for those with >100 ELOC, but the coverage of the middle-size category varies.

Finally, the *Overall* column in Table 6 shows the overall patch coverage, i.e. the percentage of covered ELOC across all patches. For Binutils, Git and Memcached, it is within five percentage points from the overall program coverage, while for the other benchmarks it is substantially lower—for example, the average overall program coverage in Redis is 59.97%, while the overall patch coverage is only 30.4%.

RQ7: What fraction of patch code is tested within a few revisions after it is added, i.e. what is the *latent patch coverage*?

In some projects, tests exercising the patch are added only after the code has been submitted, or the patch is only enabled (e.g. by changing the value of a configuration parameter) after related patches or tests have been added. To account for this development style, we also recorded the number of ELOC in each patch which are only covered in the next few revisions (we considered up to ten subsequent revisions). We refer to the ratio between the number of such ELOC and the total patch ELOC as *latent patch coverage*.

We counted these lines by keeping a sliding window of uncovered patch lines from the past ten revisions and checking whether the current revision covers them. When a patch modifies a source file, all entries from the sliding window associated with lines from that file are remapped if needed, using the line mapping algorithm discussed in Section 2.

Table 6 shows the overall latent patch coverage i.e. the fraction of patch lines that are covered in the next few revisions after the patch is introduced. We report the results for three sliding window sizes: one, five and ten revisions.

Table 7: The median coverage and the number of revisions achieving 100% coverage for the revisions containing bug fixes. The overall metrics are included for comparison.

	Coverage (med)		Fully Covered	
App	**Overall**	**Fix**	**Overall**	**Fix**
Memcached	89.0%	100%	45.4%	58.5%
Redis	0%	94.1%	25.5%	50.0%
ØMQ	76.0%	55.4%	33.3%	31.8%

The latent patch coverage is significantly smaller compared to the overall patch coverage, accounting for at most 6.4% in Redis, where, as previously pointed out, the developers almost never add code and tests in the same revision.

As conjectured, we found two main causes of latent patch coverage: tests being added only after the patch was written (this was the case in Lighttpd, where 12 revisions which only add tests cover an additional 74 ELOC) and patch code being enabled later on. In fact, the majority of latent patch coverage in Lighttpd—337 lines—is obtained by 6 revisions which change no test files. Upon manual inspection, we found that the code involved was initially unused, and only later revisions added calls to it.

Latent patch coverage is important to consider in various coverage analyses. The delay of several revisions until obtaining the patch coverage can be an artefact of the development methodology, in which case it should be assimilated into the normal patch coverage. Furthermore, our results show that in most of the systems analysed, latent patch coverage is small but non-negligible.

3.5 Bug Analysis

RQ8: Are bug fixes better covered than other types of patches?

RQ9: Is the coverage of buggy code less than average?

To answer these RQs, we collected bug data according to the methodology presented in Section 2 and we limited our analysis to the three systems which lend themselves to automatic identification of bug fixes based on commit messages: Memcached, Redis and ØMQ. The other three systems use non-specific commit messages for bug fixes, requiring an extensive manual analysis or more complex algorithms such as machine learning and natural language processing to understand the contents of a specific revision [25]. We ignored revisions which do not affect executable files, such as fixes to the build infrastructure or the documentation and then manually confirmed that the remaining revisions are indeed bug fixes [13] and further removed fixes which do not add or modify executable lines. We thus obtained 41 fixes in Memcached and 22 fixes each in Redis and ØMQ.

We measured the patch coverage of these revisions and report the median values in Table 7, together with the corresponding overall metric for comparison. For both Memcached and Redis, the coverage for fixes is higher than that for other types of patches. For Redis, the median value jumps from 0% to 94.1%, while for Memcached the difference is less pronounced. On the other hand, the fixes in ØMQ are covered less than on average. The fraction of fixes which have 100% coverage follows the same trend.

To try to understand whether buggy code is less thoroughly tested than the rest of the code, we started from the observation that bug-fixing revisions are usually only addressing the bug, without touching unrelated code. Because of this, we can identify the code responsible for the bugs by looking at the code which is removed or modified by bug-fixing revisions and compute its coverage in the revision before the fix. The overall coverage for this code is 72.7% for Memcached—roughly the same as the overall patch coverage, 65.2% for Redis—much larger than the overall patch coverage, and 35.8% for ØMQ—significantly lower.

While these numbers cannot be used to infer the correlation between the level of coverage and the occurrence of bugs—the sample is too small, and the bugs collected are biased by the way they are reported—they suggest the limitations of line coverage as a testing metric, with bugs even being introduced by patches with a high coverage. Therefore, even for well-tested code, tools which thoroughly check each program statement for bugs using techniques such as symbolic execution can be useful in practice—for instance, our tool ZESTI [22] was specifically designed to enhance existing regression tests to check for corner-case scenarios.

3.6 Threats to Validity

The main threat to validity in our study regards the generalisation of our results. The patterns we have observed in our data may not generalise to other systems, or even to other development periods for the same systems. However, we regard the selected systems to be representative for open-source C/C++ code, and the analysis period was chosen in an unbiased way, starting with the current version at the time of our experiments.

Errors in the software underlying our framework could have interfered with our experiments. Both Docker and LXC were under development and not recommended for use in production systems at the time of our study. Furthermore, in case of some applications, we have observed test failures caused by the AuFS[21] filesystem used by Docker. However, we have thoroughly investigated these failures and we believe they did not affect the results presented in our study.

Given the large quantity of data that we collected from a large number of software revisions, errors in our scripts cannot be excluded. However, we have thoroughly checked our results and scripts, and we are making our framework and data available for further validation.

4. RELATED WORK

Despite the significant role that coverage information plays in software testing, there are relatively few empirical studies on this topic. We discuss some representative studies below.

Early work on this topic was done by Elbaum et al. [8], who have analysed how the overall program coverage changes when software evolves, using a controlled experiment involving the `space` program, and seven versions of the Bash shell. One of the key findings of this study was that even small changes in the code can lead to large differences in program coverage, relative to a given test suite. This is a different finding from previous work, such as that by Rosenblum and Weyuker [29], which has found that coverage remains

[21] `http://aufs.sourceforge.net/`

stable over time for the KornShell benchmark. In this paper, we have looked at a related question, of whether overall coverage remains stable over time, taking into consideration the changes to the evolving test suite as well.

Zaidman et al. [41] have examined the co-evolution of code and tests on two open-source and one industrial Java applications. The study looks at the evolution of program coverage over time, but only computes coverage for the major and minor releases of each system, providing around ten data points for each system. By looking at the co-evolution of code and tests, the analysis can infer the development style employed by each project: one key finding is that code and tests co-evolve either *synchronously*, as when agile methods are used; or *phased*, with periods of intense coding followed by periods of intense testing. For our benchmarks, we have observed both development styles.

To the best of our knowledge, this is the first paper that specifically looks at how well patches are covered over a large number of program versions. Our prior work on testing software patches [23] reported the aggregate patch coverage achieved by regression test suites, but the focus was on evaluating the patch testing technique proposed.

There is a rich literature on predicting software bugs by mining software repositories [12, 13]; however, prior work has focused almost exclusively on static metrics, while in this work we propose using dynamic metrics such as patch coverage to aid the task.

Our ongoing effort is to develop Covrig into a flexible platform for mining static and dynamic metrics from software repositories. In terms of similar infrastructure efforts, SIR [7] is a well-known repository for software testing research, which offers a variety of programs written in several different languages, together with test suites, bug data, and scripts. SIR also provides multiple versions for the same application, but typically less than a dozen. Furthermore, SIR does not include any support for running versions in isolation. Ideally, the mechanisms provided by Covrig would be integrated with the rich data in SIR to enable more types of analyses at the intersection of software testing and evolution.

While SIR contains mostly artificially-generated faults, iBUGS [5] provides a semi-automated approach for extracting benchmarks with real bugs from project histories, using an approach based on commit messages and regression tests. iBUGS' idea of using the regression tests as a semi-automatic bug confirmation mechanism could be borrowed by Covrig whenever fixes are accompanied by tests, reducing the manual effort needed to apply it to new projects.

5. CONCLUSION

Despite the important role that regression test suites play in software testing, there are surprisingly few empirical studies that report how they co-evolve with the application code, and the coverage level that they achieve. Our empirical study on six popular open-source applications, spanning a combined period of twelve years, aims to contribute to this knowledge base. To the best of our knowledge, the number of revisions executed in the context of this study—1,500—is significantly larger than in prior work, and this is also the first study that specifically examines patch coverage.

Our experience has revealed two main types of challenges for conducting similar or larger studies that involve running a large number of program revisions. The first category relates to the inherent difficulty of running revisions:

1. Decentralised repositories have non-linear histories, so even defining what a revision is can be difficult, and should be done with respect to the research questions being answered. In our case, we chose a granularity at the level of commits and merges to the main branch.

2. Older revisions have undocumented dependencies on specific compiler versions, libraries, and tools. We found it critical to run each revision in a separate virtualised environment as provided by Covrig, to make it easy to install the right dependencies, or adjust build scripts.

3. Some revisions do not compile. This may be due to errors introduced during development and fixed later, or due to incompatible dependencies. The execution infrastructure has to be flexible in tolerating such cases, and one needs a methodology for dealing with non-compilable revisions. In our case, we have skipped over the non-compilable revisions and incorporated their changes into the next compilable one.

4. The execution of the regression test suite is often nondeterministic—the test suite may nondeterministically fail and some lines may be nondeterministically executed. Studies monitoring program execution need to take nondeterminism into account.

The second category of challenges relates to reproducibility and performance. To address these challenges, we have designed and implemented Covrig, a flexible framework that ensures reproducibility through the use of software containers technology. Performance has two different aspects: at the level of an individual revision, we have found it essential to use a form of operating system-level virtualisation (in our case, LXC and Docker), in order to minimise the time and space overhead typically associated with hardware virtualisation. Across revisions, we found it necessary to provide the ability of running our set of revisions on multiple local and cloud machines. For example, running the Git regression suite took in our case 26 machine days (250 revisions $\times$ 30 min/revision $\times$ 5 runs), which would have been too expensive if we used a single machine, especially since we also had to repeat some runs during our experimentation.

We are making Covrig available to the wider research community, together with our demonstrative empirical study and experimental data, hoping it will encourage further studies examining both static and dynamic metrics related to the evolution of real systems. The project webpage can be found at `http://srg.doc.ic.ac.uk/projects/covrig`.

6. ACKNOWLEDGEMENTS

We thank our anonymous reviewers for their constructive comments, and Oscar Soria Dustmann and Hristina Palikareva for their careful proofreading of the text. This research has been generously supported by EPSRC through a DTA studentship, an Early-Career Fellowship and the grant EP/J00636X/1, and by Google through a PhD fellowship.

7. REFERENCES

[1] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'11)*, July 2011.

[2] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Regression tests to expose change interaction errors. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Aug. 2013.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[4] G. Canfora, L. Cerulo, and M. Di Penta. Identifying changed source code lines from version repositories. In *Proc. of the 4th International Workshop on Mining Software Repositories (MSR'07)*, May 2007.

[5] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proc. of the 22nd IEEE International Conference on Automated Software Engineering (ASE'07)*, Nov. 2007.

[6] H. Dan and R. Hierons. Semantic mutation analysis of floating-point comparison. In *Proc. of the IEEE International Conference on Software Testing, Verification, and Validation (ICST'12)*, Apr. 2012.

[7] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering (EMSE)*, 10(4):405–435, Oct. 2005.

[8] 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.

[9] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 38(2):278–292, 2012.

[10] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)*, Feb. 2008.

[11] M. J. Harrold, C. Unwersity, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 2:270–285, 1993.

[12] K. Herzig, S. Just, A. Rau, and A. Zeller. Predicting defects using change genealogies. In *Proc. of the 24th International Symposium on Software Reliability Engineering (ISSRE'13)*, Nov. 2013.

[13] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

[14] P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

[15] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*, Sept. 2011.

[16] P. Joshi, K. Sen, and M. Shlimovich. Predictive testing: amplifying the effectiveness of software testing. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, Sept. 2007.

[17] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *Proc. of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, Sept. 2006.

[18] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[19] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[20] K.-K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Proc. of the 18th International Static Analysis Symposium (SAS'11)*, Sept. 2011.

[21] P. D. Marinescu and C. Cadar. High-coverage symbolic patch testing. In *Proc. of the 19th International SPIN Workshop on Model Checking of Software (SPIN'12)*, July 2012.

[22] P. D. Marinescu and C. Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)*, June 2012.

[23] P. D. Marinescu and C. Cadar. KATCH: High-coverage testing of software patches. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Aug. 2013.

[24] M. Maurer and D. Brumley. TACHYON: Tandem execution for efficient live patch testing. In *Proc. of the 21st USENIX Security Symposium (USENIX Security'12)*, Aug. 2012.

[25] A. Murgia, G. Concas, M. Marchesi, and R. Tonelli. A machine learning approach for text categorization of fixing-issue commits on CVS. In *Proc. of the 4th International Symposium on Empirical Software Engineering and Measurement (ESEM'10)*, Sept. 2010.

[26] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .NET with feedback-directed random testing. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'08)*, July 2008.

[27] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proc. of the ACM Symposium on the Foundations of Software Engineering (FSE'08)*, Nov. 2008.

[28] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'11)*, June 2011.

[29] D. Rosenblum and E. Weyuker. Using coverage information to predict the cost-effectiveness of

regression testing strategies. *IEEE Transactions on Software Engineering (TSE)*, 23(3):146–156, 1997.

[30] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering (TSE)*, 22, 1996.

[31] A. Singhal. Modern information retrieval: a brief overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 24:35–43, 2001.

[32] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. of the 2005 International Workshop on Mining Software Repositories (MSR'05)*, May 2005.

[33] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proc. of the 2nd European Conference on Computer Systems (EuroSys'07)*, Mar. 2007.

[34] K. Sparck Jones. Document retrieval systems. chapter A Statistical Interpretation of Term Specificity and Its Application in Retrieval, pages 132–142. 1988.

[35] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'02)*, July 2002.

[36] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. eXpress: guided path exploration for efficient regression test generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'11)*, July 2011.

[37] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *Proc. of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, Mar. 2009.

[38] C. Williams and J. Spacco. SZZ revisited: Verifying when changes induce fixes. In *Proc. of the 2008 International Workshop on Defects in Large Software Systems (DEFECTS'08)*, July 2008.

[39] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proc. of the ACM Symposium on the Foundations of Software Engineering (FSE'10)*, Nov. 2010.

[40] Z. Xu and G. Rothermel. Directed test suite augmentation. In *Proc. of the 16th Asia-Pacific Software Engineering Conference (ASPEC'09)*, Dec. 2009.

[41] 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)*, 16(3):325–364, 2011.