Mind the Gap: What Working With Developers on Fuzz Tests Taught Us About Coverage Gaps

Carolin Brandt
c.e.brandt@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Marco Castelluccio
mcastelluccio@mozilla.com
Mozilla Corporation
London, UK

Christian Holler
choller@mozilla.com
Mozilla Corporation
Bonn, Germany

Jason Kratzer
jkratzer@mozilla.com
Mozilla Corporation
Asheville, NC USA

Andy Zaidman
a.e.zaidman@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Alberto Bacchelli
bacchelli@ifi.uzh.ch
University of Zurich
Zurich, Switzerland

ABSTRACT

Can fuzzers generate partial tests that developers find useful enough to complete into functional tests (e.g., by adding assertions)? To address this question, we develop a prototype within the Mozilla ecosystem and open 13 bug reports proposing partial generated tests for currently uncovered code. We found that the majority of the reactions focus on whether the targeted coverage gap is actually worth testing. To investigate further which coverage gaps developers find relevant to close, we design an automated filter to exclude irrelevant coverage gaps before generating tests. From conversations with 13 developers about whether the remaining coverage gaps are worth closing when a partially generated test is available, we learn that the filtering indeed removes clearly non-test-worthy gaps. The developers propose a variety of additional strategies to address the coverage gaps and how to make fuzz tests and reports more useful for developers.

ACM Reference Format:
Carolin Brandt, Marco Castelluccio, Christian Holler, Jason Kratzer, Andy Zaidman, and Alberto Bacchelli. 2024. Mind the Gap: What Working With Developers on Fuzz Tests Taught Us About Coverage Gaps . In *46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3639477.3639721

1 INTRODUCTION

While the importance of automated tests is widely accepted [3, 7], creating them is a tedious task for developers [4–6]. Automatic test generation aims to alleviate the developer's effort when writing tests. State-of-the-art tools can reach high structural coverage [9, 16, 19, 28, 29], but face obstacles like understandability of the tests [10, 17, 25], and integration of the tools into the companies tooling [1, 11]. In contrast, automated testing tools from the security community, namely fuzzers, are successfully applied in practice[1] [2, 20]. Fuzzers explore possible inputs to a program to find crashes and potential security vulnerabilities [8, 32]. At Mozilla, fuzzers find around 25% of all critical or high rated security vulnerabilities, year after year (see Figure 2). In this experience report, we document our exploration on whether fuzzers can generate partial tests that developers find useful to complete into functional tests.

One of the sources of effort when writing tests is to create the fixture—the setup and operations needed to reach the targeted code to be tested [12]. This is where fuzzers can help: When they find a crash, they return the fixture and inputs triggering the crash. To turn the fixture into a complete functional test, a developer would then add an assertion that checks the behavior of the code under test. Let us illustrate this with an example:

> To improve their test suite, Ezra's software company introduced a tool that proposes partial fuzzing-generated tests to developers. From the tool, Ezra receives an issue report including a fuzzing-generated fixture that reaches a line of code that is not yet covered by their test suite. She inspects the targeted code and the provided fixture to judge whether the fixture is useful enough for her to complete it into a functional test. To complete it, Ezra adds an assertion that checks the behavior of the targeted code, surrounds it with their test framework's template, and then includes it in their test suite.

To explore the potential of such a tool, we conduct a study with two main phases. First, we build a prototype based on Mozilla's fuzzing infrastructure. Using this, we submit issue reports with partial tests that draw from the output of fuzzers. We analyze the responses to the reports to answer our first research question:

> **RQ1:** What are developers' reactions when proposing fuzzing-based tests to be completed into functional tests?

Our goal is to determine whether developers see enough value in these tests to develop them into functional tests, such as by adding assertions. To enhance the significance of these partial tests, we tailor them to target code sections not covered by current tests. Based on developer feedback, we observe that they first assess the

ICSE-SEIP '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0501-4/24/04.
https://doi.org/10.1145/3639477.3639721

[1] https://hacks.mozilla.org/2021/02/browser-fuzzing-at-mozilla/

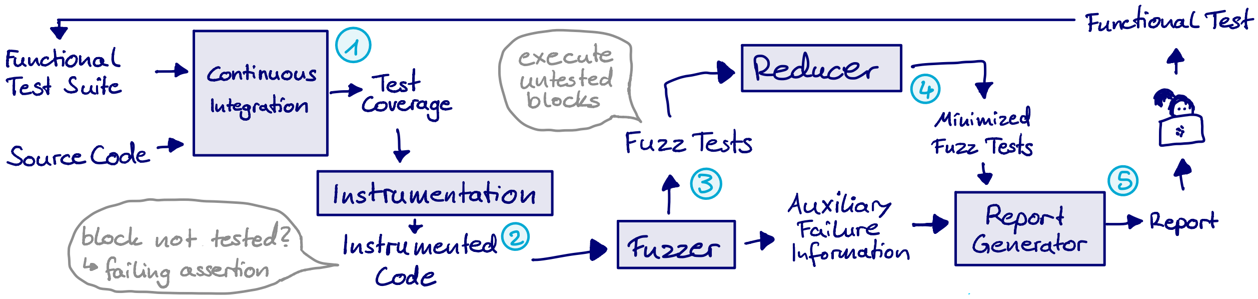


Figure 1: Overview of our fuzzing inspiration approach.

significance or 'relevance' of the specific code under test before evaluating the value of the partial tests themselves.

Drawing from these insights, in the study's second phase, we design a filter to pinpoint coverage gaps more relevant to the developers. We then engage with developers to gauge their interest in addressing these gaps, addressing our second research question:

> **RQ2:** What are developers' opinions about closing the coverage gaps remaining after our filter?

Our study reveals developers' criteria for determining the significance of closing a coverage gap and their preferred methods, including completing our partial fuzzing-based tests.

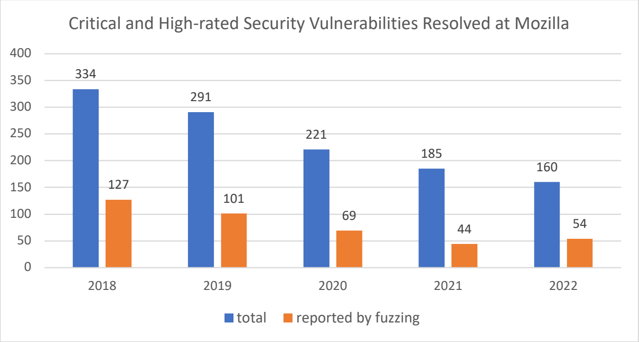


Figure 2: Number of resolved critical and high-rated security vulnerabilities over the years at Mozilla. See Appendix A for how we obtained these numbers.

2 FUZZING TO INSPIRE FUNCTIONAL TESTS

In this section, we explain our approach to generate partial, fuzzing-based tests and to create reports for developers to complete them to functional tests. We start by describing our general design, and then explain how we concretely implement it for Mozilla.

2.1 Inspiration Through Fuzzing-based Tests

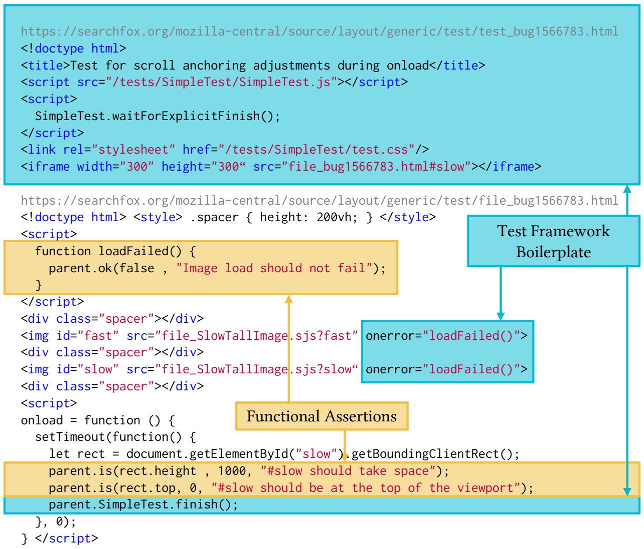
The test generation consists of five steps, which we illustrate in Figure 1.

First, we obtain the line coverage of the current regression test suite, e.g., from the continuous integration artifacts ①. We then instrument all code blocks which are not executed by the test suite, i.e., they are not yet covered by the tests ②. We instrument these blocks with an assertion that will trigger the fuzzer: When a fuzz test executes one of our inserted assertions, the fuzzer will register a crash and save the fuzz test. In the third step ③, we use a generative fuzzer to collect tests that execute the instrumented, not-yet-tested code blocks. During the fuzzing, we also collect auxiliary information such as which instrumented code block is executed and the stack trace when the fuzzer hit the assertion. Next ④, we minimize the fuzz tests to only those lines necessary to trigger the instrumented assertion. Sometimes, there are multiple minimal tests. In this case, we keep all to provide alternative tests to the developer.

After obtaining the minimal fuzz tests, we use them and the auxiliary information to create a bug report for developers ⑤. We explain that we have a partial test and link to the untested code block it executes. We provide the test with the smallest character count, and attach the stack trace, to help the developer understand how the fuzz test executes the code block. We also attach the other minimized tests as alternative inspirations for the developer. The bug report asks the developer to complete the test by adding a functional check, in xUnit terms: an assertion. If they think it is worth to do so, the developer should add the test to the regression test suite. For this, they also need to write the boilerplate code necessary to integrate the test into the test suite. Figure 3 shows an example test from the Firefox source code, highlighting the functional assertions and the boilerplate code that embeds the test into the test suite. The fuzzing-based tests don't contain the boilerplate code yet, because the fuzzer uses a different harness to execute the tests, and there are multiple possible test suites with their own frameworks that the developer might choose to add the test to.

2.2 Instantiation in the Mozilla Ecosystem

We implemented our approach for the development environment of the Mozilla Firefox browser. Several regression test suites are based on `.html` files that are executed in a sandboxed browser environment [18]. Figure 3 shows an example test that checks that the browser automatically scrolls to the right anchor on a page. The functional check happens in the `is(...)` call, where two values are compared and if they are not the same, the test fails with the provided error message. To generate partial tests matching these browser tests, we use the Domato fuzzer.[2] Domato is a state-of-the-art generative DOM fuzzer, that uses grammars to generate random HTML, JavaScript, and CSS code in one `.html` file [31, 34]. We use Mozilla's `grizzly`[3] harness to run Domato's generated tests in Firefox and record crashes and auxiliary information. We employ delta-debugging [33] implemented in the tool `lithium`[4] to reduce the fuzz tests to the minimum lines needed to trigger the crash—to execute the instrumented, not-yet-tested code block. For the exact configurations, please refer to our replication package[5] Figure 4 shows one example of a reduced fuzz test generated during our case study. The reports we generate are for Mozilla's issue tracker Bugzilla, and use their code search engine Searchfox to link to the targeted code block. Figure 5 shows an example of a report we generate.



```
https://searchfox.org/mozilla-central/source/layout/generic/test/test_bug1566783.html
<!doctype html>
<title>Test for scroll anchoring adjustments during onload</title>
<script src="/tests/SimpleTest/SimpleTest.js"></script>
<script>
  SimpleTest.waitForExplicitFinish();
</script>
<link rel="stylesheet" href="/tests/SimpleTest/test.css"/>
<iframe width="300" height="300“ src="file_bug1566783.html#slow"></iframe>
```

```
https://searchfox.org/mozilla-central/source/layout/generic/test/file_bug1566783.html
<!doctype html> <style> .spacer { height: 200vh; } </style>
<script>
  function loadFailed() {
    parent.ok(false , "Image load should not fail");
  }
</script>
<div class="spacer"></div>
<img id="fast" src="file_SlowTallImage.sjs?fast" onerror="loadFailed()">
<div class="spacer"></div>
<img id="slow" src="file_SlowTallImage.sjs?slow“ onerror="loadFailed()">
<div class="spacer"></div>
<script>
onload = function () {
  setTimeout(function() {
    let rect = document.getElementById("slow").getBoundingClientRect();
    parent.is(rect.height , 1000, "#slow should take space");
    parent.is(rect.top, 0, "#slow should be at the top of the viewport");
    parent.SimpleTest.finish();
  }, 0);
} </script>
```

Figure 3: A test from the Firefox regression test suite.

```
<script>
window.requestIdleCallback(window.close, {timeout: 10000})
</script>
<style>
html:last-of-type, #htmlvar00001 {
  text-align-last: start; }
.class0, aside:nth-last-child(2) {
  column-width: 1em;
</style>
<table>
<colgroup width="3" span="20">+GEE&gt;uo/c(wt6,N:1=*</colgroup>
<caption class="class0">
```

Figure 4: A partial fuzzing-based test produced in our study. To complete it, a functional assertion and test framework boilerplate code needs to be added.

3 PROPOSING INSPIRATIONAL FUZZING-BASED TESTS TO DEVELOPERS

To investigate the feasibility of our approach for proposing partial, fuzzing-based tests for completion to developers, we conduct a prototype study [21] at Mozilla. Our goal is to explore whether our approach can provide helpful tests to software developers, and what aspects require further attention to create tests and reports that the developers find useful. In the study, we generate tests for uncovered code in the Firefox code base and submit 13 Bugzilla reports proposing them to developers. We analyze the ensuing discussions on the reports to identify why the developers choose to act on the report or not, and how they resolve them. We conducted a risk analysis and sought approval from our local ethics review boards with respect to data protection.

3.1 Study Design and Execution

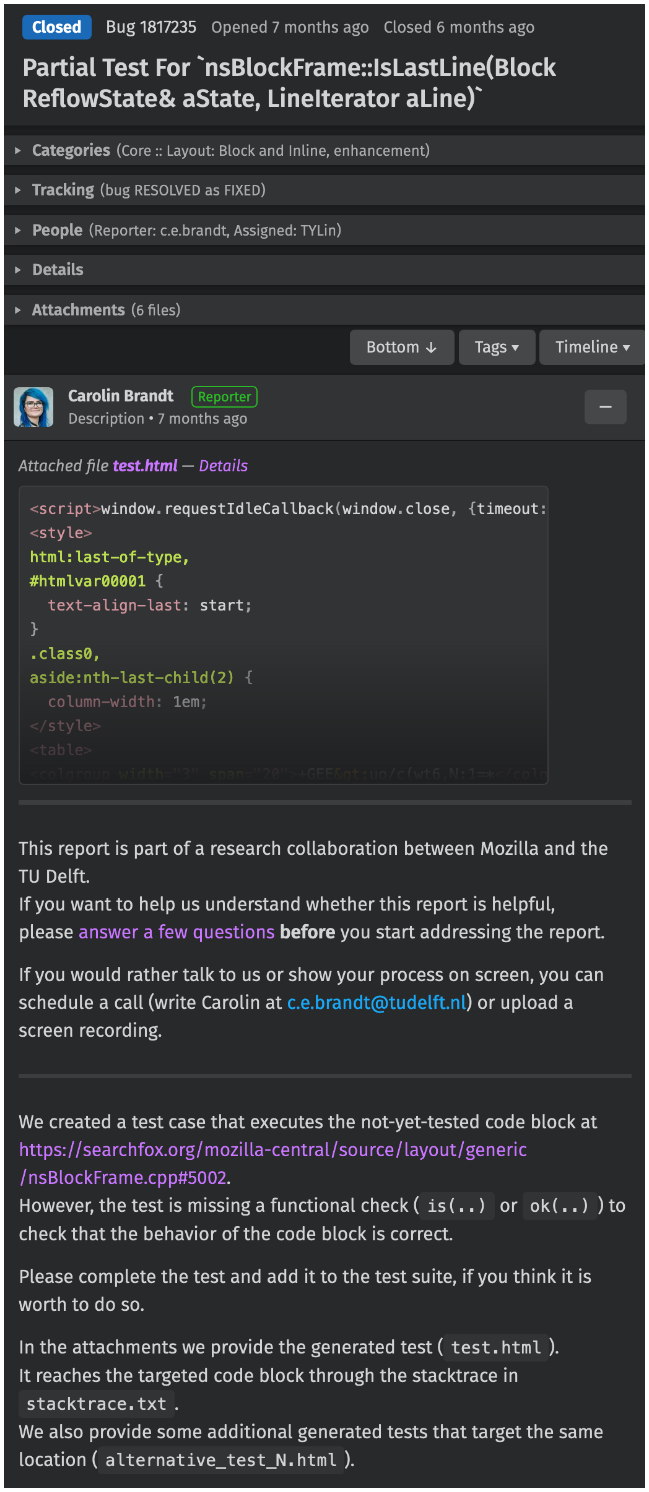
For our study, we choose two folders in the Firefox code base to instrument and generate tests for. The folder `dom/` contains the code pertaining to the implementation of the Document Object Model[6] (DOM) and its APIs. The folder `layout/` contains the layout engine, responsible for laying out the elements of the page in the correct positions.[7] In initial trials, we saw that when inspecting the fuzzing-based tests generated for these folders, we could draw clear connections between the objects and attributes in the test generated by the DOM fuzzer and the code targeted by the tests. For this pragmatic reason, we opted to use these two folders in our study.

After instrumenting the code[8], we let our fuzzer run on a desktop machine for 30 minutes. This yielded us 133 fuzz tests and corresponding reports. Of these, 36 were duplicates targeting the same coverage gap. To not overwhelm the particular contributors or groups, we decided to only open one Bugzilla report per Firefox component. Components are categories of functionality that Mozilla uses to manage responsible reviewers and triagers within Bugzilla. Each code file belongs to one component, and we identified the corresponding component by looking at the file which contains the coverage gap targeted by a generated test. We submitted 13 Bugzilla reports, one for each of the components present in our set of generated tests. In Figure 5 you can see an example of such a report. We provide the shortest test generated by the fuzzer for the targeted coverage gap and the stacktrace showing how the test reaches the coverage gap. In addition, we included alternative tests that the fuzzer produced for the same coverage gap, to provide more options in case the shortest test was not useful.

Over the following week, we responded to any comments and questions by the developers. Because we categorized our reports as "enhancement"s, many were initially not picked up through the triage processes focusing on reports labeled as "defects"s. After identifying relevant developers based on the authors or reviewers of the patches that created the code under test, we pinged them personally and then received reactions to four more of the reports (1817159, 1817173, 1817235, 1817219).

[2] https://github.com/googleprojectzero/domato
[3] https://github.com/MozillaSecurity/grizzly
[4] https://github.com/MozillaSecurity/lithium
[5] You can browse our replication package at: https://anonymous.4open.science/r/moz-fuzz-inspiration-replication/readme.md or download it at https://zenodo.org/doi/10.5281/zenodo.10470823.

[6] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction
[7] https://wiki.mozilla.org/Platform/Layout#About
[8] Revision: 63a3d733b2331033f48d10995ce09abf50def953 in `mozilla-unified`


Closed
Bug 1817235 Opened 7 months ago Closed 6 months ago

Partial Test For `nsBlockFrame::IsLastLine(Block ReflowState& aState, LineIterator aLine)`

Categories (Core :: Layout: Block and Inline, enhancement)
Tracking (bug RESOLVED as FIXED)
People (Reporter: c.e.brandt, Assigned: TYLin)
Details
Attachments (6 files)
Bottom ↓
Tags ▾
Timeline ▾
Carolin Brandt
Reporter
Description • 7 months ago
−

Attached file ***test.html*** *— Details*

```
<script>window.requestIdleCallback(window.close, {timeout:
<style>
html:last-of-type,
#htmlvar00001 {
  text-align-last: start;
}
.class0,
aside:nth-last-child(2) {
  column-width: 1em;
</style>
<table>
<colgroup width="3" span="20">+GEE&gt;uo/c(wt6,N:1=*</colg
```

This report is part of a research collaboration between Mozilla and the TU Delft.
If you want to help us understand whether this report is helpful, please answer a few questions **before** you start addressing the report.

If you would rather talk to us or show your process on screen, you can schedule a call (write Carolin at c.e.brandt@tudelft.nl) or upload a screen recording.

We created a test case that executes the not-yet-tested code block at https://searchfox.org/mozilla-central/source/layout/generic/nsBlockFrame.cpp#5002.
However, the test is missing a functional check (`is(..)` or `ok(..)`) to check that the behavior of the code block is correct.

Please complete the test and add it to the test suite, if you think it is worth to do so.

In the attachments we provide the generated test (`test.html`).
It reaches the targeted code block through the stacktrace in `stacktrace.txt` .
We also provide some additional generated tests that target the same location (`alternative_test_N.html`).

Figure 5: A Bugzilla report submitted during our study.

Bug ID	Component	Resolution
1816640	SVG	Open
1816862	DOM: Serializers	No Reaction
1817150	CSS Parsing and Computation	No Reaction
1817154	DOM: Core & HTML	No Reaction
1817158	Layout	No Reaction
1817159	XSLT	Open
1817215	MathML	Open
1817221	Audio/Video	No Reaction
1817238	Layout: Tables	No Reaction
1817173	Web Painting	Open
1817176	DOM: Networking	Resolved
1817235	Layout: Block and Inline	Resolved
1817219	Layout: Images, Video, and HTML Frames	Resolved

Table 1: List of all Bugzilla reports we submitted, including the targeted component and the resolution state of the report. The reports can be accessed via “https://bugzilla.mozilla.org/show_bug.cgi?id=<id>” and are hyperlinked in the ID column.

3.2 Developer Reactions

In this section, we give an overview of the reactions to the reports we submitted. Table 1 lists each report, the component it is related to, and a link to the full discussion. Out of the 13 reports, six received no reaction, three were resolved, and four received comments but remain open.

From the reports that received reactions, most prominently the developers focused on **whether the targeted code is worth testing**. The developer reacting to report 1816640 stated that: *“All* [method under test] *does is forward to setAttribute(’type’, value). I’m not sure that there’s much value in testing it as it’s only one line of code.”* Two developers reacted to report 1817159. One pointed out that the code for this feature is *“rather derelict”*, but then pointed to a recent zero-day bug related to this code, stating that *“maybe it would be good to invest a bit of time ensuring that we have good testing”*. A second developer later explained that, together with a colleague, they determined that the targeted code is suitable to write tests. However, they also point out that these tests *“wouldn’t teach or touch deeper xslt logic”*, which has the team’s priority at that moment. We interpreted “teach” in the vein of tests serving as documentation on how to use the code under test [3, 14, 22]. Even though tests for the targeted code would *“improve exception catching in tests without doubts”*, they decide to leave this in their backlog. The developer reacting to report 1817219 explained that each of the tests are expected to trigger the early return in the targeted coverage gap. They stated that it would not be worth to test that code, also because they recently changed the surrounding behavior.

When the reports lead to action from the developers beyond comments, we observed a variety of **ways to address the reports**. Report 1817176 received a quick reaction with a patch submitted to code review. The patch did contain a test for the targeted coverage gap, inspired by the test we submitted, but in a different format than our proposed one: instead of a '.html' file, it was an addition to a '.json' file that configures parameterized tests. The first developer reacting to report 1817235 stated that it "looks potentially interesting," explaining the behavior triggered by the test. They propose to clean up the provided test cases, making sure that the behavior of the code actually makes sense. A different developer picked up the task and submitted a patch for the targeted line of code. However, when the developer worked on the report, our tests did not trigger a printf statement they added in the while loop they targeted to test. The developer speculated that this could have been caused by changes to the code under test between the moment of fuzzing, opening the bug report and addressing the bug report. So they crafted a test case themselves, inspired by hints from our provided tests on which attributes are involved in triggering the code under test. Report 1817219 was resolved by submitting and merging a patch that removed the whole method targeted by our fuzz test. One developer linked our report 1817215 to another active report about updating tests for this component after changes in the cross-browser web-platform-tests.

We received a very detailed reaction to report 1817173, analyzing the behavior triggered by the generated tests. The targeted coverage gap was a fallback path that is no longer used because a newer component has taken over its responsibility on the operating systems used for the CI coverage data. The developer pointed to the line of test code that the newer component could not handle and which triggered the fallback path. They initially propose two solutions to resolve the bug: (1) adding a test suite variant that runs all existing tests while enforcing the fallback path, or (2) duplicating some related tests and modifying them to use the fallback path. In the ensuing discussion, we uncovered that the former option already exists, but this **test suite covering the targeted code is not executed in the current CI** coverage calculation.

Answer to RQ1: What are developers' reactions when proposing fuzzing-based tests to be completed into functional tests?: The developers reflected on whether the code targeted by the tests is worth covering, mentioning a too small coverage gap or early returns as reasons to not act upon the reports. Another reason was that the code was covered by tests not executed on the CI. Other reports were addressed in a variety of ways: submitting a syntactially different test for the same scenario, using our test as a starting point to write their own test, or removing the targeted code as it was no longer used.

4 SELECTING RELEVANT COVERAGE GAPS

In the developer's reactions to our Bugzilla reports, we observed that several of the coverage gaps we targeted with the fuzzing-based tests were considered less relevant to test by the developers (Bug 1816640, Bug 1817159, Bug 1817219). This lead to the developers not further acting upon our generated tests. To provide more relevant reports and partial tests, we decided to take a deeper look at which coverage gaps we should target with our tests.

For this, we designed an automatic filtering step that excludes less interesting coverage gaps (between ① and ② in Figure 1) before instrumenting the code in preparation for the fuzzer. The pseudo-code for the automatic filter is shown in Figure 6. The filter excludes:

- coverage gaps that are only a single line of code long, as the comments on the bug reports deemed such gaps as too small to close (Bug 1816640), and
- coverage on conditions and branches that represent an early return out of a function (Bug 1817159, Bug 1817219).

What we call an early return, is when a function returns a default value after checking for an error or warning. We detect such early returns by keywords, such as `NS_ERROR` or `Throw`, combined with a `return` in the coverage gap. We manually inspected further coverage gaps that were hit early, i.e., in two 2 minute runs, by our fuzzing approach, and extended the keyword list to detect coverage gaps that were deemed as early returns by our Mozilla collaborators. When running our filter over the coverage gaps in the folders we instrumented before (`/dom` and `/layout`), we exclude 15 054 (single line) and 8 085 (early return) coverage gaps, leading to a remainder of 8 644 coverage gaps to instrument.

```
def should_we_instrument_this_line():
  if (len(coverage_gap.lines) == 1)
  or (len(coverage_gap.lines) == 2
    and second_line_contains_only_closing_brace)
  or is_early_return():
    return False # don't instrument

def is_early_return():
  is_error_or_warning, is_return = False
  if "NS_WARN_IF" in condition_before_coverage_gap:
    is_error_or_warning = True
  for line in coverage_gap.lines:
    if ("NS_WARNING" in line
        or "NS_ERROR" in line
        or "Throw" in line
        or "WEBM_DEBUG" in line
        or "Error" in line
        or ("promise" in line and "reject" in line))):
      is_error_or_warning = True
    if "return" in line:
      is_return = True
  return is_error_or_warning and is_return
```

Figure 6: Our filter for interesting coverage gaps.

5 DO DEVELOPERS THINK THESE COVERAGE GAPS SHOULD BE TESTED?

To evaluate whether our filtering for coverage gaps indeed yields more interesting test targets, we again reach out to the developers for feedback. We retrieve a new, more recent revision and CI coverage run of the Firefox code base[9] and apply our filter to select relevant coverage gaps. To extend our reach of potential developers to talk to, we extended the folders of the source code we implemented to include `/accessible`, `/editor`, and `/gfx` in addition to `/dom` and `/layout`. Applying the filter left us with 19 050 coverage gaps to instrument, after excluding 30 672 (single line) and 13 626 (early return) coverage gaps. We instrumented the remaining coverage gaps and again generate fuzz tests for 30 minutes on a desktop computer. This yielded us a set of 44 coverage gaps that pass our relevance filter and could be hit by the fuzzer within the short time

[9] Revision: 0bcf2642f5a6e7175812623451eda2ab6cb35a0d in `mozilla-unified`

budget of 30 minutes. We manually validated if each coverage gap is still present and whether it should have been filtered: We exclude five false positives (three early return statements, two single statement coverages formatted to span two lines) and one coverage gap inside code that is only executed during fuzzing runs.

We identified developers that can likely judge the test-worthiness of the coverage gaps by looking at the authors and reviewers of the patch that introduced the targeted line of code or recent patches the surrounding lines. Patches that did not show experience with the code at hand, e.g., large scale refactorings, and contributors no longer with Mozilla, were excluded by us. The second author, who is the manager of the CI and Quality Tools team, which owns the coverage infrastructure and other development tools, contacted 13 developers. We briefly explained our project and that we are now trying to identify areas of code that are interesting to cover with a test. We then gave them one or more code locations and asked whether they can explain why or why not they are valuable to test. Additionally, we sought their consent for using their comments in this paper.

Two authors analyzed the chat conversations by independently applying open and axial coding. Then, they compared and merged the emerging themes from their independent analyses. In the following, we will describe our observations along three major categories: First, we take a look at the developers' rationales for why a coverage gap is worth testing or not, motivating the need for a more refined way of looking at code coverage and the need to close coverage gaps. Then, we present varied proposals from the developers on how to address the coverage gaps in other ways than completing the fuzz test to a functional one. Finally, we consider our proposed approach of submitting bug reports with fuzz tests to be completed to functional tests and discuss feedback from the developers on how to modify the tests, report, and workflow to better fit their needs. Table 2 lists the coverage gaps we discussed with each of the 13 developers, identified in the following by D1-D13. For example, with D4 one of the 4 coverage gaps we discussed were lines 90–105 of `dom/svg/SVGMotionSMILAnimationFunction.cpp`.

5.1 Test Relevance of the Filtered Coverage Gaps

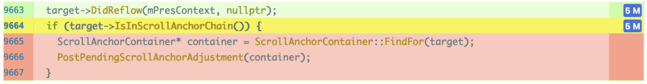
Our deeper look at filtering for interesting coverage gaps was motivated by the feedback on our initial Bugzilla reports pointing out coverage gaps too small to be worth closing (1816640), and fallback options that return early from a method in case of an error (1817173). Looking at the conversations, the filtering seems to be effective as we did not receive answers along the lines of "this is too simple code to be worthy a test." D1 reflects on a very particular reason why the code is not tested at the moment. They explain that *"our tests only test the successful part"*, which is a potential sign of confirmation bias [13]. Furthermore, D1 states that the specific failure handled by the targeted code is caused by operating systems and hardware not available on the current CI servers.

In the conversations, the developers gave **reasons why some coverage gaps should be closed**: to catch regressions (D11), increase the confidence during rewrites and larger-scale refactorings (D3, D8), documenting edge case bugs (D3), testing important edge cases (D10, Figure 7), or ensuring that the behavior matches an external specification (D2, D4). D11 motivates that the code in Figure 8 should be tested because it uses raw pointers, which are error-prone and may lead to null pointer or use-after-free errors.

D1	dom/media/platforms/PDMFactory.cpp#678-680 dom/media/platforms/PDMFactory.cpp#725-727
D2	dom/svg/SVGFEImageElement.cpp#191-196
D3	dom/events/ContentEventHandler.cpp#1752-1755
D4	dom/svg/SVGMotionSMILAnimationFunction.cpp#90-105 layout/painting/nsCSSRendering.cpp#4112-4113 dom/svg/DOMSVGLength.cpp#297-298 layout/base/PresShell.cpp#9665-9667
D5	layout/painting/nsCSSRendering.cpp#2418-2425 layout/generic/nsBlockFrame.cpp#1133-1142 dom/html/HTMLSharedElement.cpp#96-102 layout/style/GeckoBindings.cpp#1397-1401 dom/svg/SVGStyleElement.cpp#164-167 layout/base/PresShell.cpp#9665-9667
D6	layout/painting/nsCSSRendering.cpp#4112-4113 layout/svg/SVGTextFrame.cpp#3922-3926 gfx/thebes/gfxFont.cpp#1530-1533 layout/svg/SVGTextFrame.cpp#2881-2883
D7	editor/libeditor/EditorBase.cpp#2888-2890
D8	dom/svg/SVGFETileElement.cpp#51-55 layout/painting/nsCSSRenderingBorders.cpp#2845-2847 layout/painting/nsCSSRenderingBorders.cpp#2745-2748
D9	dom/xslt/xslt/txMozillaXSLTProcessor.cpp#881-890 dom/xslt/base/FtxDouble.cpp#52-53 dom/base/DirectionalityUtils.cpp#613-619 dom/base/DirectionalityUtils.cpp#1069-1070 dom/base/DirectionalityUtils.cpp#339-341
D10	layout/base/PresShell.cpp#9665-9667
D11	dom/base/DirectionalityUtils.cpp#613-619
D12	layout/tables/nsTableFrame.cpp#3018-3034
D13	dom/svg/DOMSVGLength.cpp#297-298 dom/svg/SVGElement.cpp#704-706 dom/svg/SVGPathData.cpp#476-481 dom/svg/DOMSVGAngle.cpp#105-107 dom/svg/DOMSVGAngle.cpp#28-30

Table 2: Overview of the developer chats and the discussed coverage gaps. The coverage gaps can be viewed via: "https://searchfox.org/mozilla-central/rev/8329a650e3b4f866176ae54016702eb35fb8b0d6/ <text in 2nd column>" (also hyperlinked in that column). The given line numbers are the first and last uncovered line of the coverage gap.

On the other hand, the developers gave a variety of **reasons for why the code in the coverage gap is not worth the effort of testing**. One reason is that **they think it is unlikely that there is a bug in the code**, because the function did not change



```
9663   target->DidReflow(mPresContext, nullptr);                              5 M
9664   if (target->IsInScrollAnchorChain()) {                                 5 M
9665     ScrollAnchorContainer* container = ScrollAnchorContainer::FindFor(target);
9666     PostPendingScrollAnchorAdjustment(container);
9667   }
```

Figure 7: A coverage gap that that should be closed to test important edge cases according to D10. The gap is at layout/base/PresShell.cpp#9665-9667.

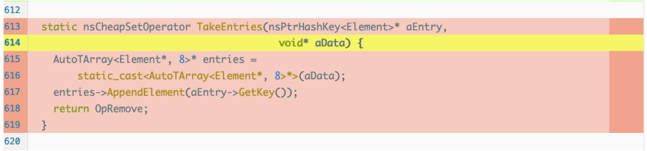
in the last 10 years (D3), no bug reports have been opened in that area for a long time (D3, D8), or it is legacy code that should not receive any changes in the future, as it serves as a fallback to a newer implementation (D8). For the coverage gap discussed with D7, *“it’s hard to find how to run the path”* because they are currently rewriting the component to use the functionality less and less, and planning to eventually remove the code all together. Other coverage gaps are described as **unlikely to be reached** because it is a do-nothing fallback for an error in a third party library (D4, D6, talking about the coverage gap shown in Figure 9), or the developers expect the case to rarely happen in practice (D4, D8).

Similar to Bugzilla report 1817173, we again encountered cases where, according to the developer, the **code should actually be covered by tests** (D5 about three coverage gaps, D8 about two coverage gaps). In the case of D8, the separate job running the relevant tests is not executed during the CI runs that calculate the coverage.

5.2 Different Ways to Address Coverage Gaps

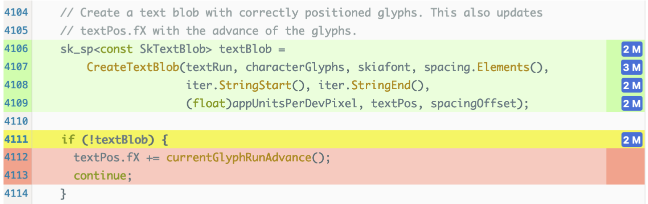
Throughout the conversations, the developers we chatted with brought up ways to address the coverage gaps that differ from completing the partial fuzz tests we could generate. An overarching concern was whether it would be **easier to manually write a test from scratch** (D4, D6). D13 points out that *“it’s not hard for a developer that knows SVG to come up with tests that hit those lines”*. D8 directly starts describing a fitting test scenario for one of the coverage gaps we asked about, and D6 explains *“I think this would be simplest to write manually, having identified the relevant code path”*. D9 stresses that *“depending on how good/bad the generated [tests] are it might be easier to just have someone write them in the first place.”*

One of the coverage gaps discussed with D6 (see Figure 9) was described by them as *“would only be used in case of some kind of failure within [a third-party] library”*, and in that case likely another failure appeared earlier, making the code under test very unlikely to be reached. To validate that this is indeed dead code, they propose to add an assertion that triggers a crash in the regular fuzzing runs, and possibly later an “unreachable” assertion to the production code base to alert the team in case the code does become reachable through future changes. D4 recounted that for one of the coverage gaps the team considered adding a crashtest, but that the value of this would be minimal as the code is already hit by the regular fuzzing runs. These examples indicate that the developers consider the **regular fuzzing runs as an alternative to address missing coverage**, increasing the confidence that these code paths do not lead to crashes or that they are unreachable.



```
612
613  static nsCheapSetOperator TakeEntries(nsPtrHashKey<Element>* aEntry,
614                                        void* aData) {
615    AutoTArray<Element*, 8>* entries =
616        static_cast<AutoTArray<Element*, 8>*>(aData);
617    entries->AppendElement(aEntry->GetKey());
618    return OpRemove;
619  }
620
```

Figure 8: A coverage gap that that should be closed according to D11, because the code uses raw pointers. The gap is at dom/base/DirectionalityUtils.cpp#613-619.



```
4104      // Create a text blob with correctly positioned glyphs. This also updates
4105      // textPos.fX with the advance of the glyphs.
4106      sk_sp<const SkTextBlob> textBlob =                                  2 M
4107          CreateTextBlob(textRun, characterGlyphs, skiafont, spacing.Elements(),   3 M
4108                         iter.StringStart(), iter.StringEnd(),             2 M
4109                         (float)appUnitsPerDevPixel, textPos, spacingOffset);   2 M
4110
4111      if (!textBlob) {                                                    2 M
4112        textPos.fX += currentGlyphRunAdvance();
4113        continue;
4114      }
```

Figure 9: A coverage gap that is a do-nothing fall-back for a third party library and therefore unlikely to be reached according to D4 and D6. The gap is at layout/painting/nsCSSRendering.cpp#4112-4113.

Based on our conversation with D5, three follow-up bug reports were filed. One, the developer filed immediately, discussing an inconsistency between different browser implementations that they discovered by looking at the coverage gap we pointed them to. The report was resolved by adding a cross-browser test for the inconsistency and removing the code causing the inconsistency, including the coverage gap. D5 also asked us to file bugs to remove the code from two of the coverage gaps as the code had become obsolete with a previous change. Together with the reaction to the Bugzilla report 1817219 in our first study, we can see that pointing to coverage gaps can also nudge developers to **delete code that became obsolete**.

5.3 Needs of Developers and How To Improve Our Approach

In our opening messages to the developers, we pointed to our project of generating tests for the coverage gaps we asked about. Because of this, several of the developers also reflected on the usefulness of such tests and the process of proposing them. D11 was open to try out the generated tests, but stressed that the test should **conform to the common test frameworks** in the project. They also proposed to directly add the test as a patch to the code review platform where the developers can edit the test assertions. D3 stated that it would be more useful to **receive the test at the time of writing the patch** that introduces the targeted line of code, as compared to receiving the tests weeks later.

Several developers saw some value in providing a generated test alongside pointing to the coverage gap. D6 explained that it is useful to know that a piece of uncovered code could be covered, the **generated test can prove that the code is reachable**. They also describe that the information about the coverage gaps can surface which *“combinations of features are going untested at present.”* With D12 we discuss a coverage gap that they described as a condition *“not going through the normal [...] process”*. We provided D12 with our generated fuzz test and they opened an issue with it to *"use the test case as a start point to investigate if the [covered] branch makes sense or not."*[10] The generated tests can also be a starting point and inspiration for a developer familiar with the code to write a complete correctness test case (D4). Three developers (D6, D11, D13) pointed out the **knowledge required to complete the tests** and determine *"what the correct behavior[...] of the test case should be"* (D6). This would require familiarity with the domain of the code (D13) or reading specifications to ensure they are followed (D11).

The **effort required by the developers to complete the tests was seen as problematic** by several of our conversation partners. D11 worries that tests that one needs to spend time to complete will be ignored because *"We're already busy enough with intermittently failing tests and what not."*. For the coverage gap discussed with D7 (see Figure 10), they state that it is fine to add a complete generated test, but *"that it's not worthwhile to use the developers' time [to write a test] for the edge case."*

Answer to RQ2: What are developers' opinions about closing the coverage gaps remaining after our filter?:
While our filter successfully excluded coverage gaps clearly not worth covering, several remaining gaps were considered not worth the effort to cover because the developers found it unlikely that there is a bug in the code, the code is unlikely to be reached, or should already be covered by other tests.
Several developers pointed out that it might be easier to manually write a test from scratch than to understand the generated test, and regular fuzzing runs covering code was seen as an alternative to address missing test coverage.
The generated tests can serve as a proof that the targeted code is reachable by a test, but the developers caution about the knowledge and effort required to complete the partial test. To improve our approach, they propose to provide tests that already conform to their common test framework and provide them at the time of submitting the patch with the code under test.

6 DISCUSSION

The feedback from the developers indicate that filtering out single-line and early return coverage gaps helps to eliminate "clearly too simple to be worth testing" coverage gaps. Nevertheless, we noted several more reasons that make a coverage gap less relevant for testing. A crucial aspect is the effort required by the developers. Even for coverage gaps described as relevant to test, developers stated that they do not have the time to write a test or complete a generated one, compared to the other tasks they have to complete.

Our initial idea was to relieve parts of the developer's efforts by generating partial tests that reach coverage gaps in their code base. However, we learned to be careful about the additional effort that we put on developer's shoulders when they have to understand a generated test before completing it. In the following, we discuss the implications of our observations for software engineering practitioners, tool builders that want to support them, and researchers in our field.

6.1 Implications for practitioners

In practice, code coverage can be used as a metric by management to judge the quality of testing performed in their teams [23, 27]. The observations in both our studies indicate that **not all "missing" coverage is equally worth testing**. This points to the need for a more refined metric that takes into account the test-worthiness and the required effort to test a coverage gap when measuring the quality of testing. The repeated mention of the effort to understand the generated test and the code under test, as well as the wish to receive the test at the time of writing the code, points to the value of investing in testing at an early stage in development, as the cost of adding the tests later is higher.

6.2 Implications for tool builders, developer supporters

A recurring concern in the second study was the effort to understand and complete the generated tests. When trying to outsource difficult parts of automation tasks like generating assertions on to human users, we need to make sure that we provide value compared to the user doing the whole task themselves, like writing complete tests from scratch. We saw understanding the generated test is a hurdle to completing it with assertions. With automated generation tools becoming much more popular these days (GitHub Copilot, ChatGPT, ...) the work of developers is moving from engineering solutions to evaluating and adapting solutions generated by machines. We conjecture that the next steps need to be to invest in supporting the developers in understanding generated code and tests. One way would be to study and build dedicated tools for this task.

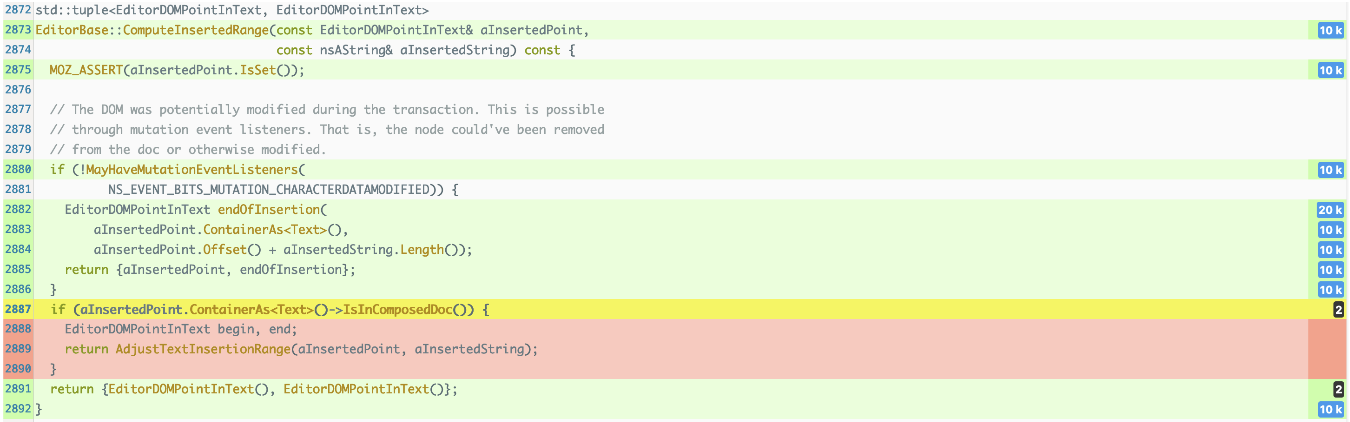
A different option would be to leverage the power of now popular large language models to make the generated fuzz tests more human-readable, or to generate assertions automatically by prompting the model with the generated test and the code under test. To make the generated tests more useful for the developers, we should extend the filter we presented in order to identify code that is worth testing. To reduce the cost of adding a test, we could generate the tests earlier in the development process: at the time the developer is writing the code or a reviewer is reviewing it, reducing the need of context switching.

6.3 Implications for researchers

In both our studies we made initial observations that not all coverage gaps are equally worth testing. This calls for a more detailed study on how to prioritize coverage gaps, what factors influence the test-worthiness of a coverage gap and how to reliably measure these factors. We conjecture that such factors would be very project / company / context dependent. Already in this context we saw that security concerns (point to zero-day bug in 1817159) might weigh stronger than pure code quality improvements (1817159 improving exception testing, but staying in backlog). In addition, we saw hesitancy to touch legacy code that has been running without a link to bugs for long years, as maybe the less risky option to not touch a running system.

Further, in both studies we saw a variety of ways the developers proposed to address our reports or pointers to coverage gaps. Next to writing a test, they also removed code or proposed to add

[10] https://bugzilla.mozilla.org/show_bug.cgi?id=1832450



```
2872 std::tuple<EditorDOMPointInText, EditorDOMPointInText>
2873 EditorBase::ComputeInsertedRange(const EditorDOMPointInText& aInsertedPoint,
2874                                  const nsAString& aInsertedString) const {
2875   MOZ_ASSERT(aInsertedPoint.IsSet());
2876
2877   // The DOM was potentially modified during the transaction. This is possible
2878   // through mutation event listeners. That is, the node could've been removed
2879   // from the doc or otherwise modified.
2880   if (!MayHaveMutationEventListeners(
2881           NS_EVENT_BITS_MUTATION_CHARACTERDATAMODIFIED)) {
2882     EditorDOMPointInText endOfInsertion(
2883         aInsertedPoint.ContainerAs<Text>(),
2884         aInsertedPoint.Offset() + aInsertedString.Length());
2885     return {aInsertedPoint, endOfInsertion};
2886   }
2887   if (aInsertedPoint.ContainerAs<Text>()->IsInComposedDoc()) {
2888     EditorDOMPointInText begin, end;
2889     return AdjustTextInsertionRange(aInsertedPoint, aInsertedString);
2890   }
2891   return {EditorDOMPointInText(), EditorDOMPointInText()};
2892 }
```

Figure 10: The coverage gap we discussed with D7. The gap is at editor/libeditor/EditorBase.cpp#2888-2890.

assertions for the regular fuzzing runs to be notified in case the coverage gap becomes reachable. This indicates that **functional testing is not the only way to "cover"/"secure" a line of code**, and metrics we develop to measure the testedness of code should include these other activities.

6.4 Threats to Validity

There are several threats to the validity of the observations in both our studies and the conclusions we draw from them. A threat to the *internal validity* is the presence of a social desirability bias, where the developer might have been inclined to answer overly positive about adding tests. To mitigate the impact on our conclusions, we closely report on the developer's statements and the visible actions on the code that followed. While we did receive rationales for why coverage gaps should be closed, we also report on the effort that developers saw and that in many cases led to them not following up and addressing the coverage gap.

Concerning *confirmability*, the threat that the results are shaped by the researcher instead of the respondents, the analysis of the chat conversations with the developers was independently performed by two authors, and we came to a consensus on the observations and conclusions. These and the summary of the observations from the Bugzilla reports were communicated to and confirmed by the other authors.

With respect to *internal generalizability*, we expect that our observations do generalize to industrial open-source projects and companies with a similar size and positioning towards testing. Looking at *external generalizability*, developers from projects with less familiarity to fuzz testing likely would not point to fuzzing as a way to address the coverage gaps. This and other findings should only carefully be generalized, and we encourage other researchers to replicate our studies in other contexts.

7 RELATED WORK

Previous work has studied the introduction of automated test generation tools in industrial contexts. Brunetto et al. [11] report on their experience introducing a tailored automatic GUI test generator in a medium-sized company. A difficulty they faced was the automatic generation of functional oracles, which they mitigated by providing rich reports to support engineers checking the effect of the test on the system. In their lessons learned, they note that automation is welcome in industry, but only useful if the testers can understand and interpret the produced tests. This matches the developer's comments in our study on the additional effort to understand the generated tests compared to writing the tests from scratch. Brunetto et al. report that integrating the output of the test generation into the workflow and tooling of the company was a key factor to enable the adoption of the tool. Further, manually-specified functional oracles would increase the effectiveness of generated test cases. However, a cost-effective way to define these automated oracles (for system-level UI tests in their case) is still an open challenge. In a similar vein, Mesbah et al. [24], who built a tool for automated test case generation for AJAX web applications, note the effort and required knowledge for a developer to specify invariants that can serve as oracles for the correctness of the software behavior. In a survey of 225 software developers, Daka and Fraser [15] find that automated test generation in mainly used with automated oracles, i.e., finding crashes or undeclared exceptions.

Almasi et al. [1] studied the industrial applicability of EvoSuite and Randoop in a financial company. Through generating tests for 25 real faults from the history of the companies' software system and a survey under their developers, they found that in more than half of the cases, more appropriate assertions would have led to the detection of the faults with generated tests. The developers expected the automated generation tools to integrate with their build pipeline and workflow, and were concerned about the readability of the generated tests, input data and assertions. Xie et al. [30] report on their experiences from industrial workshops on teaching testing tools, including the test generator Pex for C# which generates partial tests that developers have to write assertions for. They learned that the developers tended towards staying with the assertion-less automatically generated tests rather than writing assertions for them. Further they pointed to the need to explicitly teach them how they should interact with the generated tests and communicate why the tests were generated with such values.

Zhang et al. [35] present an approach to fuzz test remote procedure call APIs and evaluate it within an industrial context. They report challenges with isolating the test environment (resetting the state of the application, data preparation and mocking of external services) and propose to enable the fuzzer on the CI to promote its adoption in industry. They also point to the importance of non-flakiness and readability of the generated tests as crucial to be considered when testing industrial APIs. Plöger et al. [26] evaluated the usability of two fuzzers (AFL and libFuzzer) with computer science students. They reported very low usability across all steps of the fuzzing process and gave a variety of recommendations on how to improve, such as UI guidance through the fuzzing process, better error messages, and crash analysis support.

8 CONCLUSION AND FUTURE WORK

In this paper, we set out to explore whether partial tests generated by fuzzers and completed by developers can help alleviate the developer's effort of creating tests. For this, we developed a prototype within the Mozilla ecosystem. Through the discussions on 13 Bugzilla reports we created with our prototype, we observed that the code targeted by the tests is a main concern for developers before considering the fuzz tests. More specifically, the developers sometimes indicated that a coverage gap is not worth the effort to be tested. We dove deeper into the test-worthiness of coverage gaps by designing a filter to exclude small and early return coverage gaps. From discussing the remaining gaps with developers, we learned that the filters are effective in excluding clearly irrelevant coverage gaps, but there remain many considerations when deciding whether testing a coverage gap is worth the effort. Remaining gaps were considered not worth the effort to cover because the developers found it unlikely that there is a bug in the code, the code is unlikely to be reached, or should already be covered by other tests. In addition, we saw that there are other ways than functional tests that developers propose to address missing coverage.

Several opportunities for future work follow from our studies. Apart from the aforementioned implications for researchers, the factors of relevance concerning coverage gaps should be studied in other industrial or open source contexts, as we conjecture their priorities to be different between projects and developers. Constructing more reliable coverage calculations, that also include test suites not run on the regular CI runs and other quality assurance or security testing techniques, would provide a more accurate picture of missing coverage and therefore a more reliable basis to guide test generation efforts. Another interesting extension would be to combine the fuzz tests with an automatic approach for assertion generation and study whether confirming a generated assertion reduces the developer's understanding effort far enough to make the approach viable compared to writing tests manually from scratch.

ACKNOWLEDGMENTS

This research was partially funded by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032). A. Bacchelli acknowledges the support of the Swiss National Science Foundation for the SNSF Project 200021_197227. Further support came from the Swiss National Science Foundation (SNSF Grant 200021M 205146).

A FUZZING-DISCOVERED SECURITY VULNERABILITIES AT MOZILLA

The graph in Figure 2 is constructed by querying the Bugzilla database for all resolved security vulnerabilities with a critical or high rating on 2023-07-22. For all bugs that were opened in 2018, received a security rating "critical" or "high", and were resolved, run this query:

```
https://bugzilla.mozilla.org/buglist.cgi?
chfieldfrom=2018-01-01&
chfieldto=2018-12-31&
chfield=%5BBug%20creation%5D&
keywords=sec-high%2C%20sec-critical%2C%20&
classification=Client%20Software&classification=Developer%20
Infrastructure&classification=Components&classification=Server%20
Software&classification=Other&
query_format=advanced&
keywords_type=anywords&
resolution=FIXED&resolution=WONTFIX&resolution=INACTIVE&resolution
=DUPLICATE&resolution=WORKSFORME&resolution=INCOMPLETE&resolution=
SUPPORT&resolution=EXPIRED&
```

To only see bugs from this search that were reported by the fuzzing team, append to the above query:

```
emailreporter1=1&
emailassigned_to1=1&
emailtype1=exact&
emailcc1=1&
email1=fuzzing%40mozilla.com&
```

To obtain the values for the following years, we changed the year numbers in "chfieldfrom" and "chfieldto". As this is querying the public Bugzilla database, bugs that are (still) hidden from the public are not included in the results. Focusing on bugs that have been resolved might bias the results towards fuzzing because fuzzing reports are reproducible, which can make them quicker to fix than other bugs that are harder to diagnose or fix.

REFERENCES

[1] Mohammad Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 263–272.

[2] Domagoj Babic. 2017. SunDew: Systematic Automated Security Testing (Keynote). In *ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. ACM, 10.

[3] Kent L. Beck. 2003. *Test-Driven Development - By Example*. Addison-Wesley.

[4] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2019. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Trans. Software Eng.* 45, 3 (2019), 261–284.

[5] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, How, and Why Developers (Do Not) Test in Their IDEs. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 179–190.

[6] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2015. How (Much) Do Developers Test?. In *37th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 559–562.

[7] Antonia Bertolino. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *International Conference on Software Engineering (ISCE), Workshop on the Future of Software Engineering (FOSE)*. IEEE CS, 85–103.

[8] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Softw.* 38, 3 (2021), 79–86.

[9] Carolin Brandt and Andy Zaidman. 2022. Developer-centric Test Amplification. *Empir. Softw. Eng.* 27, 4 (2022), 96.

[10] Carolin Brandt and Andy Zaidman. 2022. How Does This New Developer Test Fit In? A Visualization to Understand Amplified Test Cases. In *Working Conference on Software Visualization (VISSOFT)*. IEEE, 17–28.

[11] Matteo Brunetto, Giovanni Denaro, Leonardo Mariani, and Mauro Pezzè. 2021. On introducing automatic test case generation in practice: A success story and lessons learned. *J. Syst. Softw.* 176 (2021), 110933.

[12] Magiel Bruntink and Arie van Deursen. 2004. Predicting Class Testability Using Object-Oriented Metrics. In *4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*. IEEE, 136–145.

[13] Gul Calikli and Ayse Bener. 2015. Empirical Analysis of Factors Affecting Confirmation Bias Levels of Software Engineers. *Softw. Qual. J.* 23, 4 (2015), 695–722.

[14] Bas Cornelissen, Arie van Deursen, Leon Moonen, and Andy Zaidman. 2007. Visualizing Testsuites to Aid in Software Understanding. In *11th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 213–222.

[15] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *25th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE CS, 201–211.

[16] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. 2019. Automatic Test Improvement With DSpot: A Study With Ten Mature Open-Source Projects. *Empir. Softw. Eng.* 24, 4 (2019), 2603–2635.

[17] Amirhossein Deljouyi and Andy Zaidman. 2023. Generating Understandable Unit Tests through End-to-End Test Scenario Carving. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 107–118.

[18] Mozilla Documentation. [n.d.]. Mochitest. https://firefox-source-docs.mozilla.org/testing/mochitest-plain/index.html

[19] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) and 13th European Software Engineering Conference (ESEC)*. ACM, 416–419.

[20] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM* 55, 3 (2012), 40–44.

[21] Bruce Hanington and Bella Martin. 2012. *Universal Methods of Design: 100 Ways to Research Complex Problems, Develop Innovative Ideas, and Design Effective Solutions*. Rockport.

[22] Pavneet Singh Kochhar, Xin Xia, and David Lo. 2019. Practitioners' Views on Good Software Testing Practices. In *41st IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE/ACM, 61–70.

[23] Brian Marick, John Smith, and Mark Jones. 1999. How to misuse code coverage. In *Proceedings of the 16th Interational Conference on Testing Computer Software*. 16–18.

[24] Ali Mesbah, Arie van Deursen, and Danny Roest. 2012. Invariant-Based Automatic Testing of Modern Web Applications. *IEEE Trans. Software Eng.* 38, 1 (2012), 35–53.

[25] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. 2016. The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation. In *38th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 547–558.

[26] Stephan Plöger, Mischa Meier, and Matthew Smith. 2023. A Usability Evaluation of AFL and libFuzzer With CS Students. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 186:1–186:18.

[27] Christian R. Prause, Jürgen Werner, Kay Hornig, Sascha Bosecker, and Marco Kuhrmann. 2017. Is 100% Test Coverage a Reasonable Requirement? Lessons Learned From a Space Software Project. In *Product-Focused Software Process Improvement - 18th International Conference, (PROFES) (Lecture Notes in Computer Science, Vol. 10611)*. Springer, 351–367.

[28] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2015. Automated Unit Test Generation During Software Development: A Controlled Experiment and Think-Aloud Observations. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 338–349.

[29] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C. Gall, and Alberto Bacchelli. 2019. On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later. In *16th International Conference on Mining Software Repositories (MSR)*. IEEE/ACM, 121–125.

[30] Tao Xie, Jonathan de Halleux, Nikolai Tillmann, and Wolfram Schulte. 2010. Teaching and Training Developer-testing Techniques and Tool Support. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 175–182.

[31] Wen Xu, Soyeon Park, and Taesoo Kim. 2020. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 971–986.

[32] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2023. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security.

[33] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.

[34] Google Project Zero. [n.d.]. The Great DOM Fuzz-off of 2017. https://googleprojectzero.blogspot.com/2017/09/the-great-dom-fuzz-off-of-2017.html

[35] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2022. White-box Fuzzing RPC-based APIs With EvoMaster: An Industrial Case Study. *CoRR* abs/2208.12743 (2022).