Crash-avoiding Program Repair

Xiang Gao
National University of Singapore
gaoxiang@comp.nus.edu.sg

Sergey Mechtaev
University College London
s.mechtaev@ucl.ac.uk

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

ABSTRACT
Existing program repair systems modify a buggy program so that the modified program passes given tests. The repaired program may not satisfy even the most basic notion of correctness, namely crash-freedom. In other words, repair tools might generate patches which over-fit the test data driving the repair, and the automatically repaired programs may even introduce crashes or vulnerabilities.

We propose an integrated approach for detecting and discarding crashing patches. Our approach fuses test and patch generation into a single process, in which patches are generated with the objective of passing existing tests, and new tests are generated with the objective of filtering out over-fitted patches by distinguishing candidate patches in terms of behavior. We use crash-freedom as the oracle to discard patch candidates which crash on the new tests. In its core, our approach defines a grey-box fuzzing strategy that gives higher priority to new tests that separate patches behaving equivalently on existing tests. This test generation strategy identifies semantic differences between patch candidates, and reduces over-fitting in program repair.

We evaluated our approach on real-world vulnerabilities and open-source subjects from the Google OSS-Fuzz infrastructure. We found that our tool Fix2Fit (implementing patch space directed test generation), produces crash-avoiding patches. While we do not give formal guarantees about crash-freedom, cross-validation with fuzzing tools and their sanitizers provides greater confidence about the crash-freedom of our suggested patches.

1 INTRODUCTION
For a given program with a defect, the goal of program repair is to eliminate the defect by automatically transforming the program source code. Typically, program repair systems construct a space of candidate patches (set $S$ in Figure 1) and search for a patch that passes the given tests. Such patches that pass given tests are called plausible patches (set $P$ in Figure 1) in the program repair literature. Since a test suite is an incomplete specification, only part of plausible patches are correct (set $C$ in Figure 1), and the remaining patches merely overfit the tests. When we repair a program crash, the over-fitted patches may still cause program crash for the test outside of the given test suite.

In this work, we propose to divide the set of plausible patches $P$ into two subsets $P_{\text{crash-free}}$ (crash-free plausible patches), and $P_{\text{crashing}}$ (crashing plausible patches) and suggest that program repair should aim to find a patch from the set $P_{\text{crash-free}}$ that is a patch that passes given tests and does not cause crashes for the inputs outside of the repair test suite. Although crash-freedom is implicitly assumed to hold for correct patches, existing program repair systems do not guarantee this property and may generate patches causing crashes or even introduce new crashes and vulnerabilities.

Ideally, the property of crash-freedom should be proven. However, software verification techniques often suffer from scalability limitations that prevent them from being applied to real-world software. A more practical and widely-used approach is to test crash-freedom. In the context of program repair, testing crash-freedom implies finding an overapproximation of $P_{\text{crash-free}}$.

A prominent group of testing techniques that were successfully used to find serious vulnerabilities in popular software is coverage-based greybox fuzzing [1]. These techniques resort to compile time instrumentation which guides the generation of test inputs. In these algorithms, inputs are randomly mutated to generate new inputs, and higher priority is assigned to inputs that exercise new and interesting program paths. Whether a generated input exercise new paths is predicted based on whether new control flow transitions are exercised; this is found out with the help of the compile-time instrumentation. The main intuition of these techniques is that covering more program paths (that correspond to different semantic partitions of the input space) enables them to cover more parts of program functionality and therefore find more crashes.

Coverage-based greybox fuzzing can be applied to detect crashes in automatically generated patches in the following way: (1) generate a high coverage test suite using fuzzing for the original program, and (2) run this test suite on all plausible patches $P$ to discard those that introduce crashes, and thus find an over-approximation of $P_{\text{crash-free}}$. However, we argue that this approach is ineffective for the following two reasons. First, each candidate patch alters the semantics of the original program and therefore might induce different semantic partitions of the input space, so tests generated for the
original program might not adequately cover the functionality of the patched program. Second, to divide the set of plausible patches \( P \) into subsets \( P_{\text{crash-free}} \) and \( P_{\text{crashing}} \) (dotted line in Figure 1), the generated tests should also differentiate patches in the search space.

To take the above considerations into account, we suggest that test generation for program repair should not be based merely on the coverage of the original program, but also on the coverage of the divergences introduced by the patches in the search space. Thus, a test suite produced by our method is not just aimed to cover functionality of the original program, but also (1) functionality that is altered by the candidate patches, and (2) functionality that differs across candidate patches. Since such a test suite is more likely to find divergences among plausible patches \( P \), consequently it is more likely to differentiate between \( P_{\text{crash-free}} \) and \( P_{\text{crashing}} \).

As a practical realization of this concept, we propose a new algorithm that fuses patch and test generation into a single process. In this process, patches are generated with the objective of passing existing tests, and new tests are generated with the objective of differentiating patches. To increase the likelihood of differentiating two versions of a program, it is necessary to generate a test that reaches the divergent statements, and the divergent statements produce divergent values during the test execution. However, since there could be many plausible patches, it is inefficient to separately generate tests to distinguish each pair of these patches. Instead, we propose to group patches into test-equivalence classes, sets of patches that demonstrate equivalent behaviour on existing tests. These are called as patch partitions. When generating tests, we assign higher priority to those tests that refine patch partitions into finer-grained partitions, since such tests cover previously uncovered semantic differences between candidate patches. This allows us to efficiently cover divergences between candidate patches without explicitly considering all pairs of patches.

**Contributions.** Program repair techniques suffer from over-fitting, and cannot distinguish correct patches from plausible incorrect patches. Our work is a step towards rectifying this problem. First and foremost, we propose to tightly integrate testing and program repair to effectively discard crashing patches. Secondly, we devise fuzz testing strategies to guide test generation towards differentiating patches in the search space. Our fuzz testing tool Fix2Fit actively exploits the search space of patches maintained as patch partitions computed via test equivalence relations. Tests are generated with the goal of refining the patch partitions. Last but not the least, we construct a set of subject programs from OSS-Fuzz (a popular open-source repository from Google) capturing a wide variety of software vulnerabilities. We evaluate our patch-aware fuzz testing strategies as embodied by our tool Fix2Fit on the constructed benchmark, and show significant (up to 60% reduction) in the space of candidate patches. If the oracles of a few (5-10) newly generated tests are available, this reduction increases to 93% on our OSS-Fuzz subjects.

## 2 RELATED WORK

**Test-based automated program repair** Test-based automated program repair treats the provided test suite as specification of intended behavior and generates patches that make program pass all the given tests. Typically, patch generation methods include:

- **search-based approaches** search the correct patch from a huge patch space using meta-heuristic [2, 3], random search [4] or test-equivalence analysis [5].
- **constraint solving based approaches** extract constraints from test executions, and synthesize patches by solving the constraints [6–9].
- **learning-based approaches** use a model to select patches that are more likely to fix the defect based on existing patches [3, 10] or program context [11].

While these approaches are able to generate high-quality patches according to the provided tests, the weakness of test suites remains a challenging problem in test-based program repair. Due to the incompleteness of test suites, the generated patches may overfit the available tests and can break untested functionality. Over-fitting problem exists in both search-based repair [12] and semantics-based repair techniques [13]. To alleviate the over-fitting problem, existing approaches filter out overfitted patches by techniques such as defining anti-pattern [14] or generating simplest program repairs [8]. Different from those approaches, which add more heuristic in the search process, Fix2Fit enhances the program specification via test case generation. Actually, Fix2Fit is orthogonal to those techniques and can be combined with them in the future.

**Test generation for program repair** Automatically generating more tests for automated program repair is a useful strategy to alleviate the overfitting problem. Existing approaches generate additional test cases using symbolic execution, grey box fuzzing [15] (like AFL) or evolutionary algorithm [16] (like EvoSuite [17]). All those approaches are designed to generate tests with the goal of covering the patched methods or statements, but they do not take the patch semantics into consideration. DiffTGen [18], the work most relevant to us, generates test inputs that exercise syntactic differences, monitors execution results and then selects tests that uncover differences between the original faulty program and the patched program. Compared with DiffTGen where the patch is validated one by one, Fix2Fit is more efficient since it examines the patches in the same patch partition together. Besides, different from all existing approaches, Fix2Fit utilizes semantic difference between patches as a search heuristic and guides the test case generation process, so that we can efficiently find more semantic discrepancies across patches. Inferring the expected behaviors (oracles) for newly generated test inputs is another challenging problem. Existing approaches infer oracles of tests based on test similarity [19], developers’ feedback [18, 20] or some obvious oracles (like memory safety [15]). In contrast, Fix2Fit utilizes security oracles from sanitizers to avoid introducing crashes or vulnerabilities.

**Goal-directed test generation** Goal-directed test generation can be used to generate test inputs to maximize code coverage [1, 21], cover the changes in a patch [22] or find behavioral asymmetries between programs (differential testing) [23]. Symbolic execution employs constraint collection and constraint solving to systematically and effectively explore the state space of feasible execution paths [24]. By precisely controlling the constraint collection and solving process, symbolic execution could generate test inputs that achieve particular goals such as directed-ness and coverage [25–28]. In contrast to symbolic execution, grey box fuzzing does not involve heavy machinery of symbolic execution and constraint solving. Greybox fuzzing directs the search to achieve a certain goal by adjusting the mutation strategy according to the
Listing 1: Buffer overflow vulnerability in FFmpeg

```c
int decode_ddsl(ByteContext *gb, uint8_t *frame, int width, int height)
{
    ... segments = bytestream2_get_le16(gb);
    while(segments--) {
        if (bitbuf & mask) {
            ...
        } else if (bitbuf & (mask << 1)) {
            v = bytestream2_get_le16(gb)+2;
            if (frame - frame_end < v)
                return AVERROR_INVALIDDATA;
            frame ++; v; }
        else {
            int remaining_space = frame_end - frame;
            if (remaining_space < width)
                return AVERROR_INVALIDDATA;
            frame[0] = frame[1] = frame[width] =
            frame[width+1] = bytestream2_get_byte(gb);
            frame ++ 2;
            frame[0] = frame[1] = frame[width] =
            //buffer overflow location
            frame[width+1] = bytestream2_get_byte(gb);
            frame ++ 2;
        }
    }
}
```

information collected at run-time with the help of compile-time instrumentation. Greybox fuzzing has been demonstrated to be useful for increasing code coverage [1, 29], reaching target location [22], and finding behavioral asymmetries between programs [23]. Different from those fuzzing techniques, Fix2Fit is designed with the objective of finding semantic discrepancies between patches, and we take the semantic of patches into consideration.

3 OVERVIEW

In this section, we give a high-level overview of our approach to generate crash-free patches by presenting an example from FFmpeg. FFmpeg is a collection of libraries and programs for handling video, audio and other multimedia files, streams. A buffer overflow vulnerability is reported by OSS-Fuzz2 in May, 2017. This vulnerability is caused by incorrect bounds checking when FFmpeg decodes DirectDraw Surface (DDS) files2. Listing 1 shows the key code snippet as well as its patch. The decode method in Listing 1 takes four parameters, where gb stores the origin data of the input image, width and height are initialized based on the information from input image header, and frame is a buffer to store decoded data. If remaining_space is equal to width+3 (line 15), an invalid buffer access will occur in line 23, since it will overwrite the memory locations after frame_end. The correct patch3 for this vulnerability is to modify the condition in line 15 from width+3 to width+4.

Automated program repair (APR) takes a buggy program and a set of test cases (including failing tests which cause program crash) as inputs. Since the tests do not cover all program functionalities, APR tools may generate many over-fitted patches which make program pass all the test suite but do not actually fix the bug. Given the failing test case and a set of supported transformations, 1807 plausible patches are generated to fix the buffer overflow vulnerability. Column plausible patch in Table 1 shows part of patches that can make the program pass the failing test. Out of them, the fourth and sixth patches are semantically equivalent to the developers’ patch. However, other patches over-fit the existing test set. Those patches fix the crash triggered by existing test set, but they do not completely fix this vulnerability and even introduce new vulnerabilities (e.g. the patched program using first patch crashes when remaining_space is equal to width+2). The fundamental reason is that the search space of candidate patches is under-constrained.

To tighten the search space and rule out crashing patches, one solution is to automatically generate more test cases. This leads to the following research question: how to generate test cases that can filter out a large fraction of over-fitted patches?

Existing fuzzing techniques are not suitable for efficiently generating tests to constrain the patch space. Most fuzzing tools (e.g. AFL [1]) favour the mutation of input with the goal of finding unexplored statements, or enhancing code coverage. Different from program testing, the role that fuzzing plays in repair is to generate test cases to find discrepancies between patches and filter out overfitted patches instead of covering more statements or paths. In this example, test cases that can drive the execution to the patch location with different program states (values of remaining_space, width) are expected. If behavioral discrepancies between plausible patches are disclosed, we make one step forward to distinguish crash-free patches from crashing patches, so that the chance to filter out over-fitted patches is increased.

To efficiently generate test inputs that can filter out overfitted patches and differentiate patches, we propose a strategy to integrate test generation and program repair. Our main intuition is, if one test is able to find the discrepancies between patches, its neighbors are also likely to find discrepancies. Table 1 shows the patch behaviors over four tests. The patch behavior is shown by its effectiveness in repairing vulnerability and expression value, where ✓ and ✗ represent whether buffer overflow vulnerability is triggered or not by each test, T and F represent the value of patch expression (true or false). Suppose these four patches are generated in order, with values of remaining_space equals to width+2, width, width+4, and width+6 respectively. For instance, the expression value of patch 2 (remaining_space > width + 2) is false (F) under test T1, and program fixed by this patch still crashes (X) under T1, so that patch 2 is filtered out and will not be considered in the following iterations. Test input T2 is able to find the discrepancies between patches, and rule out two over-fitted patches. Correspondingly, T2 and T3, which

---

1https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=1345
2DDS is an image file format for storing texture and environments
3https://github.com/FFmpeg/FFmpeg/commit/52dfb

<table>
<thead>
<tr>
<th>Id</th>
<th>plausible patch</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>remaining_space&gt;width+1</td>
<td>✓</td>
<td>✗</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>remaining_space&gt;width+2</td>
<td>✗</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>3</td>
<td>remaining_space&gt;width+3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>remaining_space&lt;width+3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td>remaining_space&lt;=width+3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td>remaining_space&lt;width+4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>7</td>
<td>remaining_space&lt;width+5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>8</td>
<td>remaining_space&lt;width+6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
are two neighbors of $T_1$ (a single increment or decrement mutation over `width` or `v` on line 9), can also find discrepancies.

To guide the test generation process, Fix2Fit adopts an evolutionary algorithm similar to the popular AFL fuzzer [1]. AFL undergoes compile-time instrumentation to capture control flow edges, and at run-time during test generation it uses the instrumentation to predict whether a newly generated test exposes new control flows. Tests which expose new control flows are favored and they are retained for further examination by mutating them further. In addition to this code coverage enhancing heuristic used in AFL, we propose a new heuristic: we favor tests with greater ability to distinguish plausible patches. In our example program, AFL will not retain $T_1$, since it does not cover new statements (edges). However, in our proposed patch-aware fuzzing, $T_1$ will be retained for further mutation, so we have a chance of finding tests like $T_2$ or $T_3$ via mutation. In addition, this chance of finding tests which find discrepancies across patches, can be further increased by assigning higher “energy” to $T_1$ (meaning more mutations of $T_1$ will be constructed by the fuzzer).

Out of eight patches given in Table 1, three plausible patches (1, 2, 5) can be ruled out, since the program constructed by those patches still crashes over some tests. For the remaining five plausible patches, the patched program does not crash, but the semantic behaviors of them are different (two of them are correct). The remaining incorrect patches cannot be ruled out due to the lack of oracles of the generated tests. If the oracle of certain tests such as $T_3$ is provided (could come from more fine-grained program analysis or developers), all the incorrect patches can be ruled out.

4 BACKGROUND

We denote a program as $e$ where $e$ is an arbitrary expression of bounded size. Patches that pass all the given tests are called plausible patches. Since a test suite is an incomplete specification, plausible patches may not be correct, but merely overfit the given tests. Besides, the plausible patches may even introduce new bugs and break the under-tested program functionality. The most basic approach to patch generation is the generate-and-validate algorithm [30] that enumerates and tests individual patches. This algorithm, however, scales only to small search spaces because of the cost of test execution. Test-equivalence analysis [5, 31, 32] can significantly optimize this process.

Definition 4.1 (Test-equivalence). Let $p$ and $p'$ be programs, $t$ be a test. We say that $p$ is test-equivalent to $p'$ w.r.t. $t$ if both $p$ and $p'$ produce same output by executing $t$.

In some cases, test-equivalence of two programs can be detected without executing each of them individually, but instead performing dynamic analysis while executing only one of them, which helps to reduce the number of test executions required for evaluation. In this work, we consider one such analysis referred to as value-based test-equivalence [5]. The search space of patches is represented as a collection of patch partitions. The patch partitions are constructed by using a value-based test-equivalence relation.

Definition 4.2 (Value-based test-equivalence). Let $e$ and $e'$ be expressions, $p$ and $p'$ be programs such that $p' := p[e \mapsto e']$, $t$ be a test. We say that $p$ is value-based test-equivalent to $p'$ w.r.t. $t$ if $e$ is evaluated into the same sequence of values during the execution of $p$ with $t$, as $e'$ during the execution of $p'$ with $t$.

4.2 Greybox fuzzing

We briefly describe how Greybox Fuzzing (e.g. AFL [1]) works in Algorithm 1. Given a set of initial seed inputs $T$, the fuzzer chooses $t$ from $T$ (line 2) in a continuous loop. For each selected $t$, the fuzzer determines the number of tests to be generated by mutating $t$, which is called the energy of $t$, and its assignment is dictated by a power schedule. The fuzzer generates new inputs by mutating $t$ according to defined mutation operators and the power schedule. New input $t'$ will be added to the circular seed queue (line 7) for further mutation if it is a “interesting” input, meaning it potentially exposes new control flows as deemed from the compile-time instrumentation.

**Algorithm 1: Greybox Fuzzing**

```plaintext
input: seed inputs $T$
while timeout is not reached do
  $t :=$ chooseNext($T$);
  $energy := assignEnergy(t)$;
  for $i$ from 1 to $energy$ do
    $t' := mutate(t)$;
    if isInteresting($t'$) then
      $T := T \cup t'$;
  end
end
```

AFLGo [22], an extension of the popular grey-box fuzzer AFL, directs the search to given target locations. In AFLGo, an estimation of the distance of any basic block to the target(s) is instrumented at compile time, and these estimates are used during test generation to directly search to the targets. Specifically, tests with lower estimated distance to the target are preferred by assigning more energy to these tests, and this energy difference increases as temperature decreases. The temperature is controlled by a cooling schedule [33],...
which dictates how the temperature decreases over time. Based on cooling schedule, the current temperature $T_{\text{exp}}$ is defined as:

$$T_{\text{exp}} = 20 \cdot \frac{\text{time}_{\text{exp}}}{\text{time}_c}$$  \hspace{1cm} (1)$$

where $\text{time}_{\text{exp}}$ is user-defined time to enter “exploitation” (preferring tests deemed closer to the target) from exploration, $\text{time}_c$ is current execution time. Given the current temperature $T_{\text{exp}}$, normalized distance $d(t, T_k)$ between test $t$ and target location $T_k$, AFLGo introduces an annealing-based power schedule (APS):

$$\text{aps}(t) = (1 - d(t, T_k)) + (1 - T_{\text{exp}}) + 0.5T_{\text{exp}}$$  \hspace{1cm} (2)$$

and determines the energy assigned to $t$ by multiplying the energy assigned by AFL with a power factor calculated using APS:

$$\text{energy}_{\text{aflg}}(t) = \text{energy}_{\text{afl}}(t) + 2^{10 \cdot \text{aps}(t) - 5}$$  \hspace{1cm} (3)$$

5 METHODOLOGY

Fix2Fit is designed to generate new test cases to efficiently rule out over-fitted plausible patches and generate crash-free patches. Our goal is to strengthen the filtering of patches by adding additional test cases. Specifically, Fix2Fit observes the semantic differences across plausible patches, and then guides the test generation process. Fix2Fit utilizes the notion of separability: the ability to find semantic discrepancies between plausible patches. To represent the semantic discrepancies, we group all patches showing same semantic behavior under all available test cases into an equivalence class, which is called a patch partition. More formally,

**Definition 5.1 (Patch Partition).** Let $T$ be a set of available test cases and $P$ be a set of plausible patches of program $p$. The patched program by patch $p_i \in P$ is denoted as $p[e \mapsto e_i]$. For any two patches $p_i, p_j \in P$ that belong to the same equivalent partition $EP$, if $p[e \mapsto e_i] \neq p[e \mapsto e_j]$ w.r.t new test $t$, we say test $t$ refines partition $EP$. Different from existing fuzzing techniques that maximize code coverage (AFL), our approach is to maximize semantic discrepancies across patches (thereby refining partition). To find more semantic discrepancies between plausible patches, we essentially generate test cases that can make the test suite $T' = T \cup \{t\}$ to find more semantic discrepancies across patches (thereby refining partition).

### 5.1 Integration of Test Generation and Repair

Figure 2 presents a visual summary of our integrated testing and repair loop.

**Figure 2: Structure of integrated testing and repair loop**

Fix2Fit generates the initial set of plausible patches by inheriting the traditional Generate and Validate approach, where a set of patch candidates are generated and evaluated using a provided set of test cases (line 1). Incorrect patches are filtered out in the evaluation process, and a set of plausible patches are returned back. Besides plausible patches, it groups patches with same semantic behavior into a set of patch partitions (as per the value-based test equivalence Definition 5.1). The plausible patches may be over-fitting, and the patch partitions can be broken by generating more tests.

To filter out over-fitted patches by generating new tests, the newly generated tests must at least reach the patch location. We instrument program $p$ with the patch location as target (Line 3) to produce an instrumented program $p'$. At runtime, the instrumentation is used to calculate code coverage and the distance to the patch location (line 10), and also the separability for each newly generated test. The separability of a test $t'$ captures its ability to find semantic discrepancies between plausible patches.

For each newly generated input $t'$, Fix2Fit first evaluates whether $t'$ drives the execution to the patch locations. If test $t'$ reaches any target (Lines 11-13), procedure `refine_and_filter` is invoked, which refines the patch partitions and also filters out patch partitions as follows.

---

**Algorithm 2: Patch-aware Greybox Fuzzing**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$Par := \text{genPlausiblePatches}(p, T)$; // generate set of patch partitions</td>
</tr>
<tr>
<td>2</td>
<td>$plocs := \text{extractPatchLocs}(Par)$; // extract set of patch locations</td>
</tr>
<tr>
<td>3</td>
<td>$p' := \text{instrument}(p, plocs)$; // instrument fuzzing targets</td>
</tr>
<tr>
<td>4</td>
<td>$T_{\text{new}} := {}$</td>
</tr>
<tr>
<td>5</td>
<td>$T := T_{\text{new}}$</td>
</tr>
<tr>
<td>6</td>
<td>while true do</td>
</tr>
<tr>
<td>7</td>
<td>$t := \text{chooseNext}(T)$</td>
</tr>
<tr>
<td>8</td>
<td>for $i$ from 1 to $\text{energy}$ do</td>
</tr>
<tr>
<td>9</td>
<td>$t' := \text{mutate}(t)$; $T_{\text{new}} := T_{\text{new}} \cup {t'}$;</td>
</tr>
<tr>
<td>10</td>
<td>$(\text{isReached}) = \text{exec}(p', t')$;</td>
</tr>
<tr>
<td>11</td>
<td>if $\text{isReached}$ then</td>
</tr>
<tr>
<td>12</td>
<td>$Par := \text{refine_and_filter}(Par, t')$;</td>
</tr>
<tr>
<td>13</td>
<td>// Break patch partitions &amp; remove over-fitted partitions</td>
</tr>
<tr>
<td>14</td>
<td>$\text{sep} = \text{separability}(t', T_{\text{new}})$ // Equation 4</td>
</tr>
<tr>
<td>15</td>
<td>if $\text{isInteresting}(\text{coverage, sep})$ then</td>
</tr>
<tr>
<td>16</td>
<td>$T := T \cup {t'}$;</td>
</tr>
<tr>
<td>17</td>
<td>end</td>
</tr>
<tr>
<td>18</td>
<td>end</td>
</tr>
<tr>
<td>19</td>
<td>end</td>
</tr>
<tr>
<td>20</td>
<td>if $\text{timeout}</td>
</tr>
<tr>
<td>21</td>
<td>break;</td>
</tr>
<tr>
<td>22</td>
<td>end</td>
</tr>
<tr>
<td>23</td>
<td>Output: remaining patch partitions $Par$</td>
</tr>
</tbody>
</table>
• first, refine_and_filter refines the current patch partitions \( \text{Par} \) using test \( t' \). The refinement process may break the existing patch partition into several sub-partitions since the underlying value-based test-equivalence relation now also considers the newly generated test \( t' \).
• After the patch partitions are refined using \( t' \), the procedure refine_and_filter checks which of the patch partitions can be shown to be over-fitting (patches which crash on test \( t' \)) and filters out those patch partitions.

Separability of a generated test \( t' \) (the patch-awareness in our fuzzing method) is exploited along two dimensions: (1) it is used in power schedule to determine the energy assigned to new test \( t' \) as shown in line 15, and (2) it is used to determine whether the generated input \( t' \) is added to the seed input set \( T \) for further investigation/mutation (Lines 16-17).

The integrated fuzzing and repair algorithm is terminated on timeout, or when all plausible patches are filtered out.

5.2 Separability of Test Cases

In Algorithm 2, test generation is guided by the behavioral differences across plausible patches. The ability of a test to find semantic discrepancies between plausible patches is formalized as separability. We now explain how the separability is calculated.

When a new test \( t' \) is introduced, its effects on the current patch partitions can be captured in two ways: (1) patch filtering: rule out over-fitted patches (2) partition refinement: refine existing patch partition into several sub-partitions. Both of these can be used to calculate the separability of test \( t' \), which in turn determines the “energy” assigned to \( t' \) in fuzzing.

We argue that the partition refinement is a better heuristic than patch filtering for the purpose of guiding fuzzing. In the fuzzing process, by mutating a test with high separability, we hope that the generated neighbors are also tests with high separability. If we define separability in terms of number of over-fitted/crashing patches filtered, we note that whether the patch is crashes on new test \( t' \) or not often depends on very specific values, for instance divide-by-zero error can only be triggered when input is 0. Therefore, we cannot assume that by mutating a test which exposes crashes, we are also likely to get tests exposing crashes.

Compared to patch filtering, partition refinement is a smoother metric, since the patches are grouped into partitions using test-equivalence relation and whether partitions can be refined only depends on the values of patch expressions. In other words, if one test \( t' \) is able to pin-point semantic differences between patch candidates (refine patch partitions), its neighbors (obtained by mutating \( t' \)) also have high chance to find semantic differences between patch candidates. Once we generate one test that can refine patch partitions, it is more likely that we can distinguish the crash-free patches from crashing patches, and as a result, rule out over-fitted patches. Based on this intuition, we define the separability of test as its ability to refine test-equivalence based patch partitions.

Our notion of separability judges how much refinement is observed on the patch partitions once a new test is introduced. Given a set of patch partitions \( \{P_1, P_2, \ldots P_n\} \), and a newly generated test \( t' \), if the patches in partition \( P_i \) show different behaviors on test \( t' \), we say \( t' \) refines partition \( P_i \). We use \( b(t') \) to represent the number of patch partitions that can be refined by test \( t' \). Fix2Fit always maintains a set \( T_{\text{new}} \) of newly generated tests, as shown in Algorithm 2. We define the separability of test \( t' \) as \( b(t') \) divided by maximum \( b(t) \) of any pre-generated test \( t \in T_{\text{new}} \):

\[
\text{separability}(t', T_{\text{new}}) = \frac{b(t')}{\max_{t \in T_{\text{new}}} b(t)} \quad (4)
\]

5.3 Power Schedule

We now define the notion of power schedule, which is a measure of the “energy” with which the neighborhood of a test is investigated (line 14 of Algorithm 2). Our goal is to investigate those tests more, which can differentiate between plausible patch candidates.

To differentiate plausible patches in the search space, we should first generate tests that reach patch location. Therefore, we inherit the power schedule of the directed grey-box fuzzer AFLGo [22], which directs the search to given target locations. Specifically, tests with lower estimated distance to the target are preferred by assigning more energy to these tests. Apart from reaching patch locations, generating divergent program states in the patch location is necessary to differentiate plausible patches. Fix2Fit prioritizes the tests with higher separability by assigning more energy to these tests. Note that separability of a test is calculated at run-time with the help of compile-time instrumentation.

To generate divergent program states in the patch location, two kinds of tests are needed: (1) tests that make execution reach patch location following various paths (2) tests that make execution reach patch location following same path but with different values (to refine value-based test-equivalence relation). To take both kinds of tests into consideration, we utilize the cooling schedule [33] notion adapted from simulated annealing. Specifically, the degree to which a test with high separability is preferred (over a test with low separability) is increased over execution time ("temperature decreases" using the simulated annealing terminology). In other words, Fix2Fit performs exploration at the very beginning to explore various paths, and gradually changes to exploitation to differentiate plausible patches. Given current temperature \( T_{\text{exp}} \) (as defined in Equation 1) as well the separability \( t' \) of test \( t \), our power schedule is defined as:

\[
\text{schedule}(t') = \text{separability}(t') \cdot (1 - T_{\text{exp}}) \quad (5)
\]

Thus \( \text{schedule}(t') \in [0, 1] \). The behavior of this power schedule is illustrated in Figure 3. We describe the integration of this power schedule into a fuzzer. Suppose energy\(_{\text{AFLGo}}(t')\) is the energy assigned to \( t' \) by AFLGo, we define the integrated energy as:

\[
\text{energy}(t') = \text{energy}_{\text{AFLGo}}(t') \cdot \text{schedule}(t')^{\log_2 \text{Max Factor}} \quad (6)
\]
"interesting", if it is predicted to expose new control flows (and

The absence of program crashes may not be sufficient to guarantee

This appears as the procedure isInteresting

are not only checked for crashes, but are also checked against all

errors or accesses to a dangling pointer. The patch partitions

e.g., checking by introducing sanitizers, including UndefinedBehavior-

improving the chance to refine patch partitions. Besides tests which

new control flows is aided by compile-time instrumentation. In our

2. In existing coverage based grey-box fuzzers, a test is deemed

Coverage-based greybox fuzzers always maintain a seed queue
to save “interesting” tests for further mutation and investigation. This

Different from AFLGo where the optimization goal (reaching
target location) is fixed, our optimization goal keeps changing as the

Specifically, if test \( t_1 \) refines a specific patch partition \( EP \), \( t_1 \) will

Moreover \( EP \) will be refined into several sub-partitions. However, when we generate test \( t_2 \) which can potentially refine

the partition \( EP \) into same sub-partitions, this does not necessarily

contribute to high separability of \( t_2 \) since \( EP \) has already been

refined (broken into many partitions) by now.

5.4 Is Interesting?

Coverage-based greybox fuzzers always maintain a seed queue
to save “interesting” tests for further mutation and investigation. This

appears as the procedure isInteresting in line 15 of Algorithm

2. In existing coverage based grey-box fuzzers, a test is deemed

“interesting”, if it is predicted to expose new control flows (and

hence improve code coverage); the prediction about discovering

new control flows is aided by compile-time instrumentation. In our

patch-generation guided fuzzier Fix2Fit, on top of retaining tests

exposing new control flows, we also want to retain tests which

makes execution follow a path but with different values thereby

improving the chance to refine patch partitions. Besides tests which

improve code coverage, Fix2Fit also regards the tests with non-zero

separability as “interesting” and adds them to seed queue for

further mutation. As a result, we retain tests which are capable of

distinguishing between existing patch partitions, and the mutations

of such tests are examined by the fuzzier in Algorithm 2.

5.5 Sanitizer as Oracles

The absence of program crashes may not be sufficient to guarantee

program correctness. To mitigate this problem, we enhance patch

checking by introducing sanitizers, including UndefinedBehavior-

Sanitizer\(^2\) (UBSan) and AddressSanitizer\(^3\) (ASan). UBSan is used to

catch various kind of undefined behaviors during program execution,
e.g. using misaligned or null pointer, signed integer overflow,

ASan is a tool that detects memory corruption bugs such as buffer

overflows or accesses to a dangling pointer. The patch partitions

are not only checked for crashes, but are also checked against all

available sanitizers.

6 IMPLEMENTATION

The architecture of Fix2Fit tool is shown in Figure 4. Fix2Fit takes as

inputs the buggy program and test suites extracted from OSS-Fuzz

benchmark, and outputs a set of crash-free patches. The initial test-

suite is composed of available developer test cases and the failing

tests from OSS-fuzz benchmark. Fix2Fit consists of three main

components: Candidate generator, Runtime and Guidance engine.

Candidate Generator takes the buggy program and tests as inputs and

generates a pool of patch candidates. The Runtime executes the

tests generated by fuzzing, collects necessary information, including

code coverage \( cov \), distance to the patch location \( dis \) and separability

\( sep \), and refines the patch pool. Guidance engine is used to guide

the fuzzing according to all the information collected at runtime.

Instrumentation: To enable Fix2Fit’s grey-box guidance, we

first of all instrument the buggy program to gather information on the

path executed for each test input. To collect the distance to patch

locations, we inherit the instrumentation strategy used in

AFLGo\([22]\), where the estimated distances between basic blocks are

calculated and injected at compile-time. Besides, we insert a

logging instruction after each basic block to collect the execution

trace, which is then used for fault localization and for determining

whether the patch location is reached. To enhance the checking of

patch candidates, we instrument the buggy program using Clang’s

sanitizers, including Undefined Behavior Sanitizer (UBSan) and

Address Sanitizer (ASan). After the instrumentation with sanitizers,

we can treat the violation of sanitizer as normal program crash. All

instrumentation is completed at compile-time using Clang.

Candidate Generator We first generate the search space according to pre-defined transformation operators. The transformations supported in our prototype include: changing the right-hand side of an assignment, condition refinement and adding if-guard. All the operators are borrowed from Prophet\([34]\), Angelix\([9]\) or F1X\([5]\). The plausible patch candidates are grouped into patch partitions based on their runtime value. To collect the run-time values of patches, Fix2Fit synthesizes a procedure, say \( proc_{\text{allpatch}} \) enumerating all plausible patches, and generates a meta-program by dynamicaly replacing the to-be-fixed expression with a call to this procedure. At runtime, the procedure \( proc_{\text{allpatch}} \) is invoked when the patch location is reached. By controlling the enumeration strategy, this procedure \( proc_{\text{allpatch}} \) can generate run-time values for all the patches with one run and can select the run-time value of one particular patch to return. This mechanism enables us to generate and refine patch partitions with one run for each test. Patch partitions are saved and maintained in the patch pool.

Runtime and Guidance engine The main procedure of fuzzing

is built on top of the directed greybox fuzzer AFLGo\([22]\). Different from AFLGo, Fix2Fit evaluates each of the newly generated tests on the patched programs at runtime, and continuously maintains and refines the patch partitions based on test-equivalence relations. Since new tests are generated, the test-equivalence relations are refined and hence the patch partitions are refined. Since our goal is to expose the semantic discrepancies between patches, the Guidance engine also takes the separability (Equation 4) into account.

\(^2\)UBSan website: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

\(^3\)ASan website: https://clang.llvm.org/docs/AddressSanitizer.html
7 EVALUATION

We perform the evaluation on the effectiveness of Fix2Fit in generating test inputs, filtering out over-fitted patches and refining patch partitions. Our research questions are as follows.

RQ1 What is the overall effectiveness of Fix2Fit in ruling out over-fitted patches?

RQ2 Is Fix2Fit effective for generating crash-free patches?

RQ3 How far can Fix2Fit reduce the pool of patch candidates, if the oracles of only a few (say 5-10) tests are available? Can this reduction lead to a sufficiently small number of patch partitions which can be manually examined?

7.1 Benchmark Selection

To evaluate our technique, we do not use existing benchmarks since (1) some existing benchmarks are over-engineered where the given tests are already complete enough to generate correct patches (2) we focus on generating crash-free patches for software crash or vulnerabilities, while most of the defects in existing subjects are logic errors e.g. Manybugs [35] and Defects4j [36]. Instead, we select a set of real-world subjects from the OSS-Fuzz (Continuous Fuzzing for Open Source Software) dataset. OSS-Fuzz, which has recently been announced by Google, is a continuous testing platform for security-critical libraries and other open-source projects. We select projects which contain a large number of bugs and try to reproduce the defects by installing the corresponding versions in our environment. The defects that cannot be reproduced in our environment are not considered. Furthermore, for our current experiments, we focus on subjects which are written in C, since our repair infra-structure works on C programs. Last but not the least, the goal of our experiments is not to measure the effectiveness of automated repair; instead our goal is to measure the effectiveness of curbing over-fitting in automated repair. Hence we do not consider those defects for which the correct patch cannot be generated by the supported program transformations in program repair.

Eventually, we select six well-known open source projects: Proj.4, FFmpeg, Libarchive, Openjpeg, Libssh and Libchewing. Brief descriptions of those projects are given in Table 2. Column #Test denotes the number of tests from developers accompanying each software project in the OSS-Fuzz repository. For each project, we select a set of reproducible defects based on the above criteria. Column #Defect shows the number of selected defects for each project. Totally, 81 unique defects are selected as our subjects. Besides, the bug type of the selected defects is various. Table 3 shows the number of defect for each bug type. Specifically, 49 defects are caused by integer overflow or buffer overflow, 7 of them are caused by invalid access, and 25 by arithmetic error or other bugs (e.g. memory leak).

7.2 Experimental Setup

To answer RQ1, we compare Fix2Fit with AFL and AFLGo based approaches in generating tests to rule out overfitted patches. AFL (AFLGo) based approach constructs candidate patch space using same operators as Fix2Fit, but rules out patches using tests generated by AFL (AFLGo). We choose AFL as our baseline, since it is a fuzz testing which is widely used in industry and academia. AFLGo, a directed greybox fuzzer, can be used for patch testing.

Deciding whether a patch is over-fitted using whether the patched program fails on tests is imprecise [12]. Opad [15] proposes a new over-fitting measure (O-measure), which is built based on the assumption that a correctly patched program should not behave worse than the buggy program. Given a test suite T,

\[ \bar{B}: \text{the set of test cases that make the buggy version pass} \]
\[ P: \text{the set of test cases that make the patched version fail} \]

O-measure is defined as the size of \( \bar{B} \cap P \). Opad determines a patch is over-fitted if it has a non-zero O-measure. In our experiment, we utilize a similar metric, but we change the definition of \( \bar{B} \). We define \( \bar{B} \bar{B} \) as the set of test cases that (i) either make the buggy version pass, or (ii) make buggy version crash due to ‘same’ defect as the one we try to fix (by comparing stack trace). The intuition is as follows: if the patched program still crashes due to same defect, we regard the corresponding patch as over-fitted patch.

To address RQ2, we compare the number of crash-free patches generated by Fix2Fit, AFL and AFLGo-based approach. In our experiment, cross-validation is used to evaluate the crash-free property, where the remaining patches after the filtering of one approach is validated by the tests generated by other techniques. Specifically, suppose \((T, P)\) is a pair of test set and plausible patch set, where the patched program using any patch \(p \in P\) does not crash under any test \(t \in T\). Let \((T_1, P_1), (T_2, P_2)\) and \((T_3, P_3)\) be the test-patch pairs generated by Fix2Fit, AFL and AFLGo, respectively. We regard \(p \in P_i\) as crash-free patch, if and only if the patched program by \(p\) does not crash under any test \(t \in T_1 \cup T_2 \cup T_3\). Then, we evaluate the percentage of crash-free patches of different techniques.

We answer RQ3 by evaluating how many plausible patches can be further ruled out if the newly generated tests are empowered with a few oracles. For any test case which is able to break one partition into several sub-partitions, it finds semantic discrepancies between patches. However, the sub-partitions cannot be ruled out if the patched programs do not crash, even though they show different behaviors. If the program is deterministic, it is highly possible that the patches from only one subpartition are correct. We can thus study the reduction in the pool of candidate patches if detailed oracles (such as expected output) for a few (say 5) tests are available. Assuming better oracle of test is given and each subpartition has equal probability to be filtered out, we evaluate the number of patches that can be ruled out (Fig. 7).

<table>
<thead>
<tr>
<th>Subject</th>
<th>#Defect</th>
<th>#Test</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proj.4</td>
<td>10</td>
<td>3</td>
<td>cartographic projection and geometric transformation library</td>
</tr>
<tr>
<td>FFmpeg</td>
<td>26</td>
<td>11</td>
<td>audio &amp; video processing library</td>
</tr>
<tr>
<td>Libarchive</td>
<td>12</td>
<td>4</td>
<td>multi-format archive library</td>
</tr>
<tr>
<td>Openjpeg</td>
<td>12</td>
<td>13</td>
<td>open-source library to encode and decode JPEG 2000 images</td>
</tr>
<tr>
<td>Libssh</td>
<td>8</td>
<td>23</td>
<td>C library for the SSHv2 protocol</td>
</tr>
<tr>
<td>Libchewing</td>
<td>13</td>
<td>11</td>
<td>phonetic input method library</td>
</tr>
<tr>
<td>Total</td>
<td>81</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 2: Subject Programs
7.3 Results

RQ1: Effectiveness in ruling out plausible patches

Figure 5 shows the percentage of plausible patch that is ruled out by AFL, AFLGo and Fix2Fit within 8 and 24 hours, where the percentage of filtered patch within the first 8 hours is marked using diagonal stripes. Note that the AFL-based approach is almost same as Opad [15], except that we utilize a more precise over-fitting measure. For each project, we give the average number of all defects. Compared with AFL and AFLGo, our technique is able to rule out more plausible patches for all those six subjects within both 8 and 24 hours. For instance, Fix2Fit is able to filter out 61% plausible patches for FFmpeg, while only 52% of them can be ruled out by AFL and 53% by AFLGo within 24 hours. To investigate the reason why Fix2Fit is able to rule out more patches, we give the number of tests generated by each technique that can filter out plausible patches in Table 4. On average, Fix2Fit generates 23% more tests that can rule out patches than AFL, and 18% more than AFLGo.

Although our approach mostly outperforms existing approaches in terms of filtering patches, there are scenarios where our technique does not perform better. According to our manual investigation, there are two situations where our technique does not outperform existing approaches (1) most of the generated tests cannot find discrepancies between patches and their separability are 0, so that our approach of preferring inputs with non-zero separability is rarely used (2) existing approaches already reduce the plausible patches to a very small number (for some defects in libchewing), so there is not much room to improve.

To filter out over-fitted patches, fuzzing in Fix2Fit is guided to generate tests that can uncover semantic discrepancies between plausible patches. Therefore, we also evaluate the patch partition refinement effectiveness of AFL, AFLGo and Fix2Fit. Figure 6 shows the number of generated tests that can refine partitions and number of patch partitions after refinement. Origin is the number of test-equivalence patch partitions with respect to the provided test suite. The histogram represents the number of partitions after refinement, which corresponds to the primary axis (left), while the line chart shows the number of partition-refining tests, which corresponds to the secondary axis (right). Fix2Fit performs better than AFL and AFLGo in both generating partition-refining tests and refined partitions. On average, Fix2Fit breaks 34% and 30% more partitions than AFL and AFLGo, respectively.

Table 3: The statistic of defect type

<table>
<thead>
<tr>
<th>Defect Type</th>
<th>#Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer overflow</td>
<td>29</td>
</tr>
<tr>
<td>Buffer overflow</td>
<td>20</td>
</tr>
<tr>
<td>Unknown address</td>
<td>4</td>
</tr>
<tr>
<td>Invalid array access</td>
<td>3</td>
</tr>
<tr>
<td>Arithmetic error</td>
<td>4</td>
</tr>
<tr>
<td>Others</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 4: The number of generated test cases that can rule out plausible patches

<table>
<thead>
<tr>
<th>Projects</th>
<th>AFL(Opad)</th>
<th>AFLGo</th>
<th>Fix2Fit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proj.4</td>
<td>4.8</td>
<td>5.9</td>
<td>12.5</td>
</tr>
<tr>
<td>Libarchive</td>
<td>11.2</td>
<td>12.8</td>
<td>16.0</td>
</tr>
<tr>
<td>FFmpeg</td>
<td>9.8</td>
<td>10.2</td>
<td>13.8</td>
</tr>
<tr>
<td>Openjpeg</td>
<td>35.3</td>
<td>35.8</td>
<td>50.3</td>
</tr>
<tr>
<td>Libssh</td>
<td>5.1</td>
<td>7.9</td>
<td>8.6</td>
</tr>
<tr>
<td>Libchewing</td>
<td>10.7</td>
<td>11.5</td>
<td>11.5</td>
</tr>
</tbody>
</table>

Figure 5: Percentage of plausible patches that are ruled out

Although we argue that partition refinement is a better heuristic than patch filtering for the purpose of guiding fuzzing, we also evaluate heuristics based on patch filtering. For patch filtering based heuristic, we change the definition of separability in Equation 4 to

\[
\text{separability}(t', T_{new}) = \frac{r(t')}{\max_{t \in T_{new}} r(t)}
\]

where \(r(t')\) represents the number of over-fitted patches that are ruled out by test \(t'\). Table 5 shows the percentage of patches that are ruled out using the heuristic based on patch filtering (PF) and partition refinement (PR). The results show that PR outperforms PF on five subjects and performs equally on one subject.

<table>
<thead>
<tr>
<th>Projects</th>
<th>Proj.4</th>
<th>Libarchive</th>
<th>FFmpeg</th>
<th>Openjpeg</th>
<th>Libssh</th>
<th>Libchewing</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF</td>
<td>68%</td>
<td>27%</td>
<td>56%</td>
<td>55%</td>
<td>51%</td>
<td>92%</td>
</tr>
<tr>
<td>PR</td>
<td>71%</td>
<td>28%</td>
<td>61%</td>
<td>56%</td>
<td>51%</td>
<td>95%</td>
</tr>
</tbody>
</table>

Fix2Fit is able to rule out 18% and 12% more over-fitted patches than AFL and AFLGo based approaches.

RQ2: Crash-free patches

To fix a bug, new bugs or security vulnerabilities should not be introduced. If one generated test makes the patched program crash, a patch will be directly ruled out. However, since fuzzing does not exhaustively generate all possible tests, the remaining patches may still cause program crash or introduce new software crashes and vulnerabilities. In this experiment, we evaluate the crash-freedom of patches generated via cross-validation. Based on cross-validation, a crash-free patch should not make program crash under any test.

---

The ability of test cases to filter out over-fitted patches is limited by the non-availability of oracles (expected output) of the generated tests. We also evaluate whether the automatically generated test case can further reduce plausible patches if empowered with better oracles (for at least a few of the generated tests).

Figure 7 shows how the number of patch candidates reduces as the number of tests empowered with oracles. For a test which can break a patch partition into several sub-partitions, we assume only one of the sub-partitions is correct if the correct behavior of this test is given. This is because the patch partitions rely on a value-based test equivalence; it is highly possible that only one of the sub-partitions will produce an output value same as the expected output captured by a given oracle. We select the top-10 tests with highest separability (heuristic based on partition refinement), and collect the number of patches if one, two,...ten oracles are given. Generally, the plausible patches for most of the defects can be reduced to a reasonable number. For defects in Openjpeg, the number of plausible patches can be reduced to around 20. In other words, if the oracles of a few of the generated tests is available, the pool of candidate patches can be reduced sufficiently so that the remaining patches can be examined manually by the developers.

For the defects which are left with large number of remaining plausible patches after patch-aware greybox fuzzing, we are faced with the task of examining these remaining plausible patches. Fortunately, developers do not need to examine the remaining patches one by one. They can examine the patches in the same patch partition together, since they show same behaviors over all the available tests. Table 7 shows the average number of remaining patches after the partition refinement by Fix2Fit. The number of remaining partitions, and hence the number of patches to examine, varies between 1-100 in each project. We feel that there might be opportunities for visualization techniques to choose from these remaining 1 − 100 patch partitions, using criteria such as syntactic or semantic “distance” from the buggy program. Such support for patch choices can further reduce over-fitting in program repair, and can be studied in the future.

The plausible patches can be reduced to a reasonable number if few tests (<10) are empowered with better oracles.

### 7.4 Threats to Validity

Our current experiments have been conducted for one-line fixes. While extension of the approach to multi-line fixes is entirely feasible, it can blow up the search space. While we have compared with Opad [15], we could not directly compare with [18, 19] which improve patch quality by test generation; the tools for those approaches are geared to repair Java programs while our repair infrastructure operates on C programs. Finally, our reported results are obtained from the OSS-Fuzz subjects in Table 2, and more experiments could be conducted on larger set of subject programs.
8 CONCLUSION

Automated program repair, specifically test-suite driven program repair, has gained traction in recent years. This includes a recent use of test-driven automated repair at scale in Facebook [37], reporting positive developer feedback. However, the automatically generated patches can over-fit the test suite T driving the repair, and their behavior on tests outside T is unknown. In this paper, we have taken a step towards tackling this problem by filtering crash introducing patch candidates via fuzz testing. Our solution integrates fuzzing and automated repair tightly by modifying a fuzzer to prioritize tests which can rule out large segments of the patch space, represented conveniently as patch partitions. Results from the continuous fuzzing service OSS-Fuzz from Google show significant promise. By systematically prioritising crash-avoiding patches in the patch search space, we take a step to tackle the over-fitting problem in program repair.

ACKNOWLEDGMENTS

This work was supported in part by Office of Naval Research grant ONRG-NICOP-N62909-18-1-2052. This work was partially supported by the National Satellite of Excellence in Trustworthy Software Systems, funded by NRF Singapore under National Cyber-security R&D (NCR) programme.

REFERENCES