Crash-Avoiding Program Repair

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

Sergey Mechtaev
University College London, UK
mechtaev@gmail.com

Abhik Roychoudhury
National University of Singapore
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.

CCS CONCEPTS

• **Software and its engineering** → **Automatic programming**; **Software testing and debugging**.

KEYWORDS

Automated program repair, Overfitting, Fuzzing

ACM Reference Format:
Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-Avoiding Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330558

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330558

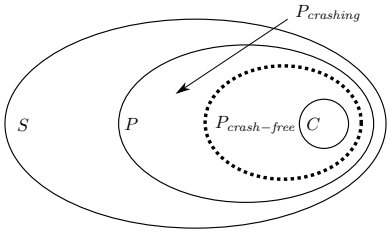

Figure 1: Structure of program repair search space, where S is a space of candidate patches, P is a set of plausible patches, $P_{crashfree}$ is a set of crash-free patches, C is a set of correct patches, $P_{crashing} = P \setminus P_{crashfree}$ is a set of crashing patches.

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 $P_{crashfree}$ (crash-free plausible patches), and $P_{crashing}$ (crashing plausible patches) and suggest that program repair should aim to find a patch from the set $P_{crashfree}$, 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.

A prominent group of testing techniques that were successfully used to find serious vulnerabilities in popular software is coverage-based greybox fuzzing [1, 2]. 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 $P_{crashfree}$. 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 $P_{crashfree}$ and $P_{crashing}$ (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 $P_{crashfree}$ and $P_{crashing}$.

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. 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. Fix2Fit is available open-source from https://www.github.com/gaoxiang9430/fix2fit.

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: (1)*search-based approaches* search the correct patch from a huge patch space using meta-heuristic [3, 4], random search [5] or test-equivalence analysis [6] (2)*constraint solving based approaches* extract constraints from test executions, and synthesize patches by solving the constraints [7–10], and (3) potentially *learning-based approaches* use a model to select patches that are more likely to fix the defect based on existing patches [4, 11] or program context [12]. 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 [13]. To alleviate the over-fitting problem, existing approaches filter out *overfitted* patches by defining anti-pattern [14] or generating smallest program repairs [9]. Our work on 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 behavioral differences 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, 2], cover the changes in patch [21] or find behavioral asymmetries between programs (differential testing) [22]. Symbolic execution employs constraint collection and solving to systematically and effectively explore the state space of feasible execution paths [23], and can be used for directed testing [24–27]. 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 information collected at run-time with the help of compile-time instrumentation. Greybox fuzzing has

```
1  int decode_dds1(ByteContext *gb, uint8_t *frame,
       int width, int height){
2    ...
3    segments = bytestream2_get_le16(gb);
4    while(segments--) {
5      if (bitbuf & mask) {
6        ...
7      }
8      else if (bitbuf & (mask << 1)) {
9        v = bytestream2_get_le16(gb)*2;
10       if (frame - frame_end < v)
11         return AVERROR_INVALIDDATA;
12       frame += v;
13     } else {
14       int remaining_space = frame_end-frame;
15       if(remaining_space < width+3)
16       //"width+3" → "width+4" (correct patch)
17         return AVERROR_INVALIDDATA;
18       frame[0] = frame[1] = frame[width] =
19       frame[width+1] = bytestream2_get_byte(gb);
20       frame += 2;
21       frame[0] = frame[1] = frame[width] =
22       //buffer overflow location
23       frame[width+1] = bytestream2_get_byte(gb);
24       frame += 2;
25     }
26   }}
```

Listing 1: Buffer overflow vulnerability in *FFmpeg*

been demonstrated to be useful for increasing code coverage [1, 28], reaching target location [21], and finding behavioral asymmetries between programs [22]. Different from those techniques, Fix2Fit takes the semantic of patches into consideration and it is designed with the goal of finding semantic discrepancies between patches.

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-Fuzz[1] in May, 2017. This vulnerability is caused by incorrect bounds checking when *FFmpeg* decodes DirectDraw Surface (DDS) files[2]. 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 patch[3] 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 will 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

[1]https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=1345
[2]DDS is an image file format for storing texture and environments
[3]https://github.com/FFmpeg/FFmpeg/commit/f52fbf

Table 1: Plausible patches and their behaviors on new test

Id	plausible patch	T_1	T_2	T_3	T_4
1	remaining_space>width+1	(T) ✓	(F) ✗	–	–
2	remaining_space>width+2	(F) ✗	–	–	–
3	remaining_space! =width+3	(T) ✓	(T) ✓	(T) ✓	(T) ✓
4	remaining_space<=width+3	(T) ✓	(T) ✓	(F) ✓	(F) ✓
5	remaining_space>=width+3	(F) ✗	–	–	–
6	remaining_space<width+4	(T) ✓	(T) ✓	(F) ✓	(F) ✓
7	remaining_space<width+5	(T) ✓	(T) ✓	(T) ✓	(F) ✓
8	remaining_space<width+6	(T) ✓	(T) ✓	(T) ✓	(F) ✓

T_1: remaining_space=width+2 T_2: remaining_space=width
T_3: remaining_space=width+4 T_4: remaining_space=width+6

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 over-fitted patches instead of improving code coverage. In this example, we expect tests that can drive the execution to the patch location with different program states (values of *remaining_space*, *width*).

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 tests 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 T_1, and program fixed by this patch still crashes (✗) under T_1, so that patch 2 is filtered out and will not be considered in the following iterations. Test input T_1 is able to find the discrepancies between patches, and rule out two over-fitted patches. Correspondingly, T_2 and T_3, which 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 code coverage based heuristic used in AFL, we propose a new heuristic: we favor tests with greater ability to distinguish plausible patches. In this example, AFL will not retain T_1, since it does not improve code coverage. However, our proposed patch-aware fuzzing will retain T_1 for further mutation, so we have a chance of finding tests like T_2 or T_3 via mutation. In addition, the chance of generating tests to 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 from developers), all the incorrect patches can be ruled out.

4 BACKGROUND

We denote a program as p and a program obtained from p by substituting an expression e with e' as $p[e \mapsto e']$. The substitution $(e \mapsto e')$ of expressions is called *patch* of p, and sets of *patches* are denoted as $P, P_1, ..., P_n$. The letters $t, t_1, ..., t_n$ represent program inputs (tests), and $T, T_1, ..., T_n$ represent sets of program inputs (test suites).

4.1 Program Repair

Automated program repair techniques take in a buggy program, and a set of passing and failing tests, and aim to generate a patched program that passes all the given tests. We consider the search spaces of candidate patches that consist of only modifications of program expressions. The search space in our approach is defined by the following transformation schemas:

- Change an existing assignment:

$$\texttt{x := e;} \quad \mapsto \quad \texttt{x := e'};$$

- Change an existing if-condition:

$$\texttt{if (e) \{...\}} \quad \mapsto \quad \texttt{if (e') \{...\}}$$

- Add an if-guard to an existing statement S:

$$\texttt{S;} \quad \mapsto \quad \texttt{if (e) S;}$$

where e and e' are arbitrary expressions 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 [29] 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 [6, 30, 31] can 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* [6]. 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

```
Input: seed inputs T
1 while timeout is not reached do
2     t := chooseNext(T);
3     energy := assignEnergy(t);
4     for i from 1 to energy do
5         t' := mutate(t);
6         if isInteresting(t') then
7             T := T ∪ t';
8     end
9 end
```

AFLGo [21], 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 direct the 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* [32], which dictates how the temperature decreases over time. Based on *cooling schedule*, the current temperature T_{exp} is defined as:

$$T_{exp} = 20^{-\frac{ctime}{time_x}} \tag{1}$$

where $time_x$ is user-defined time to enter "exploitation" (preferring tests deemed closer to the target) from exploration, $ctime$ is current execution time. Given the current temperature T_{exp}, normalized

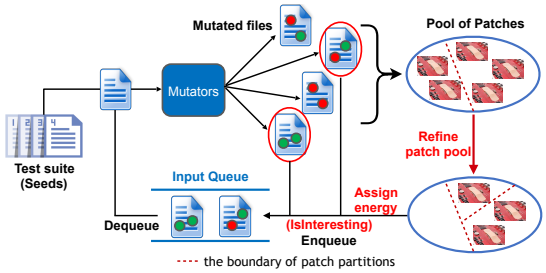

Figure 2: Structure of integrated testing and repair loop

distance $d(t, T_b)$ between test t and target location T_b, AFLGo introduces an annealing-based power schedule (APS):

$$aps(t) = (1 - d(t, T_b)) * (1 - T_{exp}) + 0.5T_{exp} \quad (2)$$

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

$$energy_{aflgo}(t) = energy_{afl}(t) * 2^{10*aps(t)-5} \quad (3)$$

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 $p[e \mapsto e_i]$. $\forall p_i, p_j \in P$, p_i and p_j belong to same equivalent patch partition if and only if $\forall t \in T$. $p[e \mapsto e_i]$ is *value-based test-equivalence* to $p[e \mapsto e_j]$ w.r.t t.

The ability of a test to find semantic discrepancies is formalized as its effectiveness in refining patch partitions. For any two patches p_i, p_j from the same equivalence partition EP, if $p[e \mapsto e_i]$ is not *value-based test-equivalence* to $p[e \mapsto e_j]$ w.r.t new test t, we say test t refines partition EP. Different from existing fuzz testing techniques that maximize the code coverage (AFL), or minimize distance to the target location (AFLGo), Fix2Fit is designed to maximize semantic discrepancies across patches (thereby refining patch partitions). To find more semantic discrepancies between plausible patches, we essentially generate test cases that can make the execution reach the patch location with divergent program states.

5.1 Integration of Test Generation and Repair

Figure 2 presents a visual summary of our integrated testing and repair loop. In directed grey-box fuzzers such as AFLGo [21], the generation of new tests are guided by distance to the target gathered at run-time with the help of compile-time instrumentation. In our fuzzer, the fuzzing is guided not only by distance feedback but also by *separability*, the ability of a test to distinguish patches. In this way, we prioritize tests which can distinguish existing patches and as a result rule out more over-fitted patches.

ALGORITHM 2: Patch-aware Greybox Fuzzing

```
Input: test suite T_in, program p
1  Par := genPlausiblePatches (p, T);     // generate set of patch partitions
2  pLocs := extractPatchLocs (Par);       // extract set of patch locations
3  p' := instrument (p, pLocs);           // instrument fuzzing targets
4  T_new := {};
5  T := T_in;
6  while true do
7      t := chooseNext(T);
8      for i from 1 to t.energy do
9          t' := mutate(t); T_new := T_new ∪ {t'};
10         (isReached, distance, coverage) := exec(p', t');
11         if isReached then
12             Par := refine_and_filter(Par, t');
               // Break patch partitions & remove over-fitted partitions
13             sep := separability(t', T_new) // Equation 4
14         end
15         t'.energy := powerSchedule(sep, distance, coverage); // Sec 5.3
16         if isInteresting(coverage, sep) then
               // Sec 5.4
17             T := T ∪ {t'};
18         end
19     end
20     if timeout || sizeOf(Par) == 0 then
21         break;
22 end
23 Output: remaining patch partitions Par
```

Algorithm 2 shows the key steps of Fix2Fit. The main procedure is built on top of an automated patching technique, and directed greybox fuzzing technique. Given a buggy program p, a test-suite T, and at least one test case in T that can trigger a bug, this algorithm will return a set of plausible patch partitions for fixing the bug. 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 (*isReached*). 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. (1) First, *refine_and_filter* refines the current patch partitions *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'. (2) After the patch partitions

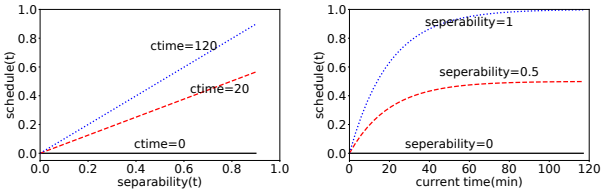

Figure 3: (a) energy of a test with different separability at 0min, 20min, 120min (b) energy of a test t at different time when separability(t)=0, 0.5, 1. $time_x$=60min

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 crashing 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 $\{P_1, P_2, ...P_n\}$, 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_{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 $t \in T_{new}$:

$$separability(t') = \frac{b(t')}{\max_{t \in T_{new}} b(t)} \tag{4}$$

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 [21], 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* [32] 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_{exp} (as defined in Equation 1) as well the *separability(t')*, our power schedule is:

$$schedule(t') = separability(t') * (1 - T_{exp}) \tag{5}$$

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

$$energy(t') = energy_{aflgo}(t') * 2^{schedule(t') * log_2 Max\_Factor} \tag{6}$$

where Max_Factor is the user-defined max factor integrated to existing energy, and $\frac{energy(t')}{energy_{aflgo}(t')} \in [1, Max\_Factor]$.

5.4 Is Interesting?

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

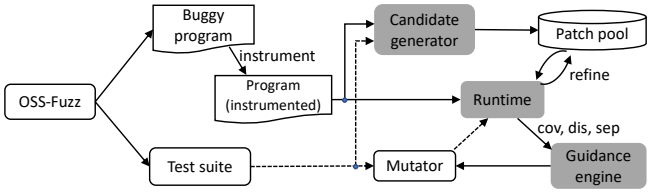

Figure 4: Architecture of tool Fix2Fit

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 fuzzer 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 fuzzer 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. Sanitizers can detect various vulnerabilities at run-time with the help of compile-time instrumentation. Generally, sanitizers convert the software vulnerabilities into normal crashes, e.g. AddressSanitizer crashes the program if a buffer overflow is detected. By using sanitizers we can rule out the patches that introduce vulnerabilities. As compared to only filtering patches based on crashes, more patches can be filted out.

The sanitizers used by Fix2Fit include UndefinedBehaviorSanitizer[4] (UBSan) and AddressSanitizer[5] (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, so that remaining patches are guaranteed not to introduce security vulnerabilities in terms of all available tests.

6 IMPLEMENTATION

The architecture of Fix2Fit is shown in Figure 4. Fix2Fit takes as inputs the buggy program and test suites extracted from OSS-Fuzz benchmark, and generates a set of crash-free patches. The initial test-suite is composed of available developer test cases and the failing tests generated OSS-fuzz. 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 test on the instrumented program and collects necessary information (e.g. code coverage and separability). Accordingly, the *Patch pool* is refined after executing each test. *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 run-time information. To collect the distance to patch locations, we inherit the instrumentation strategy used in AFLGo [21], 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.

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 [33], Angelix [10] or F1X [6]. 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 $proc_{allpatch}$ enumerating all plausible patches, and generates a meta-program by dynamically replacing the to-be-fixed expression with a call to this procedure. At runtime, the procedure $proc_{allpatch}$ is invoked when the patch location is reached. By controlling the enumeration strategy, this procedure $proc_{allpatch}$ 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 maintained in the *patch pool*, as in the F1X repair tool [6]; different from F1X, patch partitions are used to guide test generation with the objective of ruling out patches.

Runtime* and *Guidance engine The main procedure of fuzzing is built on top of the directed greybox fuzzer AFLGo [21]. Besides the heuristic used in AFLGo, the *Guidance engine* also takes *separability* (Equation 4) into account.

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?

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

[4]UBSan website: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
[5]ASan website: https://clang.llvm.org/docs/AddressSanitizer.html

Table 2: Defect categories

Defect Type	Integer overflow	Buffer overflow	Unknown address	Invalid array access	Arithmetic error	Others
#Defects	29	20	4	3	4	21

Table 3: Subject programs

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

logic errors e.g. ManyBugs [34] and Defects4j [35]. Instead, we select a set of real-world subjects from the OSS-Fuzz (Continuous Fuzzing for Open Source Software) dataset [6]. 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. We drop the defects that cannot be reproduced. Furthermore, we focus on subjects which are written in C, since our repair infra-structure works on C programs.

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 3. 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 2 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[7] and AFLGo[8] 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 [13]. *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,

$\overline{B}$: the set of test cases that make the buggy version pass($\overline{B}$⊂T)
P: the set of test cases that make the patched version fail(P⊂T)

O-measure is defined as the size of $\overline{B} \cap$ P. *Opad* determine 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 $\overline{B}$. We define $\overline{B}$ 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_3) 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. 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).

All the experiments are conducted in the *crash exploration mode*[9] of fuzzer. We start the fuzzing process with the failing test case as seed corpus, and terminate it on timeout. As in state-of-the-art fuzzing experimentation, we set timeout as 24 hours; at the same time we report the effectiveness of our patch pool reduction for smaller values of timeout such as 8 hours. Meanwhile, we set time ($time_x$ in Equation 1) to enter "exploitation" as four hours. The experiments are conducted on a device with an Intel Xeon CPU E5-2660 2.00GHz process (56 cores) 64G memory and 16.04 Ubuntu.

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

[6] https://bugs.chromium.org/p/oss-fuzz/issues/list
[7] http://lcamtuf.coredump.cx/afl/
[8] https://github.com/aflgo/aflgo
[9] https://lcamtuf.blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html

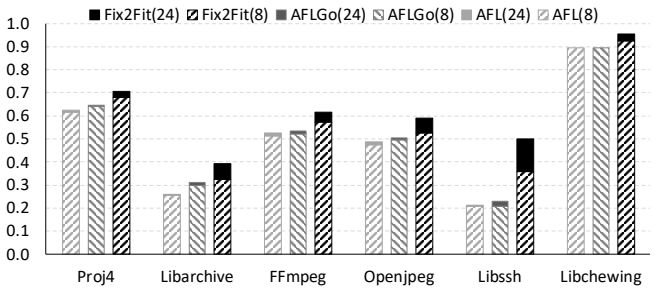

Figure 5: Percentage of plausible patches that are ruled out

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*, Fix2Fit rules out more plausible patches for all those six subjects within both 8 and 24 hours. For instance, Fix2Fit filters out 61% plausible patches for *FFmpeg*, while only 52% of them are ruled out by AFL and 53% by AFLGo within 24 hours. Since fuzzing algorithms involve random decision, we run each experiment ten times independently and report the Vargha-Delaney statistic measure ($\hat{A}_{12}$) [36] in Table 4. Vargha-Delaney statistic is a recommended standard measure for evaluating randomized algorithms [37], which measures the probability that running Fix2Fit rules out more patches than running AFL. Fix2Fit performs better than AFL when $\hat{A}_{12}$ is greater than 0.5. The evaluation results show that Fix2Fit outperforms AFL on all six subjects.

Table 4: The averaged $\hat{A}$ of each project with ten runs.

Projects	Proj.4	Libarchive	FFmpeg	Openjpeg	Libssh	Libchewing
$\hat{A}_{12}$	0.70	0.79	0.74	0.68	0.61	0.54

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 5. On average, Fix2Fit generates 23% more tests that can rule out patches than AFL, and 18% more than AFLGo.

Table 5: The number of generated test cases that can rule out plausible patches

Projects	AFL(*Opad*)	AFLGo	Fix2Fit
Proj.4	4.8	5.9	12.5
Libarchive	11.2	12.8	16.0
FFmpeg	9.8	10.2	13.8
Openjpeg	35.3	35.8	50.3
Libssh	5.1	7.9	8.6
Libchewing	10.7	11.5	11.5

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.

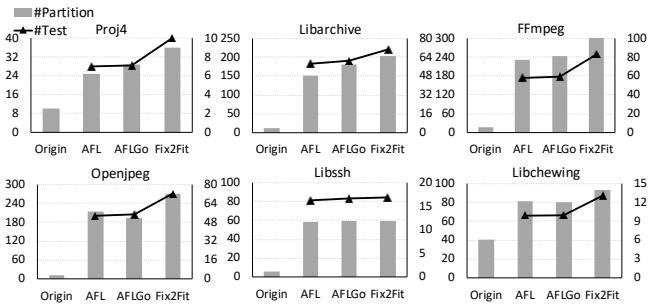

Figure 6: Number of patch partitions and the number of generated tests that can break patch partitions

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.

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

$$separability(t') = \frac{r(t')}{\max_{t \in T_{new}} r(t)} \quad (7)$$

where $r(t')$ represents the number of crashing (hence over-fitted) patches that are ruled out by test t'. Table 6 shows the percentage of patches that are ruled out using the heuristic based on *patch filtering* (*PF*) and *partition refinement* (*PR*). The results show *PR* outperforms *PF* on five subjects and performs equally on one subject.

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

Table 6: % of plausible patches ruled out using *partition refinement (PR)* and *patch filtering (PF)* based heuristic

Projects	Proj.4	Libarchive	FFmpeg	Openjpeg	Libssh	Libchewing
PF	68%	27%	56%	55%	51%	92%
PR	71%	28%	61%	56%	51%	95%

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 cases generated by any techniques. Table 7 shows the percentage of crash-free patches generated by AFL, AFLGo, Fix2Fit. Compared with AFL and AFLGo, our technique significantly improves the percentage of crash-free patches. On average, Fix2Fit generates 96.3% crash-free patches,

while 85.4% and 87% patches generated by AFL and AFLGo are crash-free. Especially for *Proj.4*, over 99.5% patches generated by Fix2Fit is crash-free, compared with 92% of AFL and 90% of AFLGo.

Table 7: The percentage of crash-free patches generated by AFL, AFLGo, Fix2Fit

Subject	AFL(*Opad*)	AFLGo	Fix2Fit
Proj.4	92%	90%	99%
Libarchive	88%	96%	97%
FFmpeg	84%	86%	95%
Openjpeg	82%	85%	91%
Libssh	83%	83%	99%
Libchewing	94%	94%	99%

Although most of patches generated by Fix2Fit are crash-free, there are some patches (3.7%) which cause program to crash under the tests generated by AFL or AFLGo. Fix2Fit may miss some corner cases since it enters the "exploitation" mode after sufficient "exploration", while AFL and AFLGo keep broadly searching.

Fix2Fit could significantly improve the percentage of crash-free patches, and more that 96% patches are crash-free.

RQ3: Improvement with better oracles. The ability of test cases to filter out over-fitted patches is limited by the non-availability of oracles (or 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 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 produces an output value same as the expected output. 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 tests are available, the pool of candidate patches can be reduced sufficiently so that the remaining patches can be examined manually by the developers.

Table 8: Number of remaining partitions after refinement

Projects	Proj.4	Libarchive	FFmpeg	Openjpeg	Libssh	Libchewing
#Partition	4.8	74.4	98.9	47.3	28.3	1.3

For the defects which are left with large number of plausible patches, 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 8 shows the average number of remaining patch partitions after the *partition refinement* by Fix2Fit.

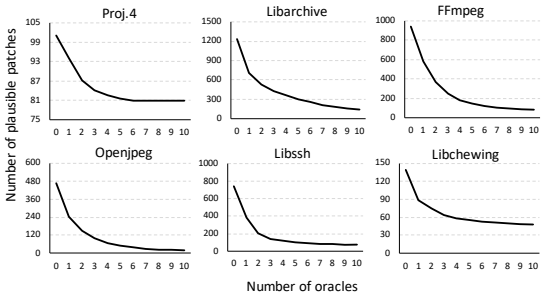

Figure 7: Number of plausible patches that can be reduced if the tests are empowered with more oracles

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.

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 3, 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 [38], 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 Cybersecurity R& D (NCR) programme.

REFERENCES

[1] Website. American fuzzy lop, http://lcamtuf.coredump.cx/afl/. Accessed: 2018-12-18.

[2] Website. libfuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2018-12-21.

[3] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, page 54, 2012.

[4] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[5] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[6] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. Test-equivalence analysis for automatic patch generation. *ACM Transactions on Software Engineering and Methodology*, 27(4):15, 2018.

[7] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference onSoftware Engineering*, pages 772–781. IEEE, 2013.

[8] J. Xuan, M. Martinez, F. Demarco, M. Clement, S.L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43, 2017.

[9] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering*, pages 448–458. IEEE, 2015.

[10] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 691–701. ACM, 2016.

[11] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE, 2013.

[12] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *International Conference on Software Engineering*. ACM, 2018.

[13] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[14] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering*, pages 727–738. ACM, 2016.

[15] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841. ACM, 2017.

[16] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empirical Software Engineering*, pages 1–35, 2018.

[17] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 416–419. ACM, 2011.

[18] Qi Xin and Steven P Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 226–236. ACM, 2017.

[19] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*, pages 789–799. ACM, 2018.

[20] David Shriver, Sebastian Elbaum, and Kathryn T Stolee. At the end of synthesis: narrowing program candidates. In *IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track*, pages 19–22. IEEE, 2017.

[21] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344. ACM, 2017.

[22] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy*, pages 615–632. IEEE, 2017.

[23] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, volume 8, pages 209–224, 2008.

[24] Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 235–245. ACM, 2013.

[25] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227. IEEE, 2008.

[26] Dawei Qi, Abhik Roychoudhury, and Zhenkai Liang. Test generation to expose changes in evolving programs. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 397–406. ACM, 2010.

[27] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[28] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[29] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[30] René Just, Michael D Ernst, and Gordon Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 315–326. ACM, 2014.

[31] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, 2014.

[32] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[33] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. *ACM SIGPLAN Notices*, 51(1):298–312, 2016.

[34] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, December 2015.

[35] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[36] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[37] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[38] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. Sapfix: Automated end-to-end repair at scale. In *ACM/IEEE International Conference on Software Engineering, Track Software Engineering in Practice*, 2019.