Concolic Program Repair

Ridwan Shariffdeen[*]
National University of Singapore
Singapore
ridwan@comp.nus.edu.sg

Yannic Noller[*]
National University of Singapore
Singapore
yannic.noller@acm.org

Lars Grunske
Humboldt-Universität zu Berlin
Germany
grunske@informatik.hu-berlin.de

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

Abstract

Automated program repair reduces the manual effort in fixing program errors. However, existing repair techniques modify a buggy program such that it passes given tests. Such repair techniques do not discriminate between correct patches and patches that overfit the available tests (breaking untested but desired functionality). We propose an integrated approach for detecting and discarding overfitting patches via systematic co-exploration of the patch space and input space. We leverage concolic path exploration to systematically traverse the input space (and generate inputs), while ruling out significant parts of the patch space. Given a long enough time budget, this approach allows a significant reduction in the pool of patch candidates, as shown by our experiments. We implemented our technique in the form of a tool called 'CPR' and evaluated its efficacy in reducing the patch space by discarding overfitting patches from a pool of plausible patches. We evaluated our approach for fixing real-world software vulnerabilities and defects, for fixing functionality errors in programs drawn from SV-COMP benchmarks used in software verification, as well as for test-suite guided repair. In our experiments, we observed a patch space reduction due to our concolic exploration of up to 74% for fixing software vulnerabilities and up to 63% for SV-COMP programs.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Keywords: program repair, concolic execution, program synthesis, patch overfitting

[*]Joint first authors

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM Reference Format:
Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic Program Repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454051

1 Introduction

Automated Program Repair [14, 24] is an emerging technology which seeks to rectify errors or vulnerabilities in software automatically. There are various applications of automated repair, including improving programmer productivity, reducing exposure to software security vulnerabilities, producing self-healing software systems, and even enabling intelligent tutoring systems for teaching programming.

Since program repair needs to be guided by certain notions of correctness and formal specifications of the program's behavior are usually not available, it is common to use test-suites to guide repair. The goal of automated repair is then to produce a (minimal) modification of the program so as to pass the tests in the given test-suite. While test-suite driven repair provides a practical formulation of the program repair problem, it gives rise to the phenomenon of "overfitting" [26, 30]. The patched program may pass the tests in the given test-suite while failing tests outside the test-suite, thereby overfitting the test data. Such overfitting patches are called *plausible* patches because they repair the failing test case(s), but they are not guaranteed to be *correct*, since they may fail tests outside the test-suite guiding the repair. Various solutions to alleviate the patch overfitting issue have been studied to date, including symbolic specification inference [23, 25], machine learning-based prioritization of patches [2, 20, 21] and fuzzing based test-suite augmentation [7]. These works do not guarantee any notion of correctness of the patches, and cannot guarantee even the most basic correctness criteria such as crash freedom.

In this work, we reflect on the problem of *patch overfitting* [22, 26, 30], in our attempt to produce patches which work for a *large number* of test inputs. Our goal is to devise an anytime patching algorithm; the algorithm can be stopped at any time. However, the longer it is run, the greater is the coverage of the input space, and the greater is our confidence that the patch produced works for a large class of test inputs. To ensure coverage of the test input space, we use concolic path exploration for automated test generation. Use of symbolic and concolic execution for test generation is well-known [4, 9]; symbolic execution has also been used in automated repair for computing repair constraints [25]. At the same time, our usage of concolic execution is innovative, and is the key technical contribution of this paper.

We use concolic execution [9] to generate test inputs, and additionally to generate constraints for the patch refinement, to make them work for those test inputs. We leverage a user-provided specification to detect incorrect behavior for the generated test inputs. Such specification does not need to be a full specification with regard to the program's correctness. Partial specifications like an assertion at a specific location, or the absence of crashes in a specific location, can be already sufficient to detect overfitting patches. Our outlook is to use concolic execution for computing path constraints and patch constraints at the same time. By making the symbolic execution technology serve such a dual purpose, we can systematically traverse a large portion of the test input space, and find out patch patterns which work for those traversed test inputs. Given a longer time budget, we obtain greater path coverage, and rule out a large number of patch candidates, thereby reducing overfitting in program repair.

Realizing such a dual-purpose usage of symbolic execution, requires us to overcome many technical challenges. First our symbolic execution engine needs to compute path constraints containing both input variables and patch variables. Though the patch variables are higher order variables, we avoid developing a second order symbolic execution engine for scalability reasons. Instead we provide a first order encoding of path constraints and patch constraints which contain (first order) input variables along with certain additional parameters to succinctly represent sets of patches. Secondly, and more importantly, there are additional sources of path infeasibility as compared to traditional concolic/symbolic execution, in our setup. In traditional concolic execution, a path is deemed infeasible if the path constraint is unsatisfiable. In our setup, the path contains a hole for the patch location, and we maintain a pool of patch candidates which diminishes as more paths are explored. Hence if none of the remaining patch candidates can be inserted into the patch location, we also deem the path as infeasible.

The benefits of our concolic approach for patch generation are shown by the experimental evaluation of its efficacy in repairing a large set of security vulnerabilities curated in recent works [8] based on Google's OSS-Fuzz infrastructure. The tool embodying our concolic program repair approach is called CPR, an abbreviation signaling the ability to resuscitate or recover programs via appropriate fixes.[1]

```
     ........
250  static int
251  cvtRaster(TIFF* tif, uint32* raster, uint32 width,
         uint32 height)
252  {
253      uint32 y;
254      tstrip_t strip = 0;
255      tsize_t cc, acc;
256      unsigned char* buf;
257      uint32 rwidth = roundup(width, horizSubSampling);
258      uint32 rheight = roundup(height, vertSubSampling);
259      uint32 nrows = (rowsperstrip > rheight ?
             rheight : rowsperstrip);
260      uint32 rnrows = roundup(nrows,vertSubSampling);
261      if (CONDITION) return 0;
262       /* potential divide-by-zero error */
263      cc = rnrows*rwidth + 2 * ((rnrows*rwidth)
             / (horizSubSampling*vertSubSampling));
         ........
278  }
```

Listing 1. CVE-2016-3623: Divide by Zero in LibTIFF v4.0.6

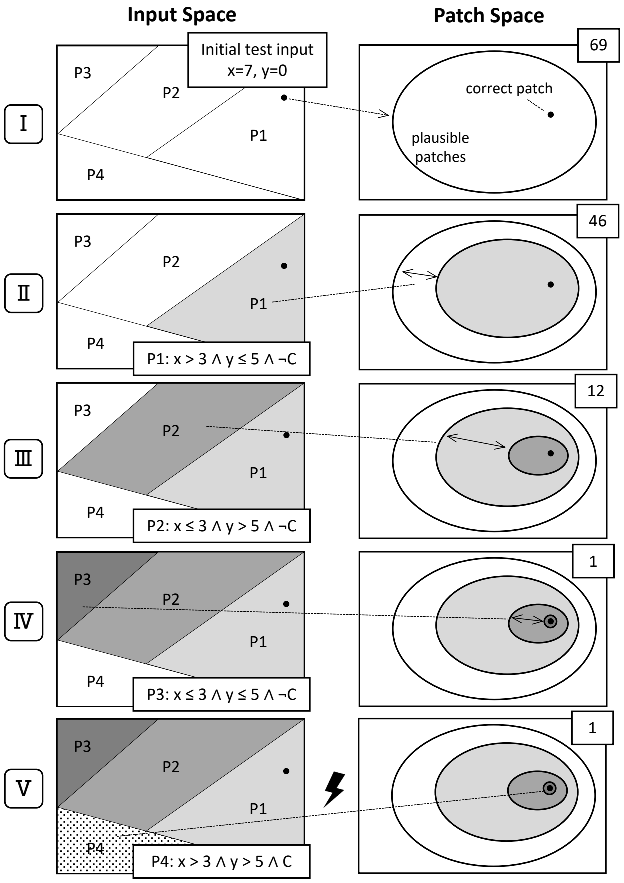
Novelty and Contributions. Overall, we provide two key novelties in program repair: (1) the concept of simultaneous exploration of input and patch space, (2) alleviate patch overfitting by checking for a user-provided specification during concolic exploration. We propose the path exploration in concolic execution as a mechanism to traverse the program input space and patch space simultaneously. The main contribution is to tackle patch overfitting, which is a key problem in the area of automated program repair [26, 30]. Our repair tool CPR generates correct patches for a variety of specifications or oracles including crash-freedom (absence of observable vulnerabilities), and satisfaction of assertions — as shown by our experiments.

2 Illustrative Example

In this section we show the advantages of *concolic program repair* by illustrating its usage for the repair of a security vulnerability in a real-world application. We make use of the security vulnerability reported as CVE-2016-3623 discovered in the LibTIFF library v4.0.6 (see Listing 1). LibTIFF is a popular open-source library that provides support for the Tag Image File Format (TIFF), a widely used format for storing image data. CVE-2016-3623 represents a divide-by-zero vulnerability, which allows a remote attacker to cause a denial of service by setting malicious inputs to the program `rgb2ycbcr`. Listing 1 depicts the relevant code snippet, which could lead to a divide-by-zero error at line 263 if the two variables `horizSubSampling` and `vertSubSampling` are not sanitized for invalid inputs. We have added a fix template in line 261, where the condition can be generated using most state-of-the-art repair tool.

Repair process. Concolic program repair works on a high-level in three phases: (1) patch pool construction, (2) path

[1] Resuscitating a program, like what Cardio-pulmonary Resuscitation (CPR) does to a patient.



Patch Details

ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	a ≥ -10 ∧ a ≤ 7	18
2	y < b	b ≥ 1 ∧ b ≤ 10	10
3	x == a || y == b	(a=7 ∧ b ≥ -10 ∧ b ≤ 10) ∨ (b=0 ∧ a ≥ -10 ∧ a ≤ 10)	41

ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	a ≥ -10 ∧ a ≤ 4	15
2	y < b	b ≥ 1 ∧ b ≤ 10	10
3	x == a || y == b	b=0 ∧ a ≥ -10 ∧ a ≤ 10	21

ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	a ≥ -10 ∧ a ≤ 0	11
~~2~~	~~y < b~~	False	0
3	x == a || y == b	a = 0 ∧ b = 0	1

ID	Patch Template	Parameter Constraint	# Conc. Patches
~~1~~	~~x >= a~~	False	0
3	x == a || y == b	a = 0 ∧ b = 0	1

ID	Patch Template	Parameter Constraint	# Conc. Patches
3	x == a || y == b	a = 0 ∧ b = 0	1

x = horizSubSampling, **y** = vertSubSampling, **C** = CONDITION

Figure 1. Illustrative concolic exploration for example CVE-2016-3623 in Listing 1 as the simultaneous exploration of the input space and the patch space. The rows I, II, III, IV, and V represent multiple exploration steps. The columns show the increasingly covered *Input Space*, the decreasing *Patch Space*, as well as more details on the identified patches. The patch space is in general limited by the synthesis language (denoted by the rectangular around the patch space illustration). The number on the top right of the patch space illustration denotes the total number of concrete patches included in this patch space.

exploration, and (3) patch reduction. The phases (2) and (3) are performed in an alternating manner: The path exploration provides input partitions (in form of path constraints), and the patch reduction refines abstract patches and rules out patches that fail the user-provided specification for the current input partition.

Illustration. Figure 1 illustrates the simultaneous space reduction (i.e., the interplay between path exploration and patch reduction): as we explore the input space, we are able to narrow down and refine the patch space (steps I, II, III, and IV), while at the same time we leverage the patch space to skip parts of the input space, which are not feasible with the available patches (step V). Therefore, each row I, II, III, IV, and V in Figure 1 represents an exploration step, which represents an increase of the *input space* coverage and a potential reduction of the *patch space*. The input space for this example is partitioned into 4 compartments *P1*, *P2*, *P3*, and *P4*, which are defined by the corresponding path constraints. Note that the constraints in Figure 1 show only the relevant parts for this example and further assume a control location, which compares the relevant variables `horizSubSampling` and `vertSubSampling` with the given constants. These path constraints are chosen artificially for this example (since details of `roundup` are not shown). As mentioned in Figure 1, we refer to `horizSubSampling` and `vertSubSampling` as x and y respectively as a notational short-hand. Our patch space is generally limited by the synthesis language (denoted by the rectangle around the patch space illustration). In order to illustrate the overall reduction in terms of concrete patches, the box in the top right corner of the patch space shows the total number of concrete patches included in this patch space. Please note that Figure 1 does not show the exploration of all possible input partitions, and hence, shows only a part of the input exploration for illustration purposes.

Patch pool construction. In this example, our approach starts with synthesizing a set of plausible patches based on an initial test case with x=7, y=0 (see step I in Figure 1). We assume that the user-defined *specification* states that there should be no divide-by-zero error at line 263 in Listing 1, i.e., that $x * y \neq 0$. The set of plausible patches is shown as the oval in the *Patch Space* column. Note that we assume that the correct patch is included in this set. The table on the right side of Figure 1 shows an illustrative list of patch templates (aka abstract patches) generated by our synthesizer. As abstract patches we consider boolean and integer expressions, which include program variables (e.g., x and y) and parameters (e.g., a and b). During the repair process, the parameter values are captured by a certain constraint (see column *Parameter Constraint*), which covers a set of concrete patches and limits the search space. The column *# Concr. Patches* shows how many concrete patches are covered by the corresponding abstract patch. For this illustrative example, we assume that the parameter values are initially in the range [-10, 10]. The constraints shown in the table are already modified by the synthesizer to pass the initial test case. In the following paragraphs, we will provide more detailed information on the interplay between path exploration and patch reduction.

Input partition P1 for patch 1. Starting with the initial input, concolic execution provides us with the input partition P1 (defined by the corresponding path constraint). Step II in Figure 1 represents the first repair iteration. For every abstract patch, we check whether a violation of the specification is feasible with the current path constraint. If yes, we try to refine the constraint on the parameter values. The light-grey shaded area in the patch space indicates the refinement to the patch space as we explore the respective path of P1. In order to refine patch 1, we search for models of:

$$\underbrace{x > 3 \land y \leq 5 \land \neg(x \geq a) \land a \in [-10, 7]}_{\text{path constraint P1 complemented with patch 1}} \land \underbrace{(x * y = 0)}_{\substack{\text{condition for}\\ \text{specification violation}}}$$

Every satisfying assignment reveals a possibility to violate the specification with the current path constraint and patch 1. In order to make this formula unsatisfiable, we need to remove the values $\{5, 6, 7\}$ from the constraint on a. Therefore, the refined variant of patch 1 is: $x \geq a$ with $a \in [-10, 4]$ (see table on the right side of row II in Figure 1). This refinement removes 3 concrete patches from the patch space.

Input partition P1 for patch 2. In order to test patch 2 on the input partition P1, we again first check whether it is possible to violate the specification with the current path constraint and patch 2. The formula to test would be: $x > 3 \land y \leq 5 \land \neg(y < b) \land b \in [1, 10] \land (x * y = 0)$. However, this formula is unsatisfiable, and hence, patch 2 cannot be refined in this step.

Input partition P1 for patch 3. For patch 3 we need to test: $x > 3 \land y \leq 5 \land \neg(x = a \lor y = b) \land (a = 7 \land b \in [-10, 10] \lor b = 0 \land a \in [-10, 10]) \land (x * y = 0)$. For this formula, only $y = 0$ is the feasible condition for a violation. Therefore, all parameter value combinations, for which $b \neq 0$ are models for a specification violation and need to be removed from the parameter constraint during refinement. The resulting parameter constraint is: $(a = 7 \land b \in [0] \lor b = 0 \land a \in [-10, 10])$, which can be simplified to $b = 0 \land a \in [-10, 10]$.

Exploration of P2 and P3. In order to generate a new input, the current path constraint of P1 can be mutated, e.g., by flipping constraints in P1 (as in concolic execution), and solved with an SMT solver. For example, we could retrieve the input x=0, y=6 corresponding to the path constraint P2: $x \leq 3 \land y > 5 \land \neg C$ (see step III in Figure 1). While exploring P2, the parameter constraint in patch 1 can be refined to $a \in [-10, 0]$. Patch 2 does violate the specification for P2 for *all* available parameter values. Therefore, patch 2 cannot be refined and needs to be removed in step III. Finally, the parameter constraint in patch 3 can be refined to $a = 0 \land b = 0$, i.e., there is only one concrete mapping left for this patch. In fact, patch 3 now is semantically equivalent to the *correct* patch. Step IV in Figure 1 shows one final step, where patch 1 can be removed and patch 3 remains as the correct patch.

Non-Exploration of P4. Step V in Figure 1 shows the consideration of P4 with the path constraint $x > 3 \land y > 5 \land C$. One of our key ingredients is, when generating a new input, we ensure the feasibility of the corresponding path constraint by selecting an appropriate patch from our patch pool. The above mentioned path constraint for P4 is satisfiable; however, our approach would not explore it because there is no patch in the current patch pool, which would allow taking this path.

Advantages of concolic program repair. Our approach has the major advantage to explore both spaces, input and patch, *simultaneously*, saving a significant cost in terms of time and space enumeration: (1) we refine the patch space based on the exploration in the input space, while (2) we also can rule out parts of the input space, which contradicts with the patch space. We are able to reason about a *large portion* of concrete patches with every single iteration of concolic execution by using *abstractions* in the *patch space*. For example, with three repair steps (II, III, and IV) we can reduce the patch space by 68 concrete patches. In general, the more paths we explore, the better the refinement would be, thus finding the most accurate patch. Furthermore, instead of focusing only on specific inputs but rather on the obtained path constraint, we are able to test a *large portion* of the *input space* captured by an input partition. Additionally, as illustrated in our example, our approach performs some *path reduction*: during concolic exploration, we make sure that for every new generated input, there is at least one patch in the current patch pool, which can exercise the corresponding path. Otherwise, the path will not be explored.

In conclusion, these advantages allow us to reduce the pool of candidate expressions, as compared to existing state-of-the-art techniques like counterexample-guided inductive synthesis (CEGIS) [31, 32] and ExtractFix [8].

3 Methodology

In this work we propose a *concolic program repair* technique, which incrementally explores the input space, while refining the patch space.

3.1 Patch Definition

Our technique supports two notions of patches: *concrete* and *abstract*. An abstract patch represents a patch template, which contains parameters that can have values satisfying a specified constraint. Concrete patches do not include such parameters. Our methodology focuses on abstract patches because, having abstract patches, the repair process needs to generate and maintain a smaller amount of patch candidates. Furthermore, the patch space reduction can attempt to refine the parameter constraints before discarding a patch. Therefore, we define a patch ρ as the 3-tuple

$$(\theta_\rho, T_\rho, \psi_\rho)$$

with the set of program variables X_P, the corresponding subset of input variables $X \subseteq X_P$, and the set of template parameters A:

- $\theta_\rho(X_P, A)$ denotes the repaired (boolean or integer) expression
- $T_\rho(A)$ represents the conjunction of constraints $\tau_\rho(a_i)$ on the parameters $a_i \in A$ included in θ_ρ:
$$T_\rho(A) = \bigwedge_{a_i \in A} \tau_\rho(a_i)$$
- $\psi_\rho(X, A)$ is the *patch formula* induced by inserting the expression θ_ρ into the buggy program

This patch definition covers both notions *abstract* and *concrete*. For concrete patches the set of parameters A is either empty and T_ρ is trivially `True`, or the constraints on the parameters $a_i \in A$ allow only one concrete value each.

Example. Assuming there is a buggy location in a program like **if**(ρ)**then**..**else**.., where the patch ρ is included in the if condition. Then a repaired expression could be $\theta_\rho := x > a$ with the parameter value constraint $T_\rho = \tau_\rho(a) := (a \geq -10 \land a \leq 10)$ and the corresponding patch formula $\psi_\rho := x > a$.

Patch Formula. In our notation ψ_ρ does not represent the patch expression but rather the constraint induced by the patch. For our approach a patch is technically represented as an expression tree, which can be transformed into an SMT formula, by considering the semantics of the operators (or components) appearing in the expression θ_ρ. The information about the patch location (i.e., where the repaired expression will be inserted) and the transformed expression tree is what we call the *patch formula*. Therefore, if the patch represents the right hand-side of an assignment like y=ρ with $\theta_\rho := x - a$, then the patch formula is derived as $\psi_\rho := y = x - a$, using the patch context information. We acknowledge that such a patch formula is generally not required for the definition of a patch. In fact, the patch formula can be derived from combining the information about the patch location and the patch expression (see Section 3.5). However, our approach technically requires such an artifact in order to reason about the patch.

3.2 Overview: Concolic Repair Algorithm

As *input*, our approach requires the buggy program, a repair budget, the fault locations, a user specification, the language components for the synthesis, and optionally, a set of initial test cases. The user specification identifies a constraint on the desired program behavior (in addition to satisfying the given test cases). It does not need to be a complete formal specification of the correct program behavior, but represents a constraint on the expected observation, provided as a logical formula. For example the user can assert crash-freedom or some specific logical behavior (e.g., a constraint on the resulting output). If no error-exposing input is available, we need to generate at least one failing input (with regard to the user-provided specification) to start the concolic exploration. Therefore, we can use offline techniques like Directed Greybox Fuzzing [3]. Note that the generation of the one failing test is a pre-processing to our technique. Otherwise, we assume that at least one failing test is available, which our method seeks to repair, apart from making sure that the user-provided specification holds for all paths traversed via concolic exploration.

As *output* our approach produces a set of patches, which satisfy the initial test case (repairing the given failing test case, if one is available) and which do not violate the given specification for (a subset of, depending on the repair budget) the other paths of the program. The patches are *ranked* based on the evidence we see during input space exploration.

Algorithm 1 shows the general workflow of concolic repair, which implements three phases: (1) *patch pool construction* (see Section 3.3), (2) *path exploration* (see Section 3.4), and (3) *patch reduction* (see Section 3.5). The initial phase of synthesis produces a pool of patches P (see line 1 in Algorithm 1) by leveraging a component-based synthesizer. This patch pool is going to be refined in the following repair loop (see line 2 to 11). The repair loop itself will be continued as long as there are remaining patches to refine or the repair budget is not exceeded. In phase (2) (i.e., inside the repair loop), we pick a new input t to explore more program paths (see line 3). With input t we also retrieve a patch candidate ρ from the patch pool P, such that inserting ρ in the patch location

Algorithm 1: General Concolic Repair

```
Input: set of initial test cases $I$, buggy locations
       $L = (patchLoc, bugLoc)$, budget $b$,
       specification $\sigma$, language components $C$
Output: set of ranked patches $P$
1  P ← Synthesize(C, I, L)
2  while $P \neq \emptyset$ and CheckBudget(b) do
3      $t, \rho$ ← PickNewInput($P$)
4      if no input $t$ available then
5          return P
6      end
7      $\phi_t, hit_{patch}, hit_{bug}$ ← ConcolicExec($t, \rho, L$)
8      if $hit_{patch}$ then
9          $P$ ← Reduce($P, \phi_t, \sigma, hit_{bug}$)
10     end
11 end
12 return P
```

allows t to have a feasible path in the patched program. If there is no such input t available, then there is no more input space to explore and the algorithm will return the identified patches (see line 4 to 6). Otherwise, we perform a concolic execution of the program with input t, patch candidate ρ, and the information about the:

- *patch location*, where the **repair** is located and
- *bug location*, where the buggy behavior is **observable**.

It results in the path constraint ϕ_t and whether the patch location (hit_{patch}) and the bug location (hit_{bug}) have been exercised by the execution (see line 7). Afterwards, in phase (3), we aim to reduce the patch pool P based on the current observations and the given specification σ. Before calling the Reduce function in line 9, we check whether the current path actually exercises the patch location (see line 8), otherwise there is no reduction possible.

3.3 Phase 1: Patch Pool Construction

In order to generate the initial patch pool P we leverage a component-based synthesizer, which focuses on the synthesis of boolean and integer expressions. Our approach assumes that the necessary patch-ingredients are provided as input to our technique. This includes the available program variables and the arithmetic/comparison operators for the synthesis. Before starting the actual synthesis we employ a *controlled* symbolic execution [23] to retrieve the path constraints for the initial test cases. Therefore, we mark the patch variables as symbolic at the patch location. The result of this symbolic execution is a set of path constraints with their corresponding expected outputs given by the test cases.

The synthesis starts with generating a set of expression trees based on the available components and the required expression type at the patch location. We support the arithmetic operations $\{+, -, *, /\}$ as well as the remainder operation, the comparison operators $\{=, \neq, <, \leq, >, \geq\}$, the boolean operators $\{\wedge, \vee, \neg\}$, and usage of parameters like $\{a, b, c, ...\}$. More components can be easily added to our synthesizer by providing them in the SMT-LIB format. For example, for each program to be repaired, the available variables are provided as additional components to the synthesizer. The final set of expression trees contains all feasible combinations of the given components that fit the required expression type. Afterwards, the synthesizer enumerates over these trees and validates that the corresponding expressions repair the program for the constraints retrieved by the controlled symbolic execution. All successfully validated expression trees, will be put in the resulting patch pool. If the expression tree includes parameters, the synthesizer will generate a constraint on these parameters (based on a pre-selected range).

3.4 Phase 2: Path Exploration

The path exploration is concerned with two issues: (a) how to pick a new input t and (2) how to efficiently retrieve the corresponding path constraint ϕ_t. In the first loop iteration the new input is chosen based on the provided test cases or randomly if there are no test cases available. Afterwards, based on the previous path constraint, the PickNewInput function (see line 3 in Algorithm 1) applies generational search [10] to obtain new inputs: by negating every suffix term in the constraint, we can retrieve the maximum number of new path constraint prefixes.

While checking the satisfiability of the obtained path constraint prefixes, we also determine whether there exists a patch candidate ρ in our current patch pool, which allows to exercise this path. In this way, we prune paths, for which no patch is feasible. We call this pruning of the input space *path reduction*. After checking the satisfiability, we can generate a set of new inputs, which are ranked based on how often they trigger the execution of the patch and bug location. In this way, a set of new inputs is maintained, which can be worked on and extended in every repair iteration. The complete path constraint is then retrieved by concolically executing the new input, and injecting the patch formula ψ_ρ (for a patch expression ρ) into the path constraint.

3.5 Phase 3: Patch Reduction

The Reduce function in Algorithm 1 (see line 9) tries to shrink the patch pool and to possibly refine the available abstract patches. Its workflow is shown in Algorithm 2.

3.5.1 Criterion for Patch Reduction.

For every patch ρ in the patch pool P we need to make sure that there is no violation of the specification σ for all inputs that are specified by the given path constraint. Otherwise, the patch needs to be removed. More specifically, we need to make sure that there exist parameter values parameters $a_i \in A$ within in the

Algorithm 2: Reduce function

```
   Input: patch pool $P$, path constraint $\phi$, specification
          $\sigma$, bug location hit $hit_{bug}$
   Output: reduced patch pool $P'$
1  $P' \leftarrow P$
2  for $\rho \in P$ do
3      $\pi \leftarrow \phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A)$
4      if IsSat($\pi$) then
5          if $hit_{bug}$ then
6              $P' \leftarrow P' \setminus \rho$
7              $T'_\rho \leftarrow$ RefinePatch($\phi$, $\rho$, $T_\rho$, $\sigma$)
8              if $T'_\rho \not\equiv$ False then
9                  $P' \leftarrow P' \cup \{\rho$ with $T'_\rho\}$
10             end
11         end
12         UpdateRanking($\rho$)
13     end
14 end
15 return $P'$
```

constraint $T_\rho(A)$ so that for all inputs $x_i \in X$, which satisfy the path constraint $\phi(X)$ and the patch formula $\psi_\rho(X, A)$, there is no violation of the specification $\sigma(X)$. Given $A = \{a_1, a_2, .., a_n\}$ and $X = \{x_1, x_2, .., x_m\}$, this means:

$$\exists a_1, a_2, .., a_n \forall x_1, x_2, .., x_m : \quad \phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A) \implies \sigma(X) \quad (1)$$

In our approach we do not only ensure that there exists **one** value for each parameters a_i, but we iteratively refine the constraint $T_\rho(A)$ to reduce the patch space as much as possible and to ensure that the specification holds for **all** (refined) values for each parameter a_i:

$$\forall a_1, a_2, .., a_n \forall x_1, x_2, .., x_m : \quad \phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A) \implies \sigma(X) \quad (2)$$

We want this formula (2) to hold after refinement, and hence it is used to guide our abstract patch refinement.

3.5.2 Reduction Algorithm. Algorithm 2 describes the reduction function for abstract patches. The function iterates over every patch and searches for specification violations. Before calling the patch refinement in line 7, there are two additional pre-checks, to make sure that we can reason about the patch within the current path constraint. First we check whether the path constraint ϕ and the current patch ρ (see line 3 and 4) are feasible. Secondly, we check whether the bug location is exercised by the current execution (see line 5) so that the buggy behavior is observable.

If both checks are passed, then we investigate whether the patch ρ with constraint T_ρ needs to be refined by searching for counterexamples for formula (2). The only option for the patch refinement, based on our definition of abstract patches (see Section 3.1), is to refine the constraint T_ρ. The implementation details for the patch refinement are presented in Section 4. If no refinement is feasible, then the patch will be eventually removed.

3.5.3 Patch Ranking. In addition to reducing the patch space, our approach attempts to rank the remaining patches. The rank of each patch ρ will be increased as long the patch is feasible with the path constraint ϕ (see line 12 in Algorithm 2). Otherwise the ranking will be not modified because we cannot reason about the patch with regard to the current path constraint. If the path exercises the bug location, then the patch will be ranked additionally higher (as compared to the situation where it does not exercise the bug location). Intuitively, this means that (1) patches that are compatible with the current path constraint will be ranked higher because we have seen more evidence for their correctness (in terms of the explored input space). In addition, (2) patches that also exercise the bug location will be ranked even higher because they exercised the program location, where potential errors are observable. Patches that are compatible with the path constraint and do not exercise the bug location could still be erroneous, but there has been no possibility to observe the error. We only rank those patches which do not show any violation of the specification for the explored input space.

In addition, we deprioritize patches that change the program behavior significantly, specifically *deletion of functionality* — which can happen if the guard of a conditional statement is changed by a patch to tautologies or their negation. Based on our formula (2) we cannot remove these patches because they do not violate the specification. However, functionality deletion is in general not desirable; as stated in a recent study [26], this kind of functionality deleting patches are present in the earlier works on search-based program repair and are overfitting. Although we cannot remove these patches, our patch ranking mechanism deprioritizes them. Therefore, for all patch candidates, we check whether the insertion of the patch affects the control flow of the inputs flowing through the path (even if the insertion of the patch does not violate the user-provided specification). We deprioritize such patches, and increase the rank of the other patches, and this ranking fine-tuning is accumulated over all the paths explored. Further fine-tuning of this heuristic is possible via model counting [5, 11] to find the proportion of inputs in a path affected by a patch insertion.

4 Abstract Patch Refinement

During patch space reduction (see Algorithm 2) we try to refine the available abstract patches whenever we identify a corresponding violation of specification σ. This is achieved by efficiently refining the parameter constraint T_ρ of the abstract patch ρ as shown in Algorithm 3.

Removal of non-refinable constraints. Before starting the fine-grained refinement of T_ρ, the Algorithm 3 checks whether there is a refinement of T_ρ feasible, which will make the specification pass. It checks whether (a) the conjunction of the path constraint with the specification (see formula ω_{pass1} in line 1) is satisfiable, followed by the check whether (b) the conjunction of the path constraint with the current patch constraint still allows to pass the specification (see formula ω_{pass2} in line 3). If (a) is satisfiable, but (b) is unsatisfiable, the parameter constraint does not contain any value that repairs the specification violation, and hence, can be discarded completely.

Counterexample exploration. After these initial checks, the algorithm searches counterexamples for the general formula (2) from Section 3.5.1 (see formula ω_{fail} in line 8). They capture violations of the specification, which need to be excluded by our refinement of T_ρ. If there exists no such model for formula ω_{fail}, then the parameter constraint needs no further refinement and the current constraint can be returned (see line 31). But if there is a model m_A, the Split function removes the model from the current constraint T_ρ and splits it into multiple regions (see line 11).

Region representation. We assume that the parameter constraint can be split into k regions $R = \{r_1, r_2, ..., r_k\}$ so that the constraint represents the disjunction of the separate regions. This limits the search space during refinement and can lead to removal of regions, which do not satisfy the specification. For example, consider a parameter space with one parameter a and the constraint $T_\rho(a) := (l \leq a) \wedge (a \leq u)$. Having the counterexample m_a, the Split function replaces the existing region with two new regions:

$$r_1 := (l \leq a) \wedge (a \leq m_a - 1)$$
$$r_2 := (m_a + 1 \leq a) \wedge (a \leq u)$$

Even if T_ρ already consists of multiple regions, only one region will be affected by the removal of the counterexample. In general there will be $3^n - 1$ additional regions introduced (where n is the number of parameters), while some of them might be merged later with surrounding regions.

Recursive refinement. The algorithm further checks for specification violations (see line 16 to 26) by recursively calling the refinement function on the regions (see line 19). Each recursive call is guarded by a check whether the current region r_i is compatible with the path constraint ϕ and the current patch formula (see line 17 and 18). Otherwise we cannot reason about the region. After iterating over all regions, the algorithm attempts to merge contiguous regions (see line 27), and finally, returns the disjunction of the refined parameter regions (see line 28).

Algorithm 3: RefinePatch function

```
   Input: path constraint $\phi$, abstract patch $\rho$, parameter
          constraint $T_\rho$, specification $\sigma$
   Output: refined constraint $T'_\rho$
1  $\omega_{pass1} \leftarrow \phi(X) \wedge \sigma(X)$
2  if IsSat($\omega_{pass1}$) then
3      $\omega_{pass2} \leftarrow \phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A) \wedge \sigma(X)$
4      if $\neg$IsSat($\omega_{pass2}$) then
5          return False
6      end
7  end
8  $\omega_{fail} \leftarrow \phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A) \wedge \neg\sigma(X)$
9  $m_A \leftarrow$ GetModel($\omega_{fail}$)
10 if $m$ exists then
11     $R = \{r_1, r_2, .., r_k\} \leftarrow$ Split($T_\rho$, $m_A$)
12     if $R = \emptyset$ then
13         return False
14     else
15         $R' \leftarrow \{\}$
16         for $r_i \in R$ do
17             $\pi \leftarrow \phi(X) \wedge \psi_\rho(X, A) \wedge r_i(A)$
18             if IsSat($\pi$) then
19                 $r'_i \leftarrow$ RefinePatch($\phi$, $\rho$, $r_i$, $\sigma$)
20                 if $r'_i \not\equiv$ False then
21                     $R' \leftarrow R' \cup \{r'_i\}$
22                 end
23             else
24                 $R' \leftarrow R' \cup \{r_i\}$
25             end
26         end
27         $R' \leftarrow$ Merge($R'$)
28         return $\bigvee_{r'_i \in R'} r'_i$
29     end
30 else
31     return $T_\rho$
32 end
```

5 Evaluation

The goal of our work is to efficiently navigate the patch space and find the correct patch that works beyond the provided test suite. We compare our technique with the related counterexample-guided inductive synthesis (CEGIS) [31, 32] because it also can be employed to navigate the patch space via patch refinement in order to generate the correct patch. Note that the above proposed technique of concolic program repair is *not* tailored to a specific class of errors. However, the low dependence on existing test cases fits well the context of repairing security vulnerabilities. Therefore, we present an empirical comparison with the state-of-the-art program repair tools Angelix [23], and Prophet [21], and also the recently proposed tool ExtractFix [8] for repairing security vulnerabilities. To highlight CPR's general repair capabilities, we also include additional subjects from the ManyBugs [13] benchmark. Furthermore, we show CPR's ability to fix logical errors for subjects from the SV-COMP benchmark [33]. All experimental data, as well as the CPR tool, is available from: https://cpr-tool.github.io/

Benchmark Suite. ExtractFix [8] is a state-of-the-art vulnerability repair tool, which generates fixes for security vulnerabilities by computing a crash-free constraint using a sanitizer. The crash-free constraint is used as the oracle for patch generation, and in our case, it can serve as the program specification. We follow a different workflow by first synthesizing patches at a given fault location and then gradually improving them based on a concolic exploration. We use their benchmark, which includes real-world applications with reported security vulnerabilities, and hence, it can be used to evaluate the efficacy of our technique in repairing security vulnerabilities. The collected subjects from the ManyBugs [13] benchmark show a partial subset of programs that can be handled with our underlying concolic engine KLEE [4]. Most of these subjects represent general errors. SV-COMP [33] is a common benchmark for evaluating the effectiveness and efficiency of state-of-the-art verification techniques. We identified C programs from SV-COMP, which include reachable assertion errors and for which there is another program in the benchmark, which represents a repaired version (i.e., the assertion is present but the error is not reachable), while the repair is not just a modification of the assertion's condition, but a logical change in the program before the assertion is reached. For our experiments, we have chosen 10 programs that satisfy the stated conditions.

Experimental Setup. Our implementation of the concolic engine is an extension of KLEE [4]. All experiments are conducted on a Dell Power Edge R530 with Intel(R) Xeon(R) CPU E5-2660 processor and 64GB RAM. We use Docker containers to exploit and repair the vulnerable applications. The experiments have been executed with the timeout of 1 hour to match the experiments of ExtractFix [8], allowing comparison with other repair tools. The language components for the synthesis are selected as needed for the specific subject and the parameters for the abstract patches have been limited to be within the range [-10,10]. For each experiment, (at least) one failing test case is provided as the initial test case. For subjects in the ExtractFix benchmark the failing test case is the exploit. For subjects in the ManyBugs benchmark there are multiple failing and passing test cases, while we provide CPR only the failing test cases. For subjects in SV-COMP we manually generate a failing test to trigger assertion errors. For ExtractFix and ManyBugs, we derive simple specifications from the programs themselves, e.g., that a program should not return an erroneous status code. The specification for the SV-COMP subjects is directly extracted based on the included assertions. For our experiments, the fault locations have been provided manually to CPR.

Our CEGIS Implementation. CEGIS comes in various forms in existing works [1, 31, 32]. We implement our own custom version of CEGIS with regard to the concepts in [32] by reusing as much components as possible from our tool CPR so that we can enable a fair comparison between the concepts with minimized impact of implementation differences. More specifically, our CEGIS implementation reuses CPR's concolic engine to provide a common path exploration for both techniques and reuses CPR's synthesizer to explore the same patch space. This custom CEGIS implementation supports the patch generation using a counterexample-guided refinement of the synthesis constraint. It starts with a concolic exploration of the input space to construct a set of path constraints. Afterwards, we synthesize a patch for the derived constraints (i.e., user-provided specification and witnessed program paths in previous concolic exploration). We then verify if the synthesized patch can produce a counterexample such that the specification is violated. If a counterexample can be found, the current patch will be thrown away, and the counterexample model is added to the synthesis constraint. The synthesizer will generate a new patch and the iteration continues until there is no further counterexample, or the patch space is covered.

It is necessary to limit the concolic exploration of CEGIS to make the techniques comparable. In our experiments, we split the overall timeout of 1 hour for CEGIS into 30 minutes initial path exploration and 30 minutes patch refinement. The conceptual difference between CEGIS and CPR is that CEGIS explores the patch space and input space *one patch / one input* at a time, while CPR explores *partitions* in both the patch space and the input space.

5.1 Our CEGIS implementation

Table 1 shows the results of the comparison between the two techniques. Column *Components* indicates the number of language components passed to our synthesizer. The sub columns *General* and *Custom* represent the number of components from the *general* synthesis language and number of *custom* components created specifically for the respective test subject. Columns $|P_{Init}|$ and $|P_{Final}|$ show the number of patches in the plausible patch space at the start of the refinement and at the end respectively. CEGIS does not maintain a patch pool like CPR, but only generates one patch that satisfies the collected constraints. However, the current patch pool size can be calculated by instructing the synthesizer to produce all currently feasible patches. $|P_{Init}|$ is for CEGIS the same as for CPR because we share the same inputs and synthesizer. Column *Ratio* shows the percentage of the patch space reduction. Column ϕ_E indicates the number of program paths *explored* for the refinement. Column ϕ_S indicates the number of program paths *skipped* during the refinement due

Table 1. Comparison between our CEGIS implementation and CPR with regard to patch pool reduction ratio and input space reduction ratio. Benchmark: ExtractFix. The experiments have been executed with timeout of 1 hour.

ID	**Buggy Program**		**Components**		**Our CEGIS Implementation**					**CPR**					
	Project	Bug ID	General	Custom	$\lvert P_{Init}\rvert$	$\lvert P_{Final}\rvert$	Ratio	ϕ_E	Correct?	$\lvert P_{Init}\rvert$	$\lvert P_{Final}\rvert$	Ratio	ϕ_E	ϕ_S	Rank
1	Libtiff	CVE-2016-5321	2	3	174	174	0 %	17	✗	174	104	40%	67	77	2
2	Libtiff	CVE-2014-8128	4	3	260	260	0%	0	✗	260	260	0%	0	0	1
3	Libtiff	CVE-2016-3186	4	3	130	130	0%	13	✗	130	130	0%	13	1	11
4	Libtiff	CVE-2016-5314	4	4	199	198	1%	10	✗	199	197	1%	21	4	2
5	Libtiff	CVE-2016-9273	4	3	260	260	0%	5	✗	260	141	46%	10	2	8
6	Libtiff	bugzilla 2633	4	3	130	130	0%	66	✗	130	130	0%	109	21	8
7	Libtiff	CVE-2016-10094	4	3	130	130	0%	23	✗	130	77	41%	34	114	6
8	Libtiff	CVE-2017-7601	4	2	94	94	0%	27	✗	94	94	0%	78	107	2
9	Libtiff	CVE-2016-3623	4	3	130	130	0%	60	✗	130	100	23%	102	21	1
10	Libtiff	CVE-2017-7595	4	3	130	130	0%	10	✗	130	130	0%	18	31	1
11	Libtiff	bugzilla 2611	4	3	130	130	0%	61	✗	130	112	14%	87	15	1
12	Binutils	CVE-2018-10372	5	3	74	74	0%	9	✗	74	39	47%	25	1	33
13	Binutils	CVE-2017-15025	4	3	130	130	0%	0	✗	130	130	0%	0	0	6
14	Libxml2	CVE-2016-1834	4	3	260	260	0%	6	✗	260	260	0%	22	0	12
15	Libxml2	CVE-2016-1838	4	4	199	199	0%	4	✗	199	199	0%	4	0	10
16	Libxml2	CVE-2016-1839	5	3	65	65	0%	0	✗	65	65	0%	0	0	14
17	Libxml2	CVE-2012-5134	4	3	260	260	0%	44	✗	260	134	48%	80	271	7
18	Libxml2	CVE-2017-5969	4	3	260	260	0%	0	✗	260	154	41%	21	2	1
19	Libjpeg	CVE-2018-14498	4	3	260	260	0%	42	✗	260	128	51%	78	108	2
20	Libjpeg	CVE-2018-19664	4	3	130	130	0%	43	✗	130	130	0%	84	26	1
21	Libjpeg	CVE-2017-15232	5	3	955	955	0%	0	✗	955	955	0%	0	0	26
22	Libjpeg	CVE-2012-2806	4	3	260	259	0%	68	✗	260	145	44%	110	3	3
23	FFmpeg	CVE-2017-9992	6	3	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
24	FFmpeg	Bugzilla-1404	4	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
25	Jasper	CVE-2016-8691	4	3	260	260	0%	72	✗	260	96	63%	69	7	1
26	Jasper	CVE-2016-9387	5	3	65	65	0%	54	✗	65	17	74%	111	1	✗
27	Coreutils	Bugzilla 26545	5	3	1025	1025	0%	74	✗	1025	949	7%	119	2	25
28	Coreutils	GNUBug 25003	4	4	199	198	1%	114	✗	199	172	14%	196	0	6
29	Coreutils	GNUBug 25023	4	2	64	64	0%	32	✗	64	64	0%	1	2	7
30	Coreutils	Bugzilla 19784	4	3	-	-	-	-	-	770	770	0%	6	0	38

Table 2. Comparison with repair tools. The experiments have been executed with timeout of 1 hour [8]. For Prophet and Angelix the results show only the top-ranked patch, while for ExtractFix the results capture the only patch generated.

Benchmark	**Program**	**#Vul**	**Generated Patches**			**Correct Patches**		
			Prophet	Angelix	ExtractFix	Prophet	Angelix	ExtractFix
ExtractFix	Libtiff	11	7	7	9	1	0	6
	Binutils	2	-	-	2	-	-	1
	Libxml2	5	3	0	4	0	0	2
	Libjpeg	4	3	-	3	1	-	2
	FFmpeg	2	-	-	2	-	-	2
	Jasper	2	2	2	2	0	0	1
	Coreutils	4	2	-	2	0	-	2
Total		30	17	9	24	2	0	16

to patch in-feasibility. Column *Correct*? indicates whether CEGIS finishes with a patch that is syntactically or semantically equivalent with the developer patch and column *Rank* shows the corresponding highest rank position. The N/A values for ID 23 and 24 in Table 1 indicate that both CEGIS and CPR have not been able to produce any results because the execution of the test driver code resulted in an unexpected memory fault for our underlying concolic execution engine. The "-" signs for CEGIS for ID 30 mean that it was not able to generate any patch within the timeout.

Input and patch space exploration. The comparison of the *Ratio* columns in Table 1 shows that in 14 of 30 cases CPR can produce significantly better patch space reduction than CEGIS. In the remaining 16 cases, both perform similarly. For a few subjects, CPR resulted in 0% reduction, partly because of the loop unrolling (and hence longer paths) in symbolic

Table 3. Performance of CPR with regard to patch pool reduction ratio and input space reduction ratio for additional subjects from the ManyBugs benchmark. The experiments have been executed with timeout of 1 hour.

ID	Buggy Program		Components		CPR					
	Project	Subject ID	General	Custom	$\lvert P_{Init}\rvert$	$\lvert P_{Final}\rvert$	Ratio	ϕ_E	ϕ_S	Rank
1	Libtiff	ee65c74	4	3	6	6	0%	29	90	1
2	Libtiff	865f7b2	4	3	130	130	0%	24	68	5
3	Libtiff	7d6e298	5	4	4	2	50%	7	7	1
4	gzip	884ef6d16c	5	4	4821	4821	0%	11	0	36
5	gzip	f17cbd13a1	5	4	2	2	0%	0	1	1

execution. While this is an area we can work on, the ϕ_S column shows that CPR is already effective in combating path explosion by skipping additional paths over and above normal concolic execution. For all subjects, for which CPR produces some patch space reduction > 1%, it outperforms CEGIS. Furthermore, the ϕ_E columns show that CPR is also more efficient in exploring the input space: in 20 of 30 cases CPR explores more path constraints than CEGIS, in 2 cases CEGIS shows better results, and for the remaining 8 cases both perform similarly. Additionally, CPR can effectively skip infeasible path constraints (see Column ϕ_S).

Furthermore, CEGIS requires initial path exploration to construct the constraint for later patch verification. Therefore, in order to verify a patch, CEGIS uses a set of symbolic paths that capture portion of the program specification. In contrast, our technique CPR is an anytime algorithm that uses a single program path at a time for patch refinement. Processing a single path at a time, compared to a set of paths is more efficient during constraint solving.

Finding 1: CPR is more effective than CEGIS with regard to input space and patch space exploration.

Identifying the correct patch. In none of our 30 test subjects CEGIS can identify a patch, which is syntactically or semantically equivalent with the developer patch (see Column *Correct?*). The reason is that as soon as CEGIS identifies a patch, which does not violate the specification for the previously collected path constraints, it terminates and returns this current patch. In our experiments, such a patch often is a tautology or contradiction, which can be semantically equivalent to code deletion, as the patch would enforce early termination of the program to avoid the bug location. CPR includes such patches in the patch space (as long as they do not violate any specification), but our ranking system de-prioritizes such patches (see Section 3.5.3). Column *Rank* shows that CPR ranks the developer patch (or a semantic equivalent) relatively high, in 20 cases in the Top-10.

Finding 2: CEGIS tends to favor a simple patch that represents the deletion of functionality, which overfits to the given specification. CPR can leverage its ranking capabilities to identify the correct patch.

5.2 Existing Program Repair tools

CPR can be leveraged for constraint-driven repair, i.e., having just a few or no test cases, but a constraint, which can be used as a repair oracle. For this purpose, we focus on the comparison with the most recently proposed constraint-driven repair technique ExtractFix [8] and their corresponding data-set. On the data-set of ExtractFix, CPR generates the correct patch in top position for 7/30 subjects and in second position in 4/30 subjects, as shown in Table 1.

As already mentioned, ExtractFix uses a crash-free constraint as the guiding oracle to generate a patch. ExtractFix computes the weakest precondition for the patch by back propagating the crash-free constraint. Conceptually, ExtractFix explores the patch space using the crash-free constraint to determine the patch and then evaluates the effectiveness of the patch for the input space. In contrast, CPR can use the same crash-free constraint but explores the input space to determine the invalid values that can violate the crash-free constraint, and use this information to evaluate the effectiveness of the patch. The tool ExtractFix is also compared with conventional test-based repair tools Prophet and Angelix in [8].

Table 2 from [8] shows the results on the same security vulnerability benchmark. Column #*Vul* shows the count of vulnerabilities for each subject, which is in total 30. The columns *Generated Patches* and *Correct Patches* show the number of vulnerabilities, for which the techniques generated *plausible* and *correct* patches (i.e., syntactically or semantically equivalent to the developer patch).

Overall, we note that ExtractFix is a customized tool for repairing security vulnerabilities which hooks into specific sanitizers, whereas ours is a general-purpose program repair machinery. Table 3 shows the results from test-based repair of Manybugs subjects [13] that require a general-purpose repair technique; these cannot be handled by ExtractFix. CPR can generate correct patches for all of them, by leveraging the failing tests to drive concolic path exploration. In future, it is also possible to experimentally evaluate the usage of passing tests to drive concolic exploration in CPR.

Since Prophet and Angelix are test-driven general repair techniques, in addition to the failing test case, available developer test-suite are provided to both Angelix and Prophet

Table 4. Performance of CPR with regard to patch pool reduction ratio and input space reduction ratio for the repair of logical errors in SV-COMP. The experiments have been executed with timeout of 1 hour.

ID	Subject	Components		CPR					
		General	Custom	$\lvert P_{Init} \rvert$	$\lvert P_{Final} \rvert$	Ratio	ϕ_E	ϕ_S	Rank
1	loops/insertion_sort	4	3	260	132	49%	120	0	1
2	loops/linear_search	4	3	260	127	51%	109	17	1
3	loops/string	2	3	676	676	0%	37	0	2
4	loops/eureka	5	3	29	29	0%	107	27	3
5	loops-crafted-1/nested_delay	4	3	260	117	55%	9	8	4
6	loops/sum	4	3	260	236	9%	116	0	1
7	array-examples/bubble_sort	4	3	260	144	45%	34	19	2
8	array-examples/unique_list	1	2	5	4	20%	134	11	1
9	array-examples/standard_run	4	3	260	126	52%	68	41	1
10	recursive/addition	5	3	38	14	63%	138	1	4

(the programs in Table 2 come with test-suites from developers). ExtractFix and CPR do not need additional tests.

Angelix and Prophet. In contrast to our approach, ExtractFix is driven only by the initial test case while Angelix and Prophet both uses additional developer test cases. Despite being provided additional test cases, both Angelix and Prophet cannot produce many *correct* patches. Prophet can only identify correct patches for 2 of the vulnerabilities and Angelix is not able to correctly fix any of them, *as the top-ranked patch*. Most of the correct patches represent updated or inserted conditions, which are in the search space of both techniques. However, as mentioned in ExtractFix [8], the developer-provided tests for this benchmark are very limited, which may lead to overfitting patches. Therefore, Angelix cannot generate a rich specification for synthesis, and Prophet suffers from a large search space. Prophet and Angelix have the potential to repair more vulnerabilities if more tests are available, and if more of their ranking is examined, i.e., beyond the top-ranked patch.

Finding 3: Experimental evidence shows CPR can be used as test-guided general-purpose repair tool, as well as a tool for repairing security vulnerabilities.

5.3 Fixing Logical Errors

We further evaluate CPR on its capability to repair logical errors of a program provided as assertions or rich-text comments on the source code. Therefore, we investigate the possibility of repairing programs beyond simple oracles such as crash-freedom. We evaluate the efficacy of CPR in fixing logical errors on subjects from the SV-COMP benchmark, which is popular for automated program verification and provides such program specifications. As mentioned earlier, for our chosen SV-COMP programs the developer provided patch is available in the form of another program (so we can check whether CPR produced the correct patch), and the developer provided patch is not merely a change of the assertion but involves a change in the functionality.

Table 4 presents the results. The meaning of the columns is similar to Table 1 in Section 5.1. For all subjects, CPR can identify correct patches in the patch pool. Furthermore, due to the efficient space exploration, CPR achieves a patch space reduction ratio of up to 63 %. Only for one subject ($loops/eureka$) CPR was not able to produce any patch space reduction. The reason is that the assertion in the program was not strong enough to identify violations. However, CPR still has been able to rank the correct patch on position 3. In fact, for all of the 10 subjects CPR can rank the correct patches in the Top-10 and for five of them as Top-1.

Finding 4: CPR effectively repairs logical errors in SV-COMP, and ranks correct patches in Top-10 for all programs in our experiments.

5.4 Internal Evaluation of CPR components

Parameter Range. As mentioned in our *Experimental Setup* section, the parameter for the abstract patches in our experiments are limited within the range [-10, 10]. We conducted additional experiments to show the effects of other ranges. The results in Table 5 show that the number of initial patch candidates ($|P_{Init}|$) is growing with a larger parameter range. The effort for the initial patch pool construction is not largely affected because the concrete values for the parameters are not enumerated but abstracted in the range. The ranking of the correct patch itself is not necessarily affected as our experiments show. For *Jasper/CVE-2016-8691* the correct patch is correctly identified after the first iteration. For *Libtiff/CVE-2016-10094* the parameter range needs to include the constant 4 so that CPR can identify the correct patch. With a too narrow range like $[-1, 1]$ CPR cannot identify the correct patch.

Input Generation. The additional generation of inputs is an essential part of our path exploration phase (see Section

Table 5. Impact of different parameter ranges on the repair success of CPR. Benchmark: selection of ExtractFix. The experiments have been executed with timeout of 1 hour.

Buggy Program		Parameter	CPR					
Project	Bug ID	Range	$\#Iter.$	ϕ_E	$\|P_{Init}\|$	$\|P_{Final}\|$	Ratio	Rank
Jasper	CVE-2016-8691	[-1, 1]	70	68	44	15	66%	1
		[-10, 10]	70	69	260	96	63%	1
		[-100, 100]	70	79	2420	907	63%	1
Libtiff	CVE-2016-10094	[-1, 1]	35	34	22	10	55%	-
		[-10, 10]	35	34	130	77	41%	6
		[-100, 100]	27	26	1210	887	27%	6

Table 6. Average ratio of the number of generated inputs that hit the patch and bug location.

Benchmark	Avg. PatchLoc Hit	Avg. BugLoc Hit
ExtractFix	74.36%	40.23%
ManyBugs	57.14%	65.15%
SV-COMP	76.33%	79.08%

3.4). Our search heuristics drive the input generation to the bug location. Hitting the bug location is crucial, not only to rule out patches, but also to improve the patch ranking. Table 6 shows how often our generated inputs hit the patch and bug location on average. The results show that to a large extent our generated inputs do exercise the patch and bug location. However, for the ExtractFix benchmark hit count for the bug location is comparably low with 40.23%. In contrast to the SV-COMP subjects, where the inputs represent primitive data types, the ExtractFix subjects require complex input structures like images or XML files. Our input generation does not use an application-specific input grammar, which could lead to a significant improvement.

Patch Ranking. The changes in our ranking are based on whether the generated inputs exercise the patch and bug location under the specific patches. For many subjects the ranking of the correct patch is already very high after the first few iterations, and is not changed later. Our path exploration starts with inputs that exercise paths that are close to the path of the failing test case: hitting the bug location is more likely for those inputs. In some subjects, the ranking improved gradually over the repair time, e.g. *Coreutils/Bugzilla 26545* starts with the correct patch ranked at position 104 and it improves to 25 (after 65^{th} iteration). Change in ranking can happen due to patch candidates violating specification in the new paths.

6 Related Work

Symbolic Execution. Symbolic execution, the execution of a program with symbolic or unknown inputs, was suggested in 1976 as a mechanism for both program verification and testing [16]. In the subsequent decades, decision procedures for quantifier-free first order logic formula with symbols drawn from various background theories, or Satisfiability Modulo Theory (SMT) solvers, have matured. The maturity of back-end SMT solvers has further enabled the development of symbolic execution engines such as KLEE [4] and SAGE [10]. These symbolic execution engines are primarily used for path coverage based software testing. However, the efficient solving of constraints in general remains a challenge for symbolic execution. Concolic execution [9] represents a significant development in this regard. In concolic execution, a given concrete test input is executed but the symbolic formula documenting the path condition is mutated to generate subsequent test inputs for exploration. Since a concrete input is available, the path condition can be simplified as needed. In the recent past, symbolic execution has also been suggested as a specification inference mechanism for program repair (e.g., [25]), and this suffers from the path explosion problem of symbolic execution. Furthermore, the repair is with respect to a given set of tests, leading to potential overfitting. Our work on concolic program repair adapts concolic path exploration to generate tests and reduce candidate patches simultaneously.

Program Repair. Automated program repair [24] is an emerging technology, which seeks to automatically rectify program errors, typically as observed via failure of tests or assertions. Common techniques for automated repair include program mutations via genetic search [18], specification inference via symbolic execution or SAT solving [12, 23, 25], repair via abstract interpretation [19], code transplantation [28], and learning and prioritization of patch candidates and fix patterns [2, 20, 21, 27]. Our work is more related to specification inference based program repair. These approaches employ symbolic execution to generate a repair constraint, which the buggy program needs to satisfy to pass a given test-suite. Solutions to the repair constraint, in the form of patch expressions, are then obtained using program synthesis. Most of the existing works on test-based program repair suffer from test data overfitting, where the patched program fails for tests outside the given test-suite [14, 26]. To alleviate overfitting, one may use more general oracles beyond tests [6], or may generate tests to rule out overfitting patches [7]. Certain works develop customized repair strategies for fixing security vulnerabilities by either employing heuristics [15], by applying fix templates that avoid specific errors [29], or by hooking up with sanitizers [8]. In contrast, ours is a general purpose repair engine, though we have also shown its efficacy on the dataset of [8]. Our work generates tests from an initial seed test by modifying the path condition, in the style of concolic execution. However, the path of a test contains yet to be inserted patches. Hence the path exploration in concolic execution is accompanied by a systematic reduction of the pool of patch candidates in our approach. Finally, counterexample-guided inductive synthesis (CEGIS) [1, 31, 32] represents a synthesis technique, in which the desired solution is iteratively refined based on a loop between a *generator* and a *verifier*. Our approach also leverages counterexamples to reduce the patch space, and has some relationship to CEGIS. In our work, we use a counterexample-guided refinement of the parameter constraints of the available patches. The work of [17] performs concolic execution on specific tests to check whether a patch candidate meets a specification; if it does not, the resultant constraint is added for the generation of future repair candidates. In contrast, CPR works on abstract patch candidates and refines them. Furthermore, [17] terminates as soon as there is no counterexample anymore, which again can lead to functionality deleting patches.

7 Discussion

Limitations and Extensions. In the formulation of our repair algorithm, as well as in our experiments, we assume that the *correct* patch is included in the initial patch pool P. This is only the case, if our synthesis language/grammar covers this patch. In general, this assumption might not hold. In such a case, our ranking allows us to still present the most *promising* patches, which can only repair the program for a portion of the input space. Our approach currently focuses on repairing boolean and integer expressions. In future we want to extend our work to repair complete assignments as well as side-effect free function calls.

Inputs to our method. Our approach requires some ingredients that differs from existing program repair strategies: the user-provided (partial) specification and the fault locations (see the input description in Section 3.2). The specification allows us to reason about many program inputs going beyond a test suite. Other techniques rely on bug templates, sanitizers, existing test cases, or probabilistic models to reason about the correct behavior. Our specifications are lightweight, and our experiments show that even simple specifications can be used to rule out overfitting patches in an incremental manner. The fault location information is an input to our approach, which can be derived from statistical fault localization. Test-based repair tools may use a set of fault locations, while our approach currently works with one fault location at a time.

8 Perspective

A key difficulty in program repair (and program debugging) comes from the lack of *complete* specification of intended program behavior. Since a detailed specification of correct behavior is usually not available, existing program repair techniques are guided by tests. This inevitably leads to the pernicious problem of patch overfitting [26], where an automatically generated (plausible) patch may be perfectly fitted to pass a given set of tests, but not other tests. Herein lies the dilemma of program repair techniques today: how to generate a patch which works for a large set of tests, even if very few of them may be available to guide the patch generation?

In this paper, we take a fresh look at the problem of patch overfitting. We note that the patches produced by current program repair techniques may not even ensure very basic notions of correctness such as crash-freedom, or assertions, even when such simple specifications are readily available. Our solution for alleviating the patch overfitting problem, is to automatically and systematically *generate* tests. Our concolic exploration identifies overfitting patches that are plausible but do not satisfy the specification for at least one of the generated inputs. Furthermore, by removing incorrect but plausible patches we shrink the patch space and increase the ranking of the correct patch, alleviating patch overfitting. Our CPR tool also applies to test-suite based repair, by using failing / passing tests to drive concolic path exploration.

Technically, our approach suggests a dual use of symbolic execution for search-based test generation [4, 9] as well as specification inference based program repair [23, 25]. One could potentially replace symbolic execution with other automated test generation techniques in our method, such as recent systematic versions of greybox fuzzing [3].

Conceptually, we present a viewpoint of "*gradual correctness*" to alleviate patch overfitting, where systematic co-exploration of the input space and patch space, leads to less over-fitting patches, over time. This notion of gradual correctness, as proposed for program repair in CPR, can be relevant for related problem domains such as program synthesis, recovery and transplantation. Gradual correctness can thus serve as a general guiding principle to enhance the quality of automatically constructed code, over time.

Our CPR tool, the dataset, and our patches are available from: https://cpr-tool.github.io/

Acknowledgments

We thank Sergey Mechtaev for valuable discussions on patch synthesis, and help with implementation. We thank the anonymous reviewers and our shepherd Martin Rinard for insightful suggestions. This research is partially supported by the National Research Foundation Singapore (National Satellite of Excellence in Trustworthy Software Systems) and by the German Research Foundation (GR 3634/6-1 FLASH).

References

[1] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-Based Program Synthesis. *Commun. ACM* 61, 12 (Nov. 2018), 84–93. https://doi.org/10.1145/3208071

[2] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 159:1–159:27. https://doi.org/10.1145/3360585

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[5] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A Scalable Approximate Model Counter. In *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings (Lecture Notes in Computer Science)*, Christian Schulte (Ed.), Vol. 8124. Springer, 200–216. https://doi.org/10.1007/978-3-642-40627-0_18

[6] Hadar Frenkel, Orna Grumberg, Corina Pasareanu, and Sarai Sheinvald. 2020. Assume, Guarantee or Repair. In *Tools and Algorithms for the Construction and Analysis of Systems*, Armin Biere and David Parker (Eds.). Springer International Publishing, Cham, 211–227. https://doi.org/10.1007/978-3-030-45190-5_12

[7] 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)* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 8–18. https://doi.org/10.1145/3293882.3330558

[8] Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 14 (Feb. 2021), 27 pages. https://doi.org/10.1145/3418461

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[10] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM* 55, 3 (mar 2012), 40–44. https://doi.org/10.1145/2093548.2093564

[11] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. 2009. Model Counting. In *Handbook of Satisfiability*, Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 633–654. https://doi.org/10.3233/978-1-58603-929-5-633

[12] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-Based Program Repair Using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 173–188.

[13] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar T. Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Trans. Software Eng.* 41, 12 (2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[14] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65. https://doi.org/10.1145/3318162

[15] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. 539–554. https://doi.org/10.1109/SP.2019.00071

[16] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[17] Robert Könighofer and Roderick Bloem. 2013. Repair with On-The-Fly Program Analysis. In *Hardware and Software: Verification and Testing*, Armin Biere, Amir Nahir, and Tanja Vos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 56–71.

[18] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[19] Francesco Logozzo and Thomas Ball. 2012. Modular and Verified Automatic Program Repair. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 133–146. https://doi.org/10.1145/2384616.2384626

[20] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 727–739. https://doi.org/10.1145/3106237.3106253

[21] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[22] Fan Long and Martin C. Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 702–713. https://doi.org/10.1145/2884781.2884872

[23] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[24] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages. https://doi.org/10.1145/3105906

[25] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[26] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. 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* (Baltimore, MD, USA) *(ISSTA 2015)*. ACM, New York, NY, USA, 24–36. https://doi.org/10.1145/2771783.2771791

[27] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. 2020. Type error feedback via analytic program repair. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina

Torlak (Eds.). ACM, 16–30. https://doi.org/10.1145/3385412.3386005

[28] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic Error Elimination by Horizontal Code Transfer across Multiple Applications. *SIGPLAN Not.* 50, 6 (June 2015), 43–54. https://doi.org/10.1145/2813885.2737988

[29] Stelios Sidiroglou-Douskos, Eric Lahtinen, and Martin Rinard. 2015. *Automatic Discovery and Patching of Buffer and Integer Overflow Errors*. Technical Report. Massachusetts Institute of Technology, Cambridge, MA, USA. http://hdl.handle.net/1721.1/97087

[30] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 532–543. https://doi.org/10.1145/2786805.2786825

[31] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.

[32] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*. https://doi.org/10.1145/1168857.1168907

[33] SV-COMP Website. 2020. International Competition on Software Verification (SV-COMP). https://sv-comp.sosy-lab.org/.