Concolic Program Repair

Detecting and discarding over-fitting patches via systematic co-exploration of the patch space and input space

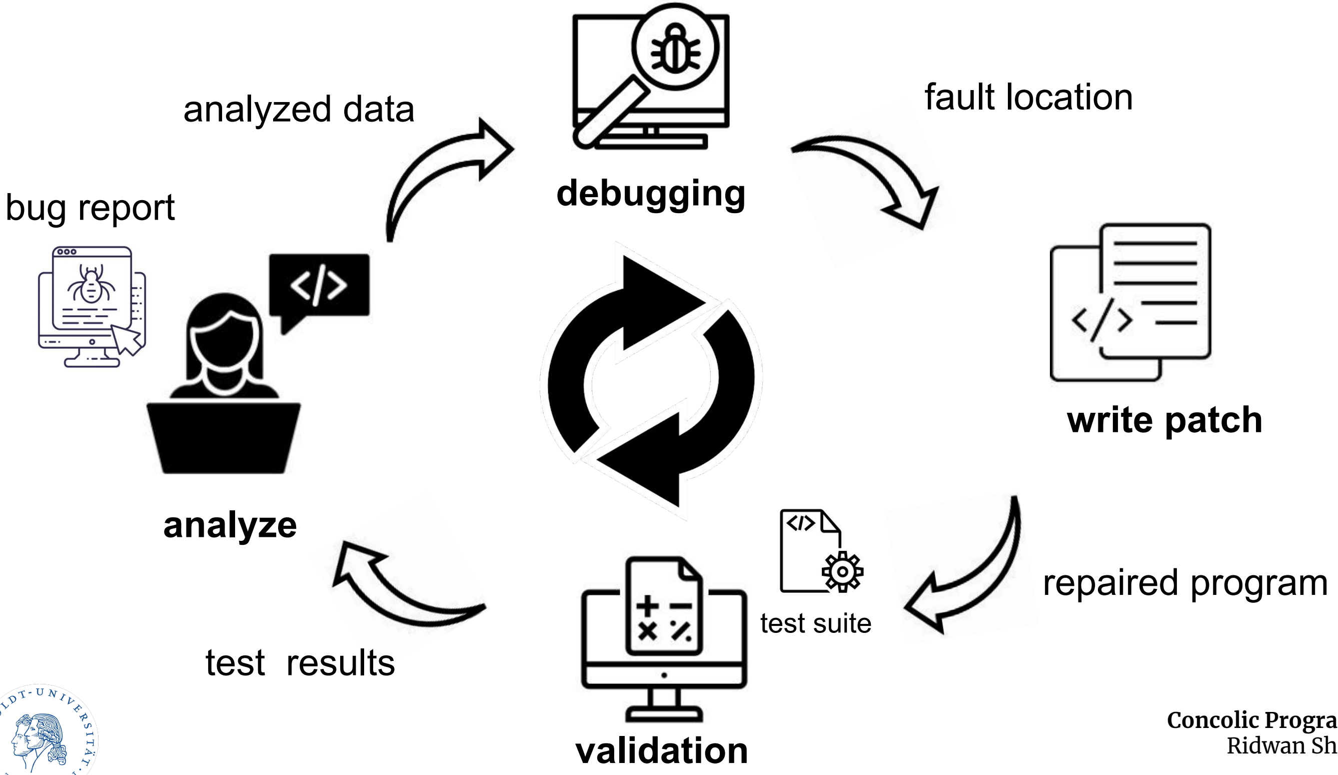
Ridwan Shariffdeen

Yannic Noller

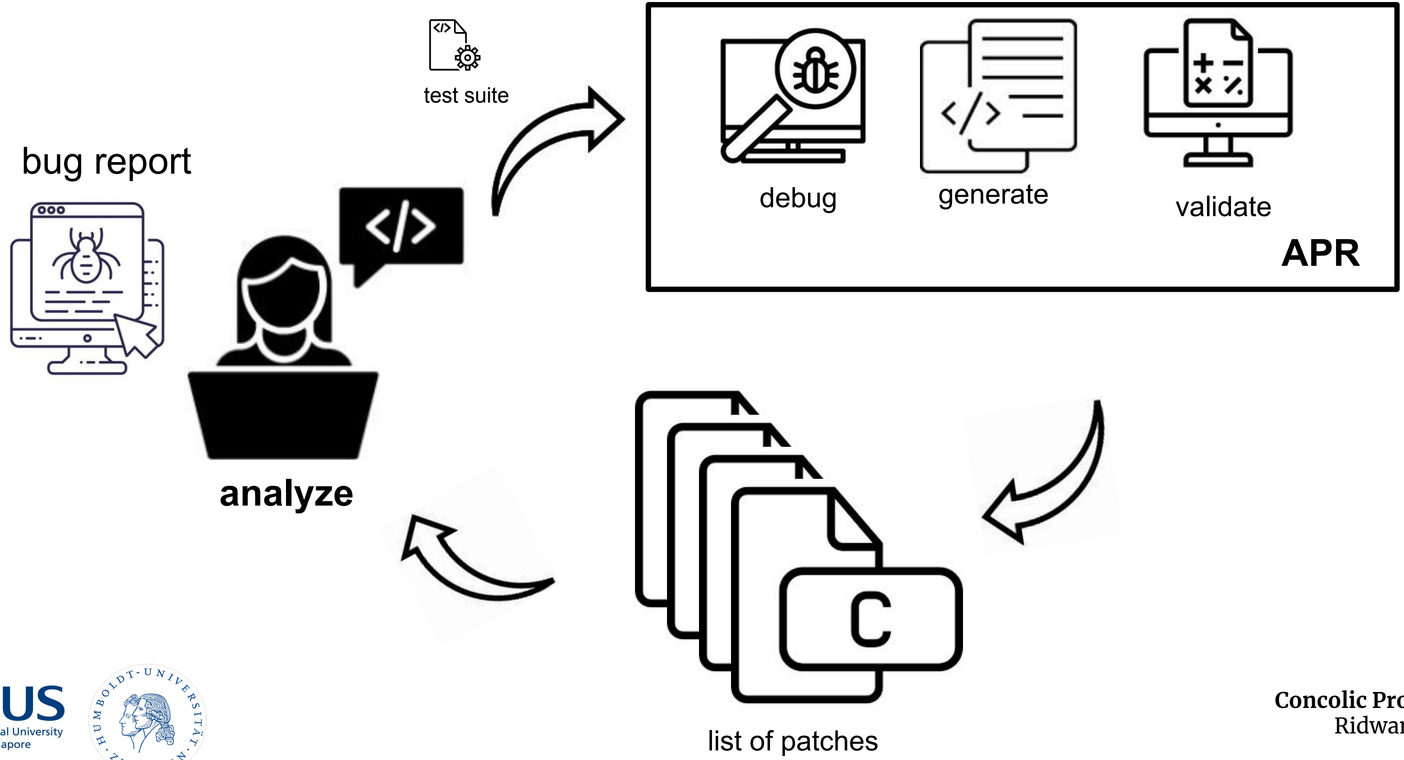
Lars Grunske

Abhik Roychoudhury

Typical Repair Workflow



Automated Program Repair



PITTSBURGH, PENNSYLVANIA
ICSE 2022
INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING

Trust Enhancement Issues in Program Repair

Yannic Noller, **Ridwan Shariffdeen**, Xiang Gao, Abhik Roychoudhury

IEEE/ACM 44th International Conference on Software Engineering (ICSE) 2022

Artifacts Available acm
Artifacts Evaluated acm Functional
Artifacts Evaluated acm Reusable

Artifacts Evaluated Reusable V1.1
Artifacts Available V1.1

Trust Enhancement Issues in Program Repair

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

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

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

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

ABSTRACT

Automated program repair is an emerging technology that seeks to automatically rectify bugs and vulnerabilities using learning, search, and semantic analysis. Trust in automatically generated patches is necessary for achieving greater adoption of program repair. Towards this goal, we survey more than 100 software practitioners to understand the artifacts and setups needed to enhance trust in automatically generated patches. Based on the feedback from the survey on developer preferences, we quantitatively evaluate existing test-suite based program repair tools. We find that they cannot produce high-quality patches within a top-10 ranking and an acceptable time period of 1 hour. The developer feedback from our qualitative study and the observations from our quantitative examination of existing repair tools point to actionable insights to drive program repair research. Specifically, we note that producing repairs within an acceptable time-bound is very much dependent on leveraging an abstract search space representation of a rich enough search space. Moreover, while additional developer inputs are valuable for generating or ranking patches, developers do not seem to be interested in a significant human-in-the-loop interaction.

ACM Reference Format:
Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510040

1 INTRODUCTION

Automated program repair technologies [14] are getting increased attention. In recent times, program repair has found its way into the automated fixing of mobile apps in the SapFix project in Facebook [28], automated repair bots as evidenced by the Repairnator project [44], and has found certain acceptability in companies such as Bloomberg [17]. While all of these are promising, large-scale adoption of program repair where it is well integrated into our programming environments is considerably out of reach as of now. In this article, we reflect on the impediments towards the usage of program repair by developers. There can be many challenges towards the adoption of program repair like scalability, applicability, and developer acceptability. A lot of the research on program repair has focused on scalability to large programs and also to large search spaces [12, 26, 28, 31]. Similarly, there have been various works on generating multi-line fixes [13, 31], or on transplanting patches from one version to another [41] — to cover various use cases or scenarios of program repair.

Surprisingly, there is very little literature or systematic studies from either academia or industry on the developer trust in program repair. In particular, what changes do we need to bring into the program repair process so that it becomes viable to have conversations on its wide-scale adoption? Part of the gulf in terms of lack of trust comes from a lack of specifications — since the intended behavior of the program is not formally documented, it is hard to trust that the automatically generated patches meet this intended behavior. Overall, we seek to examine whether the developer's reluctance to use program repair may partially stem from not relying on automatically generated code. This can have profound implications because of recent developments on AI-based pair programming[1], which holds out promise for significant parts of coding in the future to be accomplished via automated code generation.

In this article, we specifically study the issues involved in enhancing developer trust on automatically generated patches. Towards this goal, we first settle on the research questions related to developer trust in automatically generated patches. These questions are divided into two categories (a) expectations of developers from automatic repair technologies, and (b) understanding the possible shortfall of existing program repair technologies with respect to developer expectations. To understand the developer expectations from program repair, we outline the following research questions.

RQ1 To what extent are the developers interested to apply automated program repair (henceforth called APR), and how do they envision using it?

RQ2 Can software developers provide additional inputs that would cause higher trust in generated patches? If yes, what kind of inputs can they provide?

[*] Joint first authors
[†] Alternate email: gaoxiang9430@gmail.com

[1] Github Copilot https://copilot.github.com/

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510040

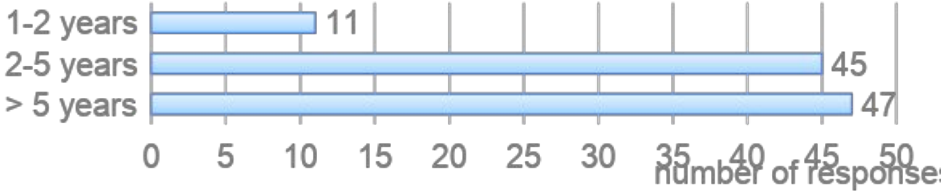
Developer Survey - Demographics

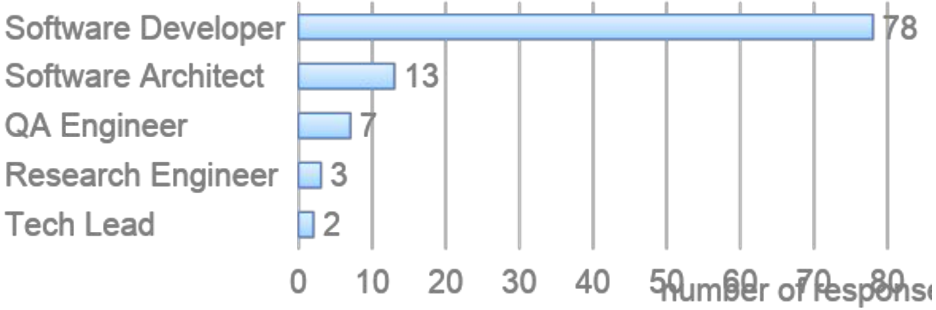
103 software practitioners

89% with **2+ years experience**

75% **Software Developers**

35 questions on trust in APR





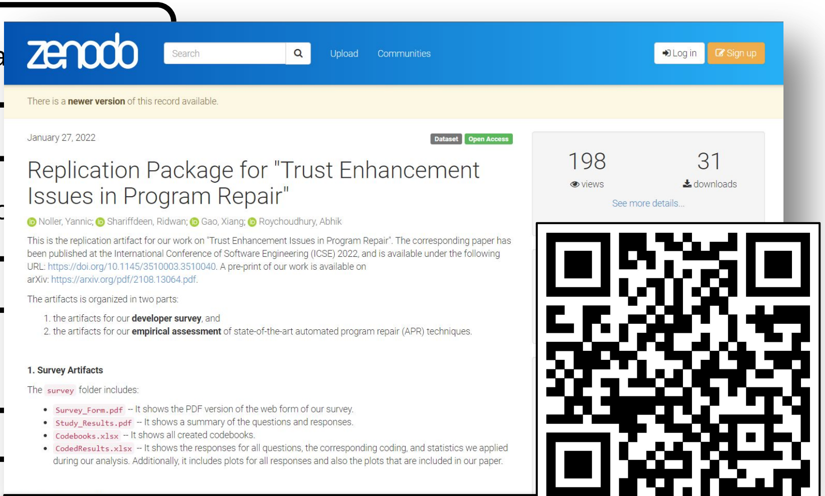
Insights from Developer Survey

I **Additional test-cases** improves trustworthiness of generated pa

II Developers are willing to **provide specification** to the repair proc

III Developers will only allow a maximum of **1-hour timeout**

IV Developers will only review up to **maximum of 5 patches**



There is a **newer version** of this record available.

January 27, 2022

Dataset Open Access

Replication Package for "Trust Enhancement Issues in Program Repair"

Noller, Yannic; Shariffdeen, Ridwan; Gao, Xiang; Roychoudhury, Abhik

This is the replication artifact for our work on "Trust Enhancement Issues in Program Repair". The corresponding paper has been published at the International Conference of Software Engineering (ICSE) 2022, and is available under the following URL: https://doi.org/10.1145/3510003.3510040. A pre-print of our work is available on arXiv: https://arxiv.org/pdf/2108.13064.pdf.

The artifacts is organized in two parts:

1. the artifacts for our **developer survey**, and
2. the artifacts for our **empirical assessment** of state-of-the-art automated program repair (APR) techniques.

1. Survey Artifacts

The `survey` folder includes:

- `Survey_Form.pdf` -- It shows the PDF version of the web form of our survey.
- `Study_Results.pdf` -- It shows a summary of the questions and responses.
- `Codebooks.xlsx` -- It shows all created codebooks.
- `CodedResults.xlsx` -- It shows the responses for all questions, the corresponding coding, and statistics we applied during our analysis. Additionally, it includes plots for all responses and also the plots that are included in our paper.

198 views

31 downloads

See more details...

https://zenodo.org/record/6303481

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. Our technique presents the viewpoint of *gradual correctness* – repair run over longer time leads to less overfitting fixes.

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

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

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

*Joint first authors

PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8391-2/21/06...$15.00
https://doi.org/10.1145/3453483.3454051

PLDI 2021

Concolic Program Repair

Ridwan Shariffdeen, Yannic Noller, Lars Grunske, Abhik Roychoudhury

42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 2021

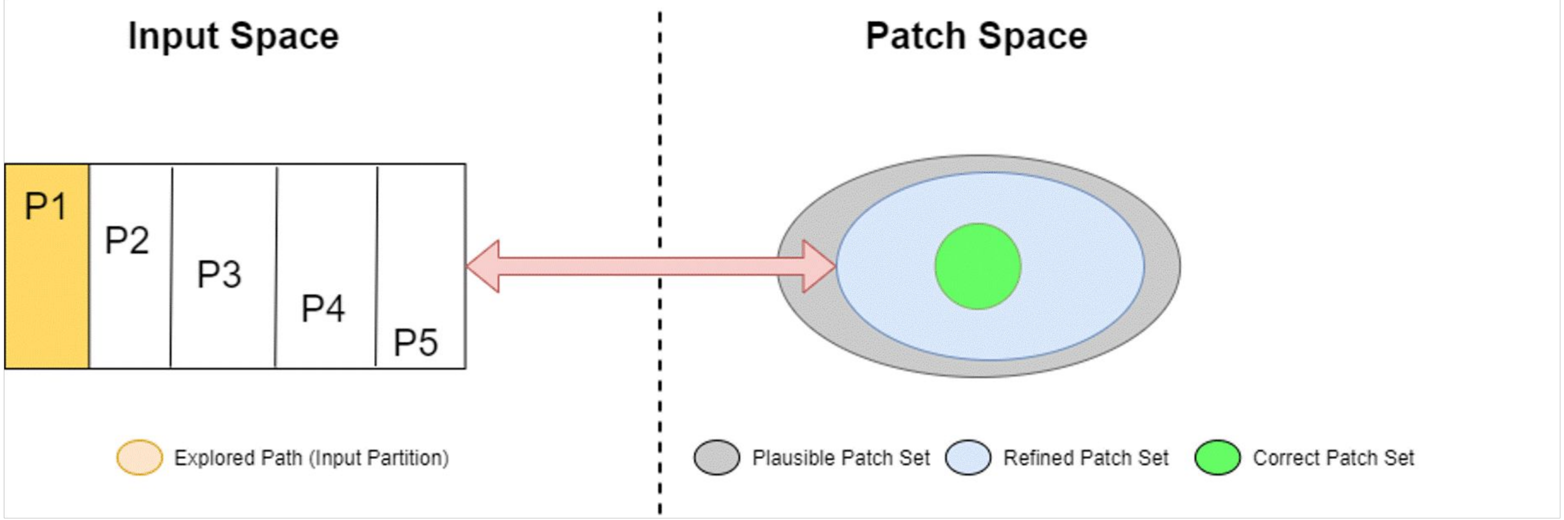
Artifacts Available acm

Artifacts Evaluated acm Functional

Artifacts Evaluated acm Reusable

Key Idea: Gradual Correctness

Detecting and discarding over-fitting patches via systematic co-exploration of the patch space and input space



Our Solution

semantic approach incl. **program synthesis**

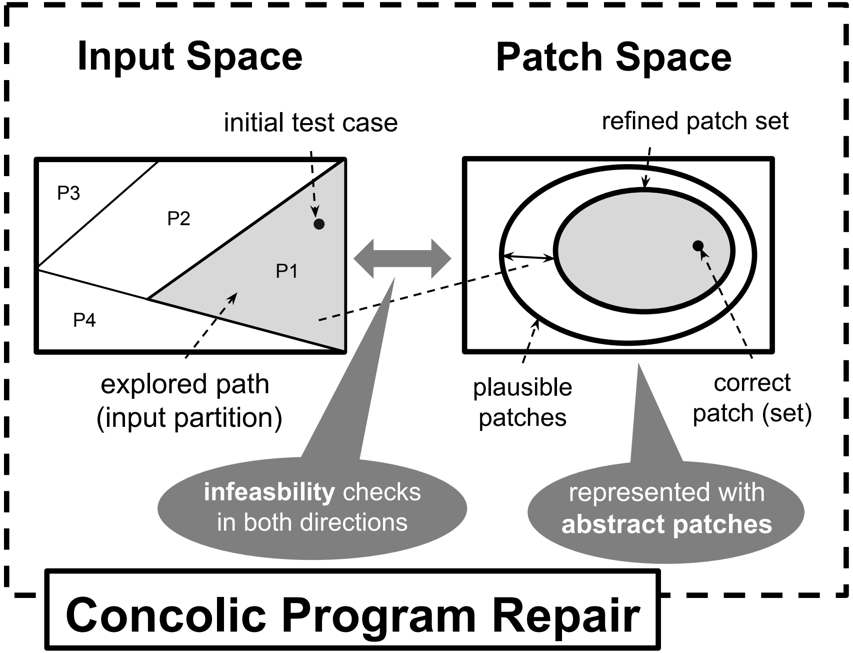
- avoids **non-compilable** patches
- provides **symbolic reasoning** capabilities

co-exploration of the **input** space and **patch** space

- prune **over-fitting** patches
- enables **gradual improvement**

user-provided specification

- to **reason** about **additional inputs**
- **key aspect** to handle absence of test cases



Patch Representation

.. **concrete** patches

```
x > 0
x > 1
x > 2
…
```

```
x + 1 > y
x - 1 > y
x + 2 > y
…
```

.. **abstract** patches

```
x > a, a ∈ [0, 10]
```

```
x + a > y, a ∈ [-10, 10]
```

Our notion of an **abstract patch** represents a **patch template** with **parameters.**

- **generate** and **maintain** smaller amount of patch candidates
- allows refinement instead of just discarding
- **subsumes** concrete patches

Abstract Patches

$$(\boldsymbol{\theta_\rho, T_\rho, \psi_\rho})$$

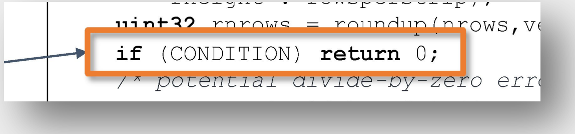
$\boldsymbol{X_\rho}$ is the set of **program variables**
$\boldsymbol{X} \subseteq X_\rho$ is the set of **input variables**
$\boldsymbol{A}$ is the set of **template parameters**

- $\theta_\rho(X_\rho, A)$ 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

1. patch is a **condition**



```
 [illegible] rowsperstrip);
uint32 rnrows = roundup(nrows,ve
if (CONDITION) return 0;
/* potential divide-by-zero erro
```

```
if (ρ)
  return 0;
```

$$\theta_\rho \coloneqq x > a$$
$$T_\rho = \tau_\rho(a) \coloneqq (a \geq -10)$$
$$\psi_\rho \coloneqq x > a$$

2. patch is a **right hand-side of an assignment**

```
…
y = ρ;
…
```

$$\theta_\rho \coloneqq x - a$$
$$T_\rho = \tau_\rho(a) \coloneqq (a \geq -10)$$
$$\psi_\rho \coloneqq (y = x - a)$$

New kind of Infeasibility

.. in the **input space**

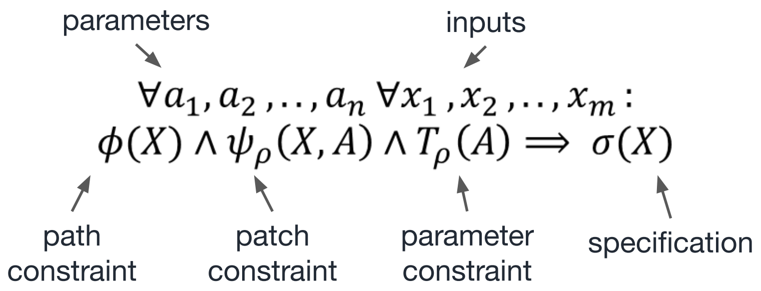
Path Reduction:
For every generated input, we check that there is one patch that can exercise the corresponding path. Otherwise, the path will not be explored.

For example:

$$\phi := x > 3 \land y > 5 \land \rho$$
$$\rho := (x = 0 \lor y = 0)$$

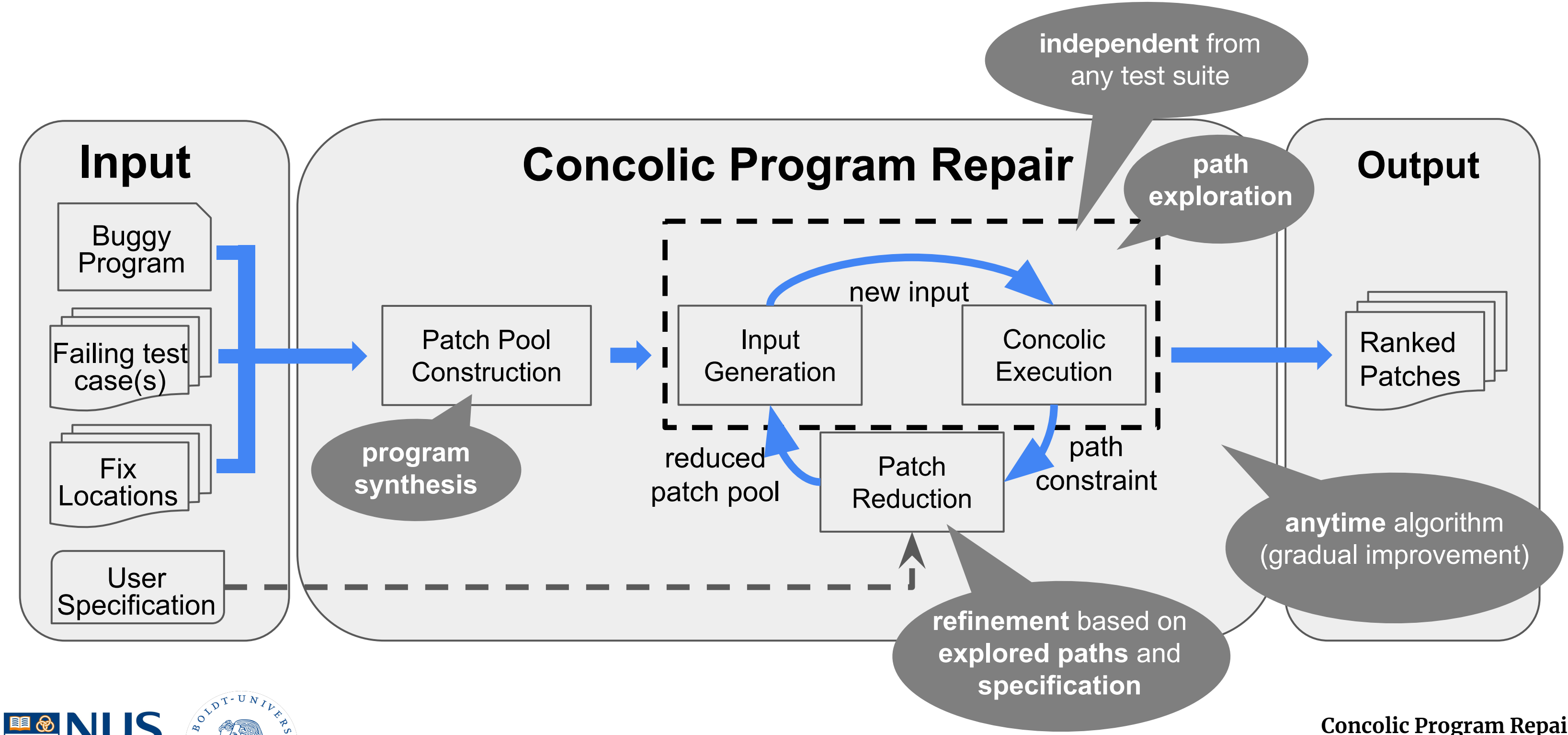
.. in the **patch space**

Patch Reduction:
If a patch allows inputs to exercise a path that violates the specification, we identify this as a patch that overfits the valid set of values and attempt to refine it.

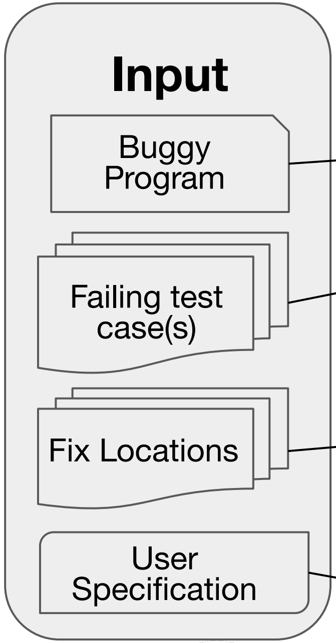


$$\forall a_1, a_2, .., a_n \ \forall x_1, x_2, .., x_m :$$
$$\phi(X) \land \psi_\rho(X, A) \land T_\rho(A) \implies \sigma(X)$$

Our Implementation



Illustrative Example



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

e.g., exploit as
TIFF picture

source location, (fix template),
synthesis components

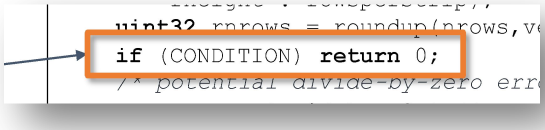
formula about correct
behavior in SMT format

```
........
static int
cvtRaster(TIFF* tif, uint32* raster, uint32 width...)
{
    uint32 y;
    tstrip_t strip = 0;
    tsize_t cc, acc;
    unsigned char* buf;
    uint32 rwidth = roundup(width, horizSubSampling);
    uint32 rheight = roundup(height, vertSubSampling);
    uint32 nrows = (rowsperstrip > rheight ?
        rheight : rowsperstrip);
    uint32 rnrows = roundup(nrows,vertSubSampling);
    if (CONDITION) return 0;
    /* potential divide-by-zero error */
    cc = rnrows*rwidth + 2 * ((rnrows*rwidth)
        / (horizSubSampling*vertSubSampling));
    ........
}
```

observation

(assert (= false (= observation 0)))

Illustrative Example

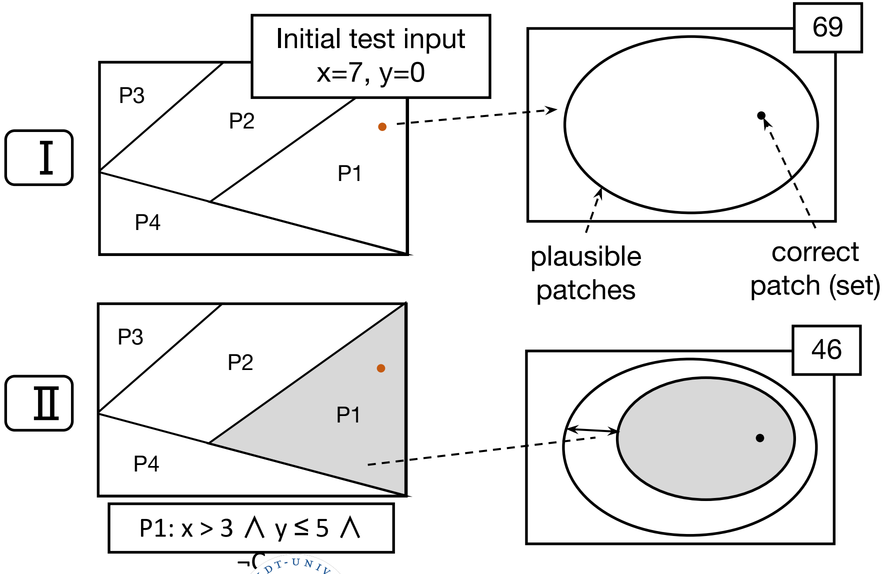


```
    imheight : rowsperstrip);
uint32 rnrows = roundup(nrows,ve
if (CONDITION) return 0;
/* potential divide-by-zero erro
```

```
x ≙ horizSubSampling
y ≙ vertSubSampling
C ≙ CONDITION
```

Input Space

Patch Space

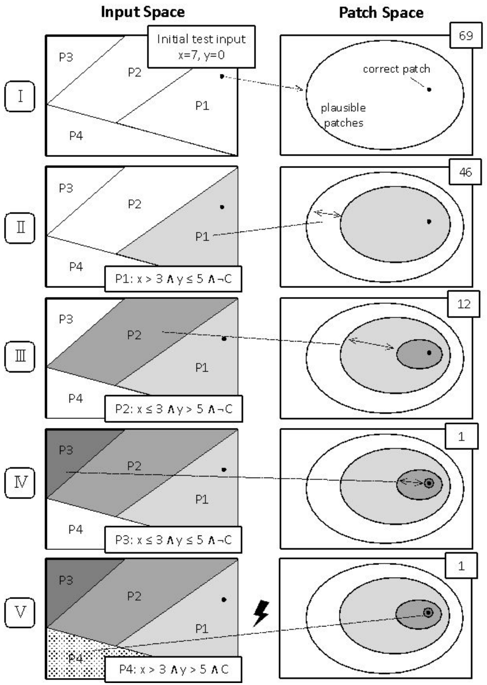


Patch Details

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

Illustrative Example



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

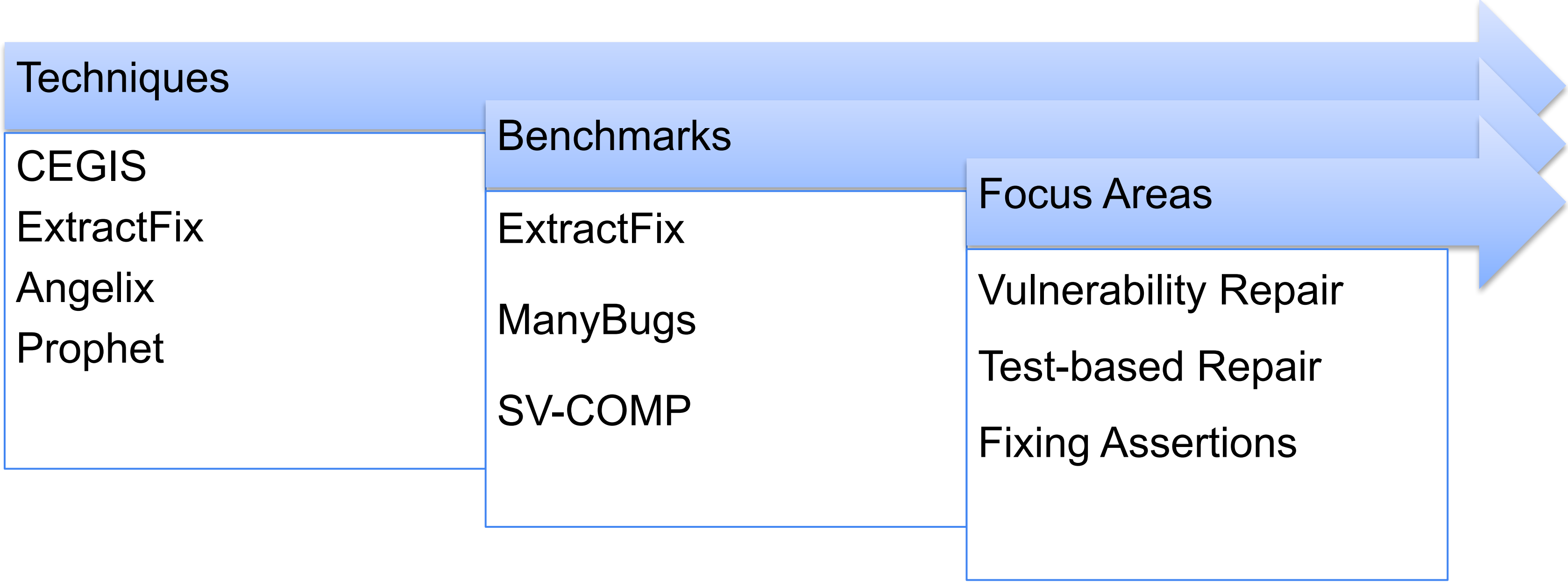
ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	a ≥ -10 ∧ a ≤ 4	15
2	y < b	b ≥ 1 ∧ b ≤ 10	10
3	x == a || y == b	b=0 ∧ a ≥ -10 ∧ a ≤ 10	21

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

Evaluation Setup



Techniques

CEGIS

ExtractFix

Angelix

Prophet

Benchmarks

ExtractFix

ManyBugs

SV-COMP

Focus Areas

Vulnerability Repair

Test-based Repair

Fixing Assertions

Evaluation Insights

Program	#Vul	Prophet	Angelix	ExtractFix	CPR
LibTIFF	11	1	0	6	**11**
Binutils	2	-	-	1	**2**
LibXML2	5	0	0	2	**5**
LibJPEG	4	1	-	2	**4**
FFmpeg	2	-	-	**2**	-
Jasper	2	0	0	**1**	**1**
Coreutils	4	0	-	2	**4**
Total	30	2	0	16	**27**

Number of correct patches generated for ExtractFix benchmark in 1h timeout

Up **74%** Patch Space Reduction

CPR **can gradually refine** the patch space via concolic exploration

CPR can be used for **test-guided general-purpose repair** and **security repair**

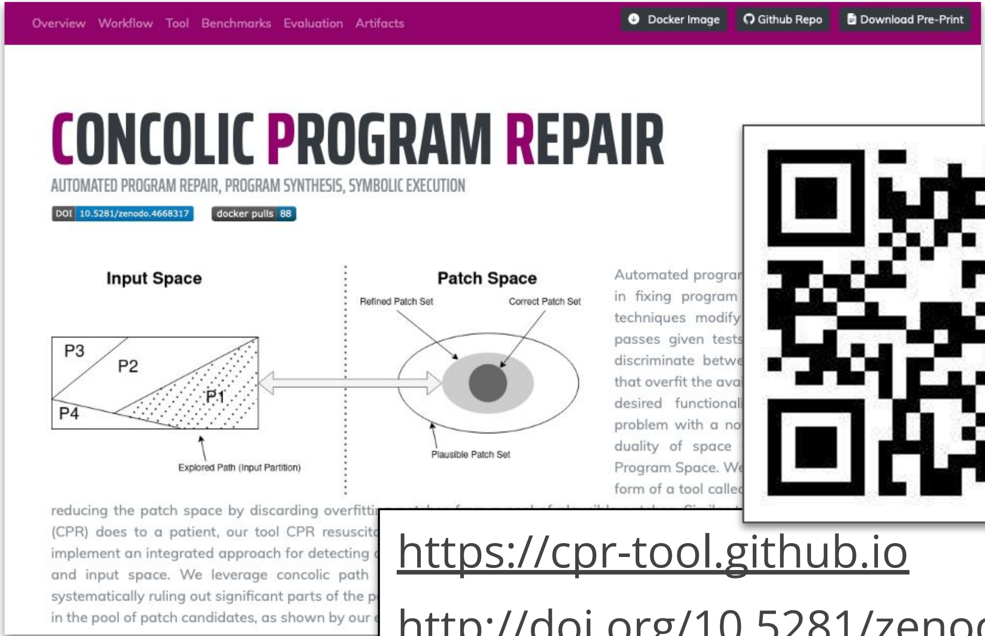
CPR is **more effective** than CEGIS wrt input and space exploration

Artifact

Artifacts Available

Artifacts Evaluated Functional

Artifacts Evaluated Reusable



CONCOLIC PROGRAM REPAIR

AUTOMATED PROGRAM REPAIR, PROGRAM SYNTHESIS, SYMBOLIC EXECUTION

DOI 10.5281/zenodo.4668317 docker pulls 88

Automated progran
in fixing program
techniques modify
passes given tests
discriminate betwe
that overfit the ava
desired functionali
problem with a no
duality of space
Program Space. We
form of a tool called

reducing the patch space by discarding overfitti
(CPR) does to a patient, our tool CPR resuscita
implement an integrated approach for detecting
and input space. We leverage concolic path
systematically ruling out significant parts of the p
in the pool of patch candidates, as shown by our

https://cpr-tool.github.io

http://doi.org/10.5281/zenodo.4668317

NUS National University of Singapore

HUMBOLDT-UNIVERSITÄT ZU BERLIN