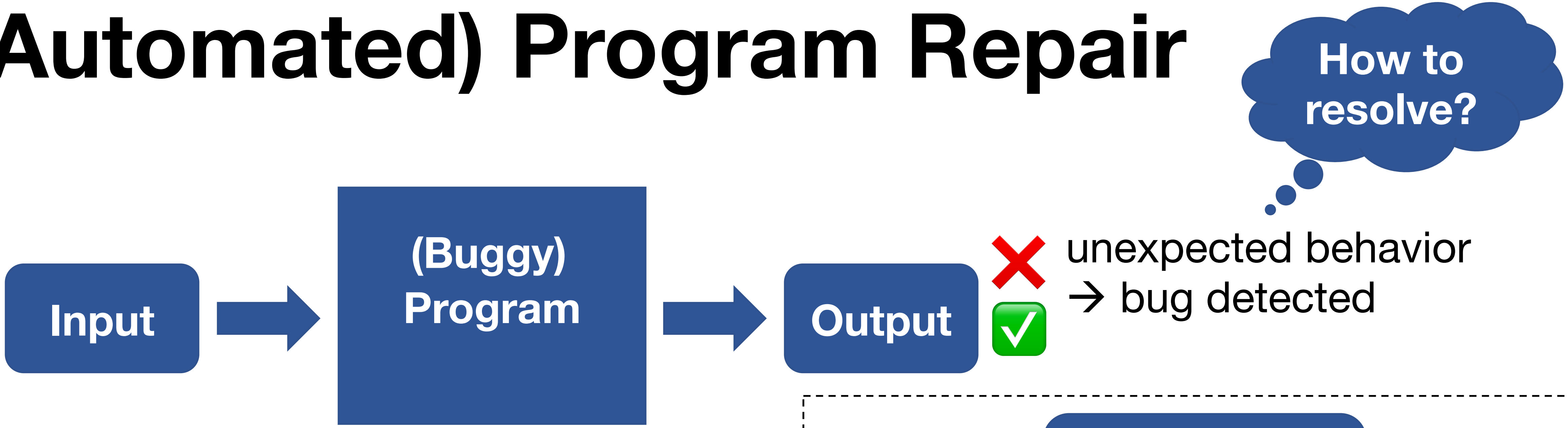
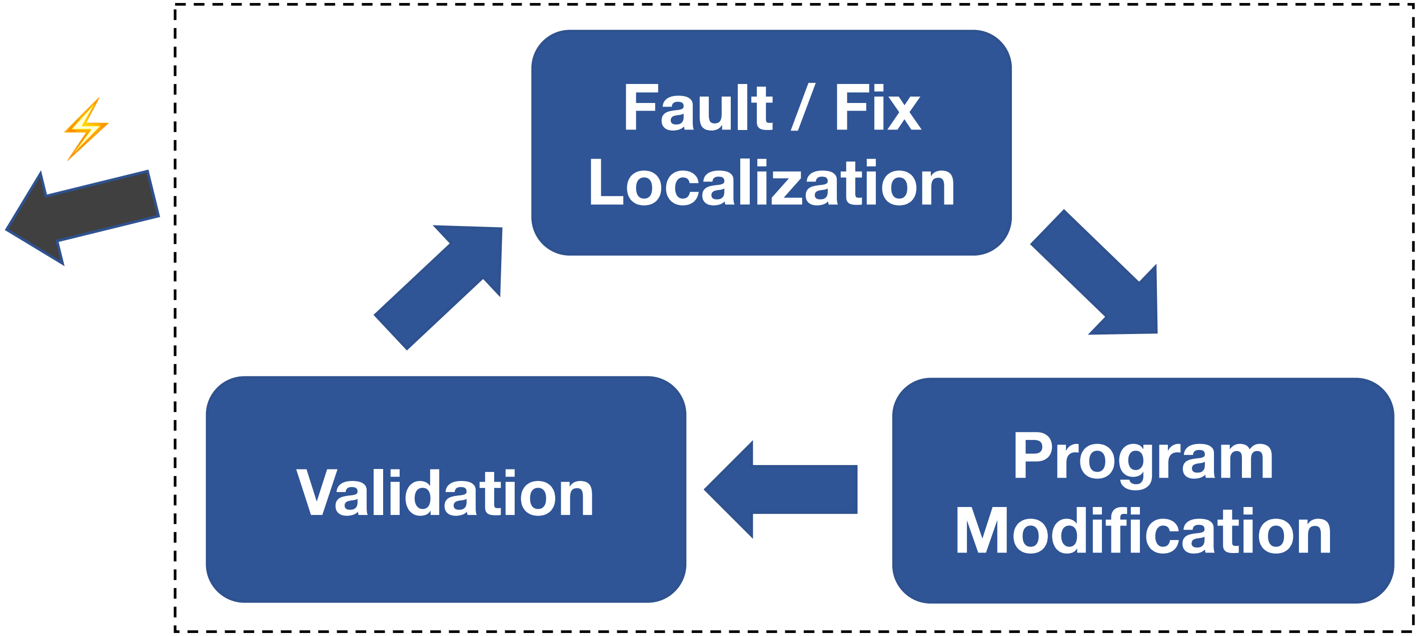
Concolic Program Repair

Yannic Noller | Research Talk

(Automated) Program Repair

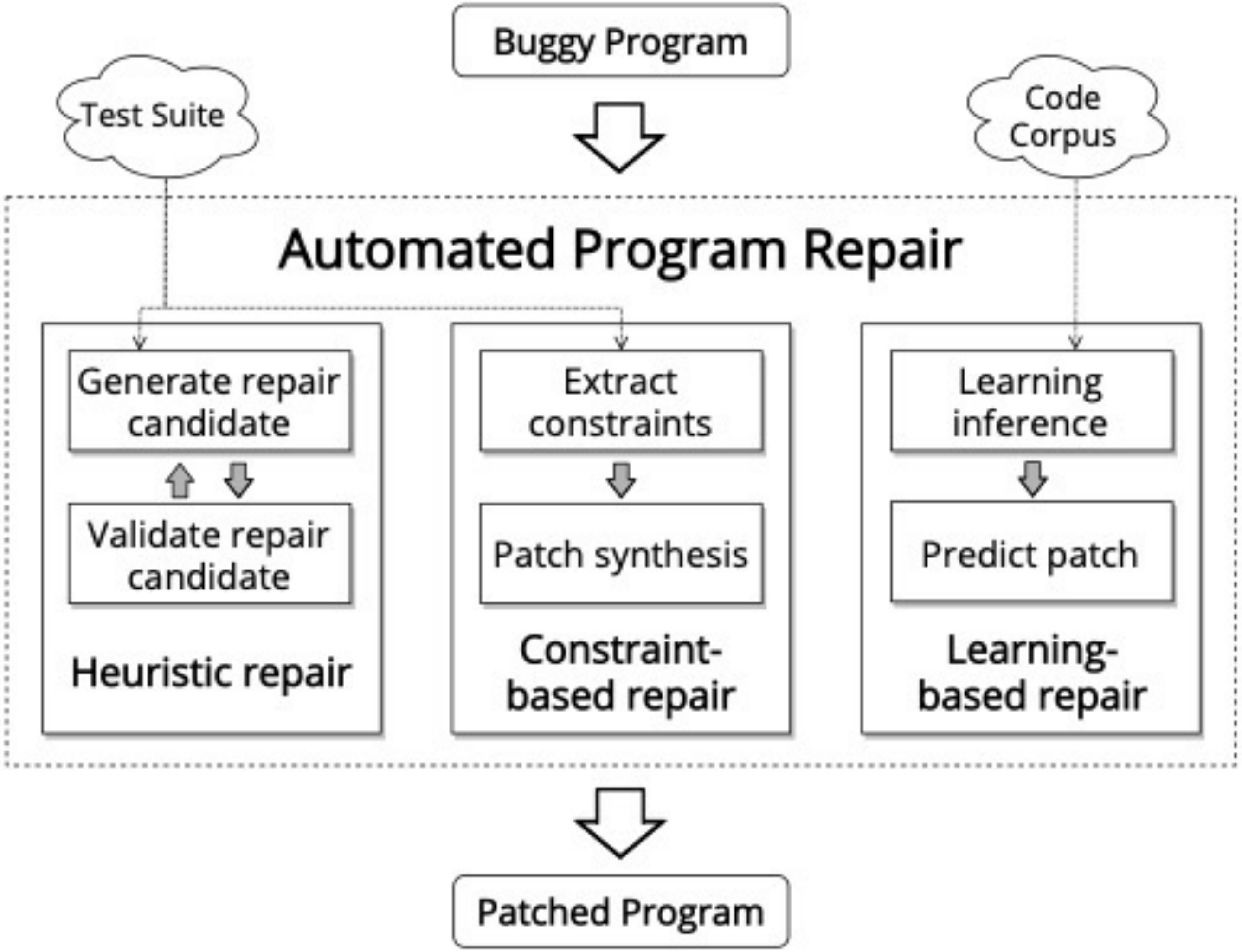




can be very tedious
… and time consuming

→ **Automated** program repair

State of the Art



Challenges

How to provide **high quality but few** patches?

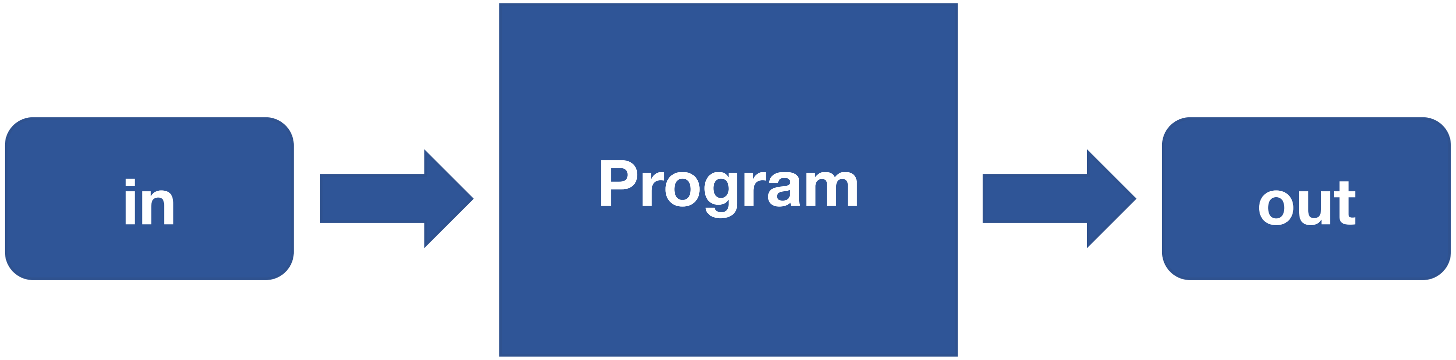
How to avoid **non-sensical** patches?

How to produce **less overfitting** patches?

How to repair bugs in the **absence** of many test cases?

Challenges

Input	Expected Output	Check
in=10	out=4	❌ ✅
in=100	out=25	❌




```
if (in == 10) {
   return 4;
}
...
```

.. but **overfitting** to test case

test cases are **only partial** specifications

Other **low-quality** patches:

```
if (((! (image->res_unit == 3)) && (! (image->res_unit == 3))))
```

```
if ( (! ((log_level && (! ((- 4) == 0))) && log_level)))
```

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**.

*Joint first authors

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

ACM ISBN 978-1-4503-8391-2/21/06...$15.00
https://doi.org/10.1145/3453483.3454051

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

PLDI 2021

International Conference on
Programming Language Design and Implementation

20-26 June 2021

Our Approach

Fresh **look on program repair**

semantic approach incl. **program synthesis**

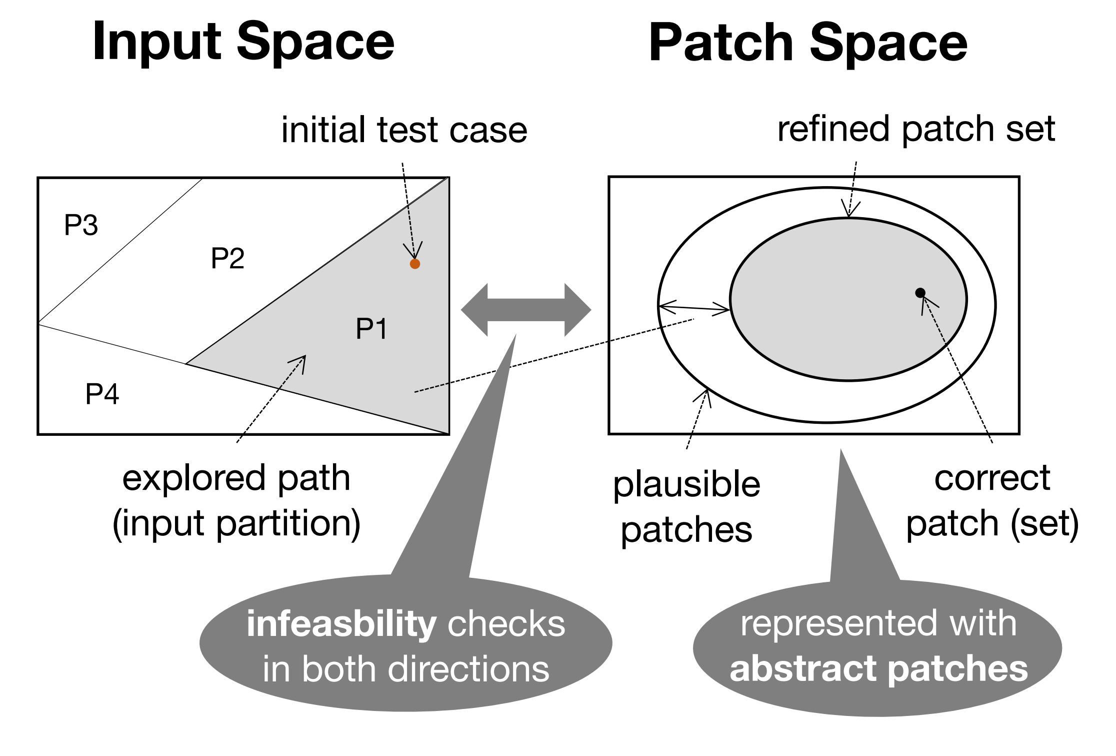
- avoids **non-compilable** patches
- provides **symbolic reasoning** cababilities

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

- prune **overfitting** patches
- enables **gradual improvement**

user-provided specification

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



Concolic Program Repair

Inputs to Concolic Program Repair

Input

- Buggy Program
- Failing test case(s)
- Fix Locations
- User Specification

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

Failing test case(s) → e.g., exploit as TIFF picture

Fix Locations → source location, (fix template), synthesis components

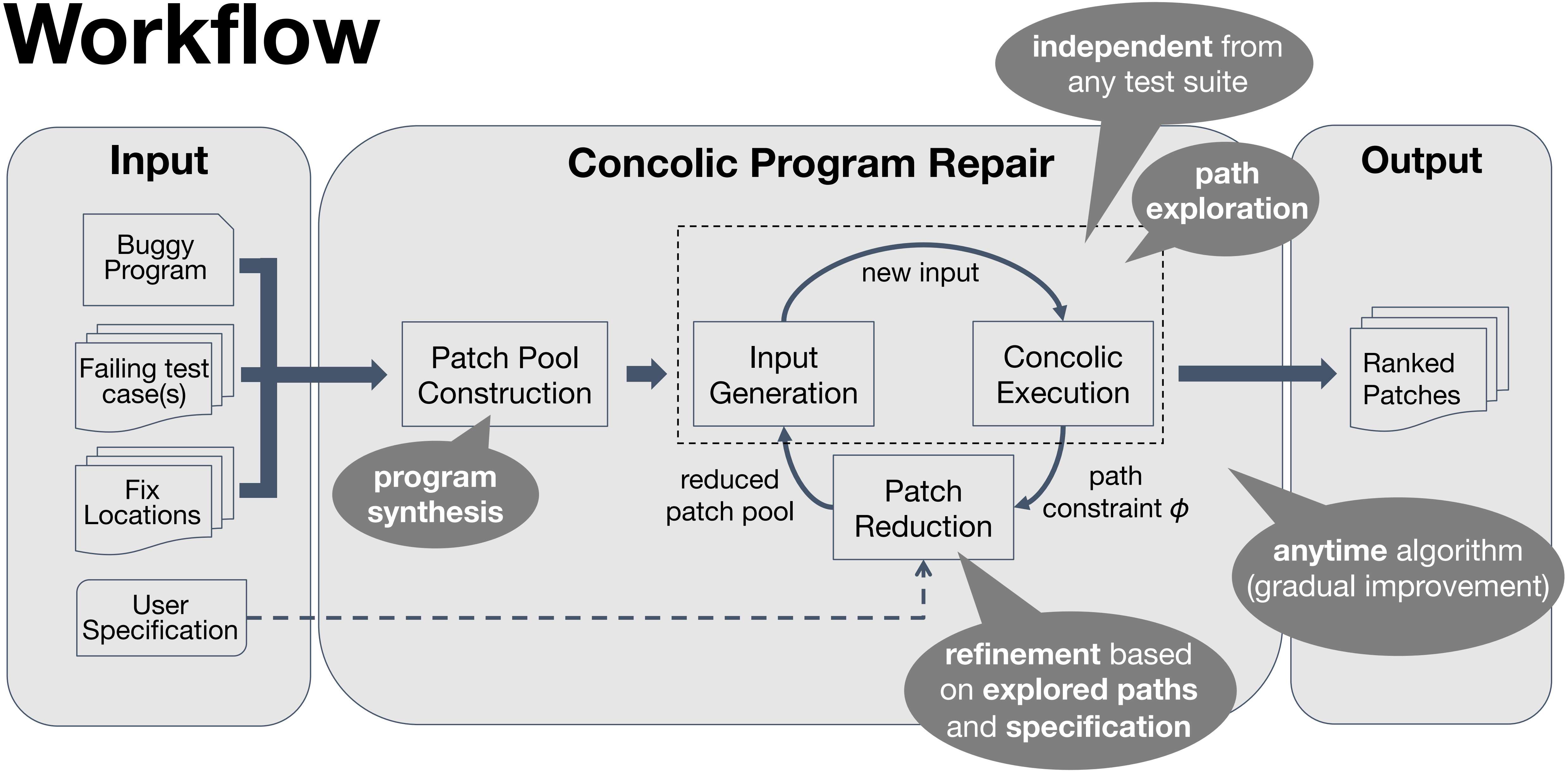
User Specification → formula about correct behavior in SMT format

```
........
static int
cvtRaster(TIFF* tif, uint32* raster, uint32 width, uint32 height)
{
    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)))

Workflow


independent from any test suite
Input
Concolic Program Repair
path exploration
Output
Buggy Program
Failing test case(s)
Fix Locations
User Specification
Patch Pool Construction
program synthesis
new input
Input Generation
Concolic Execution
reduced patch pool
Patch Reduction
path constraint ϕ
Ranked Patches
anytime algorithm (gradual improvement)
refinement based on explored paths and specification

Patch Representation

➡ **integer** and **boolean** expressions

.. **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

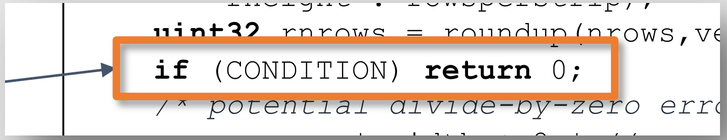
$$(\boldsymbol{\theta_\rho}, \boldsymbol{T_\rho}, \boldsymbol{\psi_\rho})$$

$\boldsymbol{X_\rho}$ 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

Examples

1. patch is a **condition**




```
if (ρ)
  return 0;
```

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

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

```
…
y = ρ;
…
```

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

Infeasability checks

.. 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.

parameters

inputs

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

path constraint

patch constraint

parameter constraint

specification

Patch Refinement

What we **want to have**:

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

What we **are checking for:**

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

$$\equiv \neg(\forall a_1, a_2, .., a_n \, \forall x_1, x_2, .., x_m : \neg(\phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A)) \vee \sigma(X))$$

$$\equiv \exists a_1, a_2, .., a_n \, \exists x_1, x_2, .., x_m : \phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A) \wedge \neg\sigma(X)$$

→ use **SMT solver** to retrieve a **model $\mathcal{M}$** to **refine** the **parameter constraint**

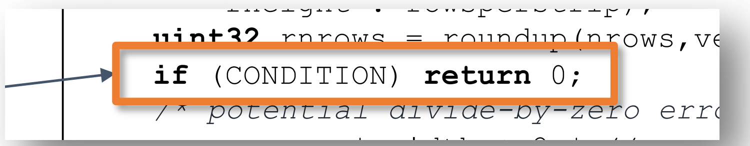
Example

```
........
static int
cvtRaster(TIFF* tif, uint32* raster, uint32 width, uint32 height)
{
    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));
    ........
}
```

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

```
x ≙ horizSubSampling
y ≙ vertSubSampling
```

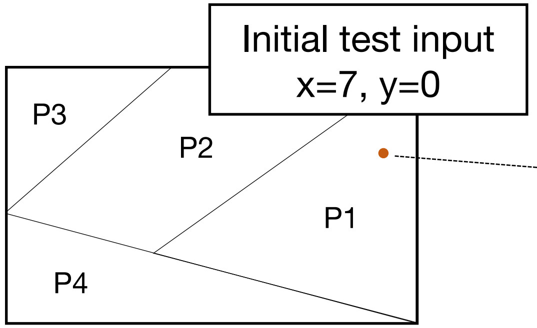
Example (2)



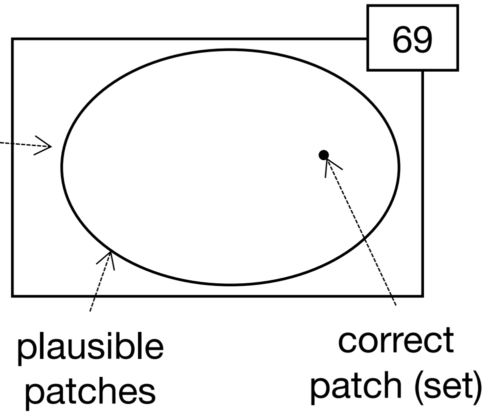

```
x ≙ horizSubSampling
y ≙ vertSubSampling
C ≙ CONDITION
```

Input Space

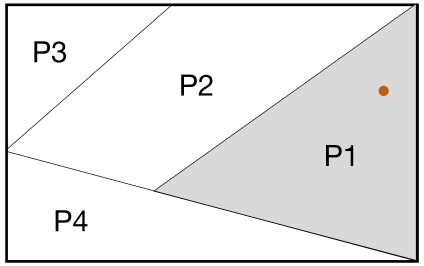
I



Patch Space



II

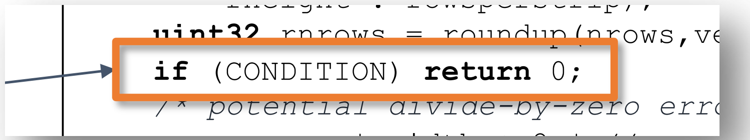


P1: $x > 3 \land y \leq 5 \land \neg C$

Patch Details

ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	$a \geq -10 \land a \leq 7$	18
2	y < b	$b \geq 1 \land b \leq 10$	10
3	x == a || y == b	$(a=7 \land b \geq -10 \land b \leq 10) \lor (b=0 \land a \geq -10 \land a \leq 10)$	41

Example (2) - Patch 1




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

$\exists a_1, a_2, .., a_n \, \exists x_1, x_2, .., x_m :$

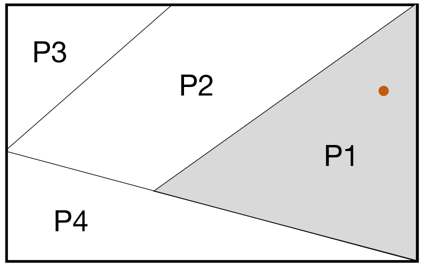
$$\phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A) \wedge \neg\sigma(X)$$

$x > 3 \wedge y \leq 5$	path constraint P1
$\wedge \ \neg(x \geq a) \wedge a \in [-10, 7]$	patch 1
$\wedge \ (x * y = 0)$	condition specification violation

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

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

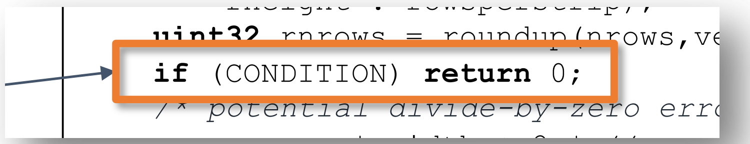
Ⅱ



P1: x > 3 ∧ y ≤ 5 ∧ ¬C

ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	a ≥ -10 ∧ a ≤ 4	15

Example (2) - Patch 2



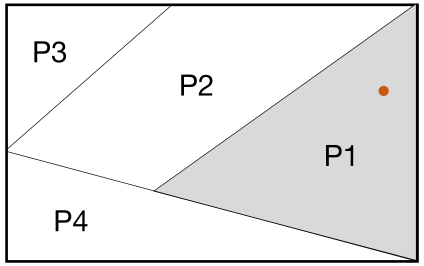
x ≙ horizSubSampling
y ≙ vertSubSampling
C ≙ CONDITION

$\exists a_1, a_2, .., a_n \, \exists x_1, x_2, .., x_m :$

$$\phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A) \wedge \neg\sigma(X)$$

$x > 3 \wedge y \leq 5$	path constraint P1
$\wedge\ \neg(y < b) \wedge b \in [1, 10]$	patch 2
$\wedge\ (x * y = 0)$	condition specification violation

Ⅱ



P1: x > 3 ∧ y ≤ 5 ∧ ¬C

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

Example (2) - Patch 3

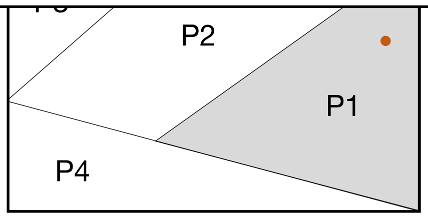
```
uint32 rnrows = roundup(nrows,ve
if (CONDITION) return 0;
/* potential divide-by-zero erro
```

x ≙ horizSubSampling
y ≙ vertSubSampling
C ≙ CONDITION

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

$x > 3 \wedge y \leq 5$	path constraint P1
$\wedge \neg(x = a \vee y = b)$	patch 3
$\wedge \ (a = 7 \wedge b \in [-10, 10] \vee b = 0 \wedge a \in [-10, 10])$	
$\wedge \ \ (x * y = 0)$	condition specification violation

II



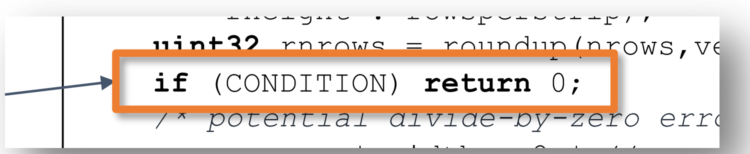
P1: x > 3 ∧ y ≤ 5 ∧ ¬C

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

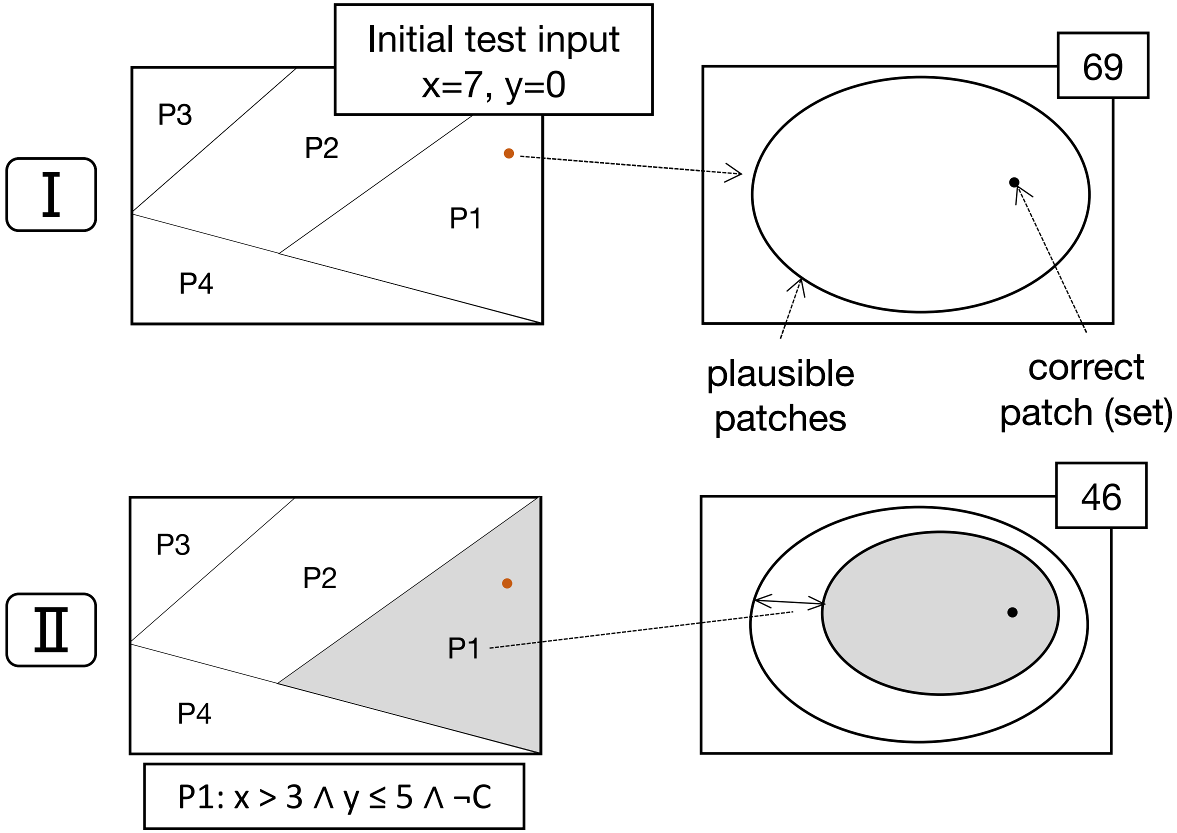
Example (2)



x ≜ horizSubSampling
y ≜ vertSubSampling
C ≜ CONDITION

Input Space

Patch Space

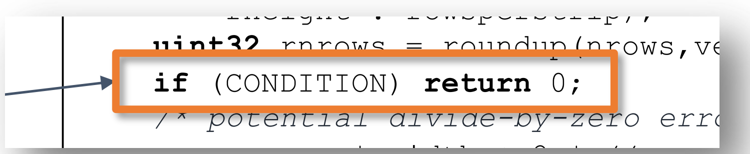


Patch Details

ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	$a \geq -10 \land a \leq 7$	18
2	y < b	$b \geq 1 \land b \leq 10$	10
3	x == a || y == b	$(a=7 \land b \geq -10 \land b \leq 10) \lor (b=0 \land a \geq -10 \land a \leq 10)$	41

ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	$a \geq -10 \land a \leq 4$	15
2	y < b	$b \geq 1 \land b \leq 10$	10
3	x == a || y == b	$b=0 \land a \geq -10 \land a \leq 10$	21

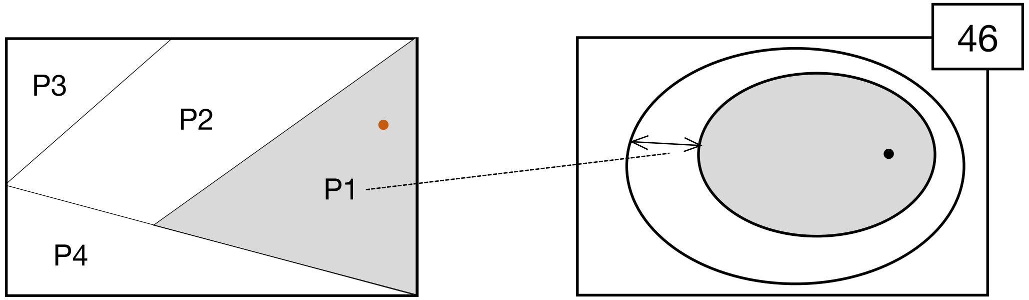
Example (3)



x ≙ horizSubSampling
y ≙ vertSubSampling
C ≙ CONDITION

Input Space | **Patch Space** | **Patch Details**

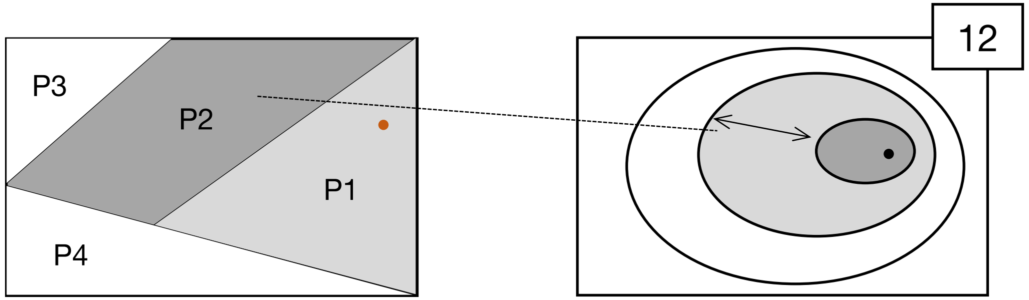
II



P1: $x > 3 \land y \leq 5 \land \neg C$

ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	$a \geq -10 \land a \leq 4$	15
2	y < b	$b \geq 1 \land b \leq 10$	10
3	x == a || y == b	$b=0 \land a \geq -10 \land a \leq 10$	21

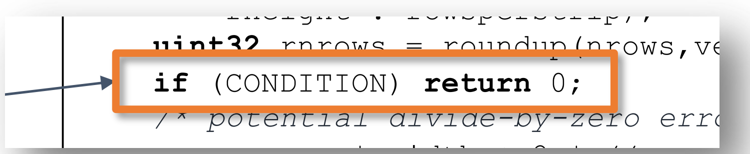
III



P2: $x \leq 3 \land y > 5 \land \neg C$

ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	$a \geq -10 \land a \leq 0$	11
~~2~~	~~y < b~~	False	0
3	x == a || y == b	$a = 0 \land b = 0$	1

Example (4)



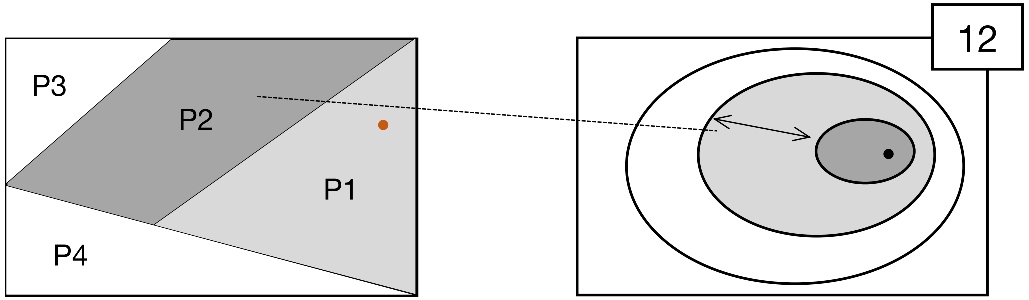

```
x ≙ horizSubSampling
y ≙ vertSubSampling
C ≙ CONDITION
```

Input Space

Patch Space

Patch Details

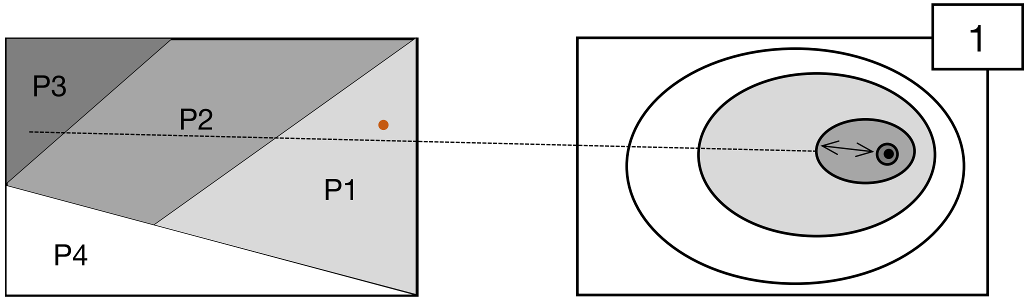
Ⅲ



P2: $x \le 3 \land y > 5 \land \neg C$

ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	$a \ge -10 \land a \le 0$	11
~~2~~	~~y < b~~	False	0
3	x == a || y == b	$a = 0 \land b = 0$	1

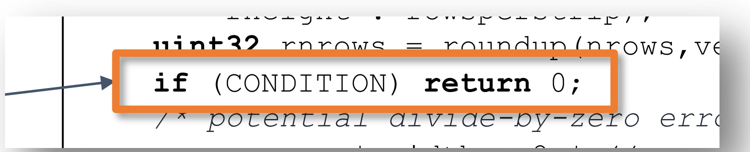
Ⅳ



P3: $x \le 3 \land y \le 5 \land \neg C$

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

Example (5)



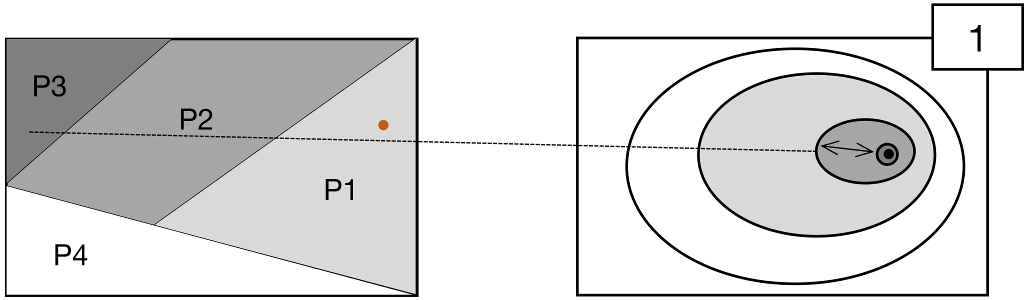

```
x ≙ horizSubSampling
y ≙ vertSubSampling
C ≙ CONDITION
```

Input Space

Patch Space

Patch Details

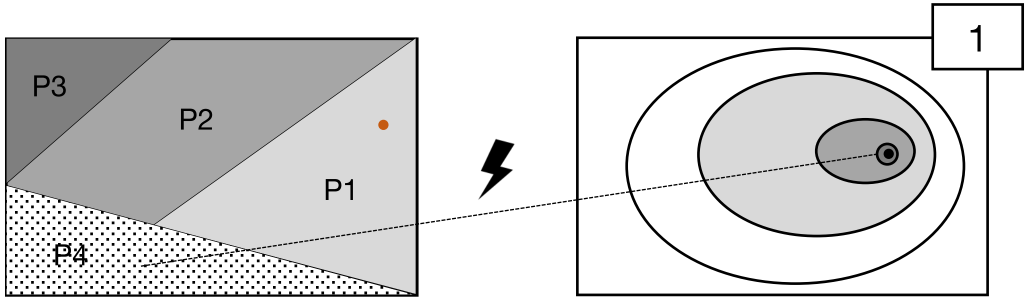
IV



P3: $x \leq 3 \land y \leq 5 \land \neg C$

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

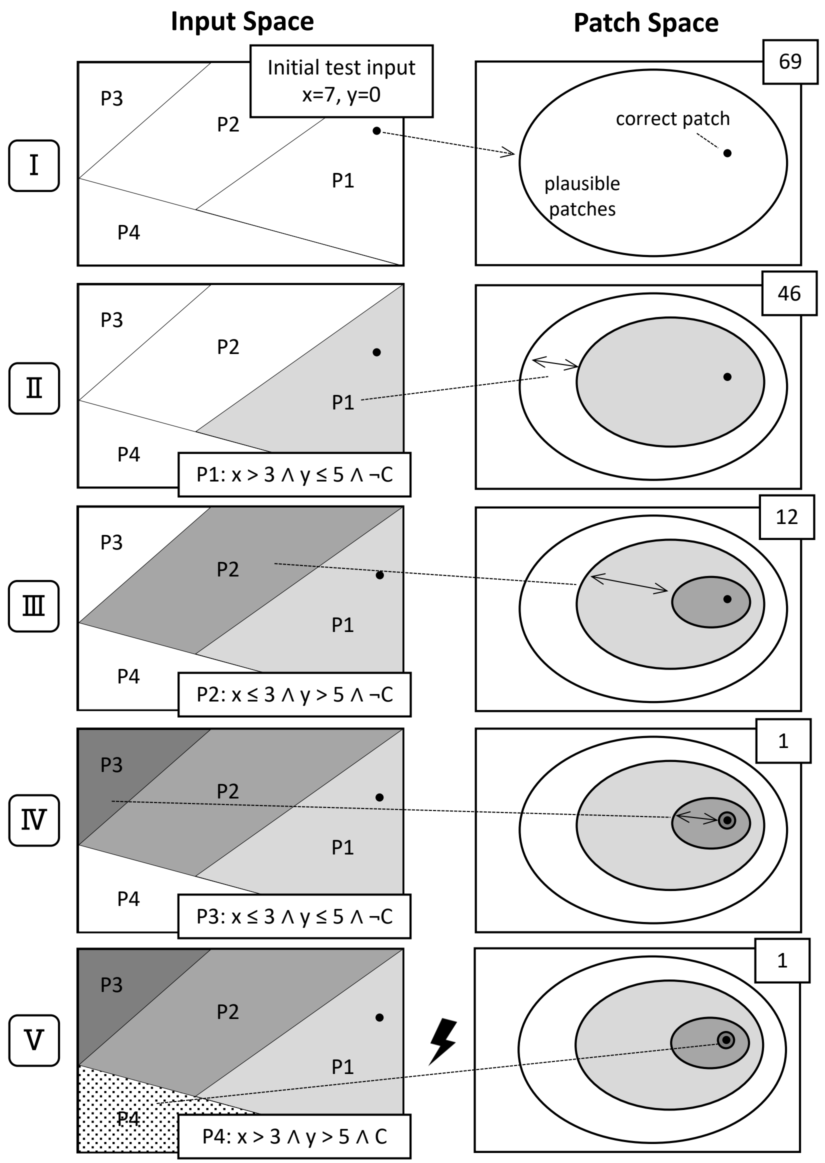
V



P4: $x > 3 \land y > 5 \land C$

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

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



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

Patch space refinement based on the exploration of input space.

Rule out parts of the **input space**, which **contradicts** with the patch space.

Abstract patches vs. **concrete** patches

Gradual improvement

Evaluation

Tools/Techniques

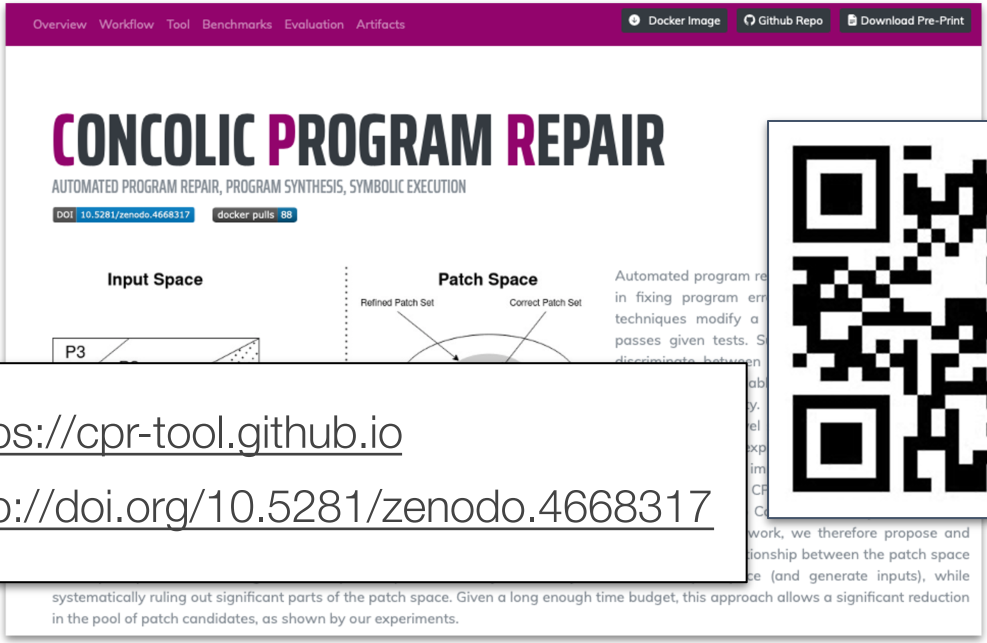
- CEGIS
- ExtractFix
- Angelix
- Prophet

Benchmarks

- ExtractFix
- ManyBugs
- SV-COMP

Repair Areas

- Security Vulnerability Repair
- General Test-based Repair
- Fixing Logical Errors



https://cpr-tool.github.io

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

Comparison with existing APR

```
static int jpc_dec_parseopts (..) {
   ------------------------------
   - return 0;
   + return opts->maxlyrs;
}
```

```
static int jpc_dec_process_siz(..) {
   ------------------------------
   - if (!(dec->cmpts = jas_malloc(dec->numcomps *
      sizeof(jpc_dec_cmpt_t)))) {
   + if ((!(dec->cmpts = jas_malloc(dec->numcomps *
      sizeof(jpc_dec_cmpt_t)))) || (1)) {
   ------------------------------
}
```

Patches generated by existing APR

Overfitting patches

Non-sensical patches

Comparison with existing APR (2)

CVE-2016-8691

```
static int jpc_siz_getparms(…) {
   ------------------------------
   + if (siz->comps[i].hsamp == 0)
      return -1;
   -------------------------------
}
```

CPR generates correct Patch

Initial Patch Space: 260

Refined Patch Space: 96

Refinement: 63%

Rank of Correct Patch: 1

Evaluation Insights

ID	Buggy Program		Components		Our CEGIS Implementation					CPR					
	Project	Bug ID	General	Custom	$\|P_{Init}\|$	$\|P_{Final}\|$	Ratio	ϕ_E	Correct?	$\|P_{Init}\|$	$\|P_{Final}\|$	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

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

Up **74%** Patch Space Reduction

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

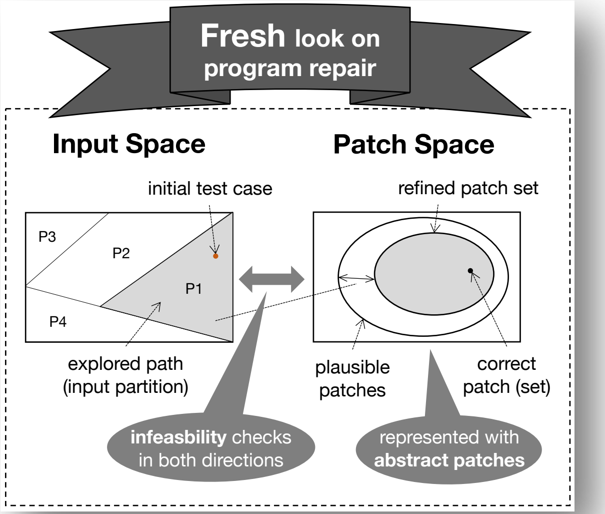
CPR can be used for **test-guided general-purpose repair** and **security repair**

CPR provides **highly ranked** patches

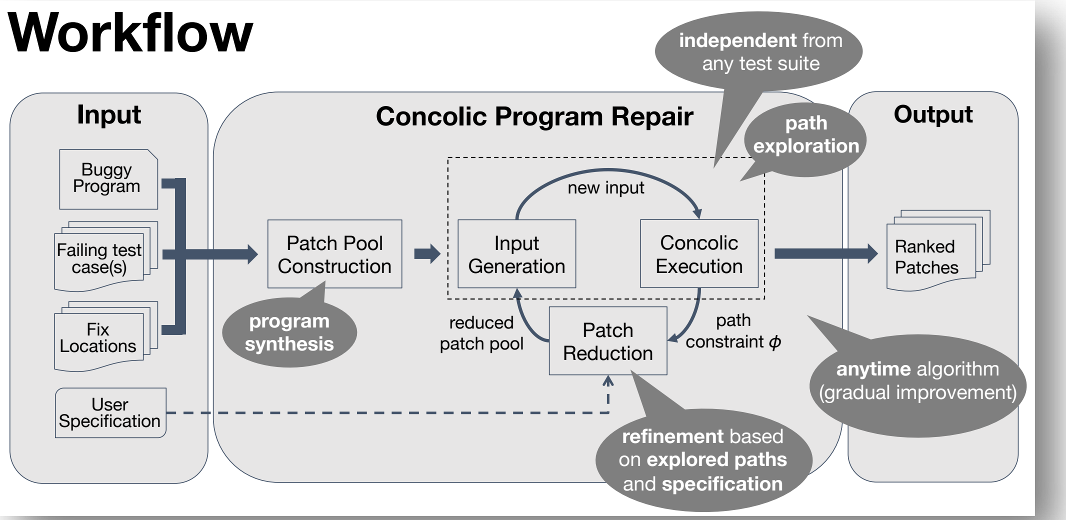
Concolic Program Repair

Challenges

- How to provide **high quality but few** patches?
- How to avoid **non-sensical** patches?
- How to produce **less overfitting** patches?
- How to repair bugs in the **absence** of many test cases?



Workflow



Infeasability checks

.. 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.

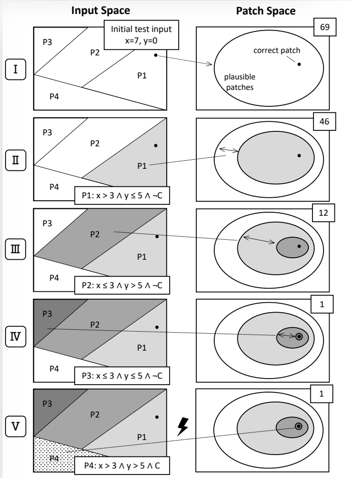
For example:

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

.. 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.

$$\underbrace{\forall a_1, a_2, .., a_n}_{\text{parameters}} \underbrace{\forall x_1, x_2, .., x_m}_{\text{input variables}} :$$
$$\underbrace{\phi(X)}_{\text{path constraint}} \land \underbrace{\psi_\rho(X, A)}_{\text{patch constraint}} \land \underbrace{T_\rho(A)}_{\text{parameter constraint}} \implies \underbrace{\sigma(X)}_{\text{specification}}$$



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