Synthesis of Software Patches Using Symbolic Execution

Sergey Mechtaev

University College London
Department of Computer Science
s.mechtaev@ucl.ac.uk

COW62, London, UK
January 20, 2020

Examples of generated patches

Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. Mechtaev et al. ICSE'16

Developer patch for Heartbleed vulnerability in OpenSSL

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
  return 0;
if (hbtype == TLS1_HB_REQUEST) {
  ...
  buffer = OPENSSL_malloc(1 + 2 + payload + padding);
  ...
}
```

Automatically generated patch

```
if (hbtype == TLS1_HB_REQUEST
    && payload + 18 < s->s3->rrec.length) {
  ...
  buffer = OPENSSL_malloc(1 + 2 + payload + padding);
  ...
}
```

Generate-and-validate approach

Buggy program P

```
scanf("%d",&x);
if (x - 1 > 0)
  printf("1");
else
  printf("0");
```

Test

$P(1) \mapsto$ “0”, expected “1”.

Repair algorithm — enumerate candidates

for *patch* $c \in$ *Candidates* **do**
  $P' := Apply(c, P)$
  $Validate(P'(1) = \text{“1”})$
end

$$Candidates = \{$$
$$x - 1 > 0 \mapsto x - 2 > 0,$$
$$x - 1 > 0 \mapsto x + 1 > 0,$$
$$\ldots \}$$

PHP interpreter test execution $\approx$ 10 sec. Substitutions to consider $\approx 10^5$. Algorithm execution time $\approx$ 11 days (for a single condition).

Semantic approach

SemFix, Nopol, SPR, Angelix, etc.

Buggy program P

```
scanf("%d",&x);
if (x - 1 > 0)
  printf("1");
else
  printf("0");
```

Test

$P(1) \mapsto$ “0”, expected “1”.

Repair algorithm — enumerate branches

$P' := Apply(x - 1 > 0 \mapsto True, P)$

$Validate(P'(1) = \text{“1”})$ // PASS

$P'' := Apply(x - 1 > 0 \mapsto False, P)$

$Validate(P''(1) = \text{“1”})$ // FAIL

for *expression* $e \in Substitutions$ **do**

  $Validate(e = True)$

end

$$Substitutions = \{ \; x - 2 > 0, \; x + 1 > 0, \; \ldots \}$$

Semantic approach: path explosion

Buggy program P

```
scanf("%d",&x);
for (i=0;i<10;i++)
  if (x - i > 0)
    printf("1");
  else
    printf("0");
```

Test

$P(1) \mapsto$ “1000000000”,
expected “1110000000”.

Repair algorithm — enumerate branches

for $s \in Sequences$ **do**
  $P' := Apply(x - i > 0 \mapsto s, P)$
  $Validate(P'(1) = \text{“1110000000”})$
end
for *expression* $e \in Substitutions$ **do**
  $Validate(e = True)$
end

$Sequences = \{$
$\{True, True, True, ...\},$
$\{False, True, True, ...\},$
$...(\text{totally, } 2^{10})\ \}$
$Substitutions = \{$
$x - 2 > 0,$
$x + 1 > 0, \quad ...\ \}$

Semantic approach: equivalence classes

Buggy program P

```
scanf("%d",&x);
for (i=0;i<10;i++)
  if (x - i > 0)
    printf("1");
  else
    printf("0");
```

Test

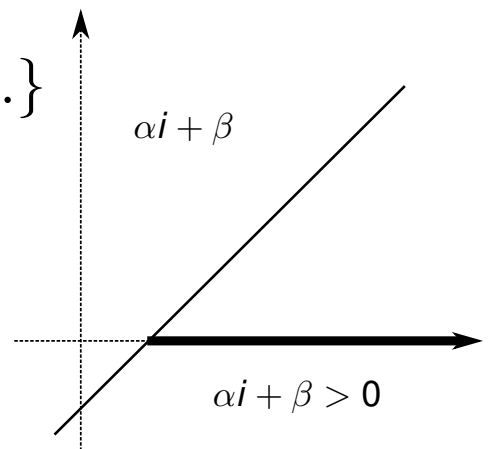
$P(1) \mapsto$ "1000000000",
expected "1110000000".

$\langle Candidate \rangle$ ::= $\langle Term \rangle$ '>' 0 | $\langle Term \rangle$ '==' 0 | $\langle Term \rangle$ '!=' 0

$\langle Term \rangle$ ::= $\langle Var \rangle$ | $\langle Constant \rangle$
| $\langle Term \rangle$ + $\langle Term \rangle$ | $\langle Term \rangle$ - $\langle Term \rangle$ | $\langle Constant \rangle$ * $\langle Term \rangle$

Equivalence classes

$\{True, True, True, True, ...\} \mapsto \{x > 0, ...\}$
$\{False, True, True, True, ...\} \mapsto \{...\}$
$\{True, False, True, True, ...\} \mapsto \{...\}$
...(totally, 2^{10})



Semantic approach: equivalence classes

Buggy program P

```
scanf("%d",&x);
for (i=0;i<10;i++)
  if (x - i > 0)
    printf("1");
  else
    printf("0");
```

Test

$P(1) \mapsto$ "1000000000",
expected "1110000000".

$\langle Candidate \rangle$ ::= $\langle Term \rangle$ '>' 0 | $\langle Term \rangle$ '==' 0 | $\langle Term \rangle$ '!=' 0

$\langle Term \rangle$ ::= $\langle Var \rangle$ | $\langle Constant \rangle$
| $\langle Term \rangle$ + $\langle Term \rangle$ | $\langle Term \rangle$ - $\langle Term \rangle$ | $\langle Constant \rangle$ * $\langle Term \rangle$

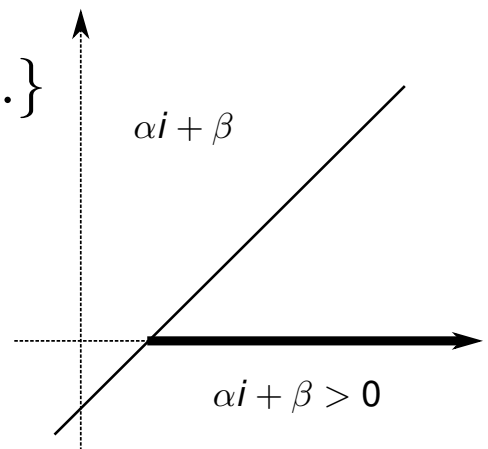
Equivalence classes

$\{True, True, True, True, ...\} \mapsto \{x > 0, ...\}$

$\{False, True, True, True, ...\} \mapsto \{...\}$

$\{True, False, True, True, ...\} \mapsto \emptyset$

...(totally, 2^{10})



Semantic approach: equivalence classes

Buggy program P

```
scanf("%d",&x);
for (i=0;i<10;i++)
  if (x - i > 0)
    printf("1");
  else
    printf("0");
```

Test

$P(1) \mapsto$ "1000000000",
expected "1110000000".

$\langle Candidate \rangle$ ::= $\langle Term \rangle$ '>' 0 | $\langle Term \rangle$ '==' 0 | $\langle Term \rangle$ '!=' 0

$\langle Term \rangle$ ::= $\langle Var \rangle$ | $\langle Constant \rangle$
| $\langle Term \rangle$ + $\langle Term \rangle$ | $\langle Term \rangle$ - $\langle Term \rangle$ | $\langle Constant \rangle$ * $\langle Term \rangle$

Equivalence classes

$\{True, True, True, True, ...\} \mapsto \{x > 0, ...\}$

$\{False, True, True, True, ...\} \mapsto \{...\}$

$\{True, False, True, True, ...\} \mapsto \emptyset$

...(totally, 2^{10})

Validate 20 sequences

Semantic approach (test-equivalence analysis)

Test-equivalence analysis for automatic patch generation. Mechtaev et al. TOSEM'18

Buggy program P

```
scanf("%d",&x);
for (i=0;i<10;i++)
  if (x - i > 0)
    printf("1");
  else
    printf("0");
```

Test

$P(1) \mapsto$ "1000000000",
expected "1110000000".

Repair algorithm

for *patch* $c \in$ *Candidates* **do**

$P' := Apply(c, P)$

$[c] := Validate(P'(1) = \text{"1110..."})$

Candidates $:=$ *Candidates* $\setminus [c]$

end

$$Candidates = \{$$
$$x - i > 0 \mapsto x - 2 > 0,$$
$$x - i > 0 \mapsto x + i > 0,$$
$$\dots \}$$

Test-equivalence analysis example

112487 modifications, 256 executions paths

```
if (tif->tif_rawcc > 0 && tif->tif_rawcc != orig_rawcc
    && (tif->tif_flags & TIFF_BEENWRITING)!= 0
    && !TIFFFlushData1(tif)) {
      TIFFErrorExt(tif->tif_clientdata, module,
        "Error flushing data before directory write");
    return (0);
  }
```

5 test-equivalence classes

```
((tif->tif_rawcc > 0) && (tif->tif_rawcc != orig_rawcc))
  || (tif->tif_flags & TIFF_BEENWRITING)

((tif->tif_rawcc > 0) || (tif->tif_rawcc != orig_rawcc))
  && (tif->tif_flags & TIFF_BEENWRITING)

((tif->tif_rawcc == 0) || (tif->tif_rawcc != orig_rawcc))
  && (tif->tif_flags & TIFF_BEENWRITING)

(((tif->tif_rawcc > 0) && (tif->tif_rawcc != orig_rawcc))
  && (tif->tif_flags & TIFF_BEENWRITING)) || (imagedone >= orig_rawcc)

(((tif->tif_rawcc > 0) && (tif->tif_rawcc != orig_rawcc))
  && (tif->tif_flags & TIFF_BEENWRITING)) || (tif->tif_flags >= 74)
```

Propagation of changes

Enumerate $\{True, False\}$

```
scanf("%d",&x);

if (x - 1 > 0)
  printf("1");
else
  printf("0");
```

Enumerate 2^{32} values

```
scanf("%d",&x);
int t = x - 1;
if (t>0)
  printf("1");
else
  printf("0");
```

SemFix

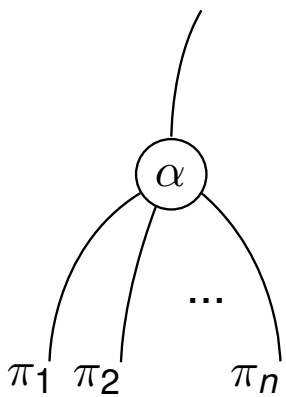
SemFix: Program Repair via Semantic Analysis. Nguyen et al. ICSE'13

```
scanf("%d",&x);
int t = α;
if (t>0)
  printf("1");
else
  printf("0");
```

Path conditions:

$$\pi_1 := \alpha > 0 \wedge \mathit{output} = \text{“1”}$$

$$\pi_2 := \alpha \leq 0 \wedge \mathit{output} = \text{“0”}$$



Specification:

$$\exists e \in \mathit{Term}.\ \bigvee_i \pi_i[\alpha \mapsto e] \wedge (\mathit{output} = \mathit{expected})$$

where *Term* — a set of well-formed expressions, α — symbolic variable, π_i — path condition.

SemFix: path explosion

```
scanf("%d",&x);
for (i=0;i<10;i++) {
  int t = α;
  if (t>0)
    printf("1");
  else
    printf("0");
}
```

Satisfiabile path conditions π_i (totally, $2^{10} = 1024$):

$$\alpha_1 > 0 \wedge \alpha_2 > 0 \wedge \alpha_3 > 0 \wedge \alpha_4 > 0 \wedge \ldots$$
$$\neg(\alpha_1 > 0) \wedge \alpha_2 > 0 \wedge \alpha_3 > 0 \wedge \alpha_4 > 0 \wedge \ldots$$
$$\alpha_1 > 0 \wedge \neg(\alpha_2 > 0) \wedge \alpha_3 > 0 \wedge \alpha_4 > 0 \wedge \ldots$$

Synthesis specification:

$$\exists e \in \mathit{Term}.\ \bigvee_i \pi_i[\alpha \mapsto e] \wedge (\mathit{output} = \mathit{expected})$$

$\langle \mathit{Term} \rangle$ ::= $\langle \mathit{Var} \rangle$ | $\langle \mathit{Constant} \rangle$
| $\langle \mathit{Term} \rangle$ + $\langle \mathit{Term} \rangle$ | $\langle \mathit{Term} \rangle$ - $\langle \mathit{Term} \rangle$ | $\langle \mathit{Constant} \rangle$ * $\langle \mathit{Term} \rangle$

Second-order symbolic execution

Symbolic Execution with Existential Second-Order Constraints. Mechtaev et al. FSE'18

```
size_t search(data, len, pred) {
  size_t i;
  for (i = 0; i < len; i++) {
    if (pred(data[i]))
      return i;
  }
  return len;
}
int positive(int x) { return x > 0; }
```

Question

What predicate would make `search` return 2, given the array `[0, 1, 2]`?

Answer

Execute `search([0, 1, 2],` ρ`)` symbolically with a second-order variable ρ, synthesize e.g. $\rho := \lambda x.\ x > 1$.

Example of second-order constraints

Assume that ρ is a second-order variable whose domain is restricted by the language LIA defined as follows:

$\langle \mathit{Term} \rangle$ ::= $\langle \mathit{Var} \rangle$ | $\langle \mathit{Constant} \rangle$
| $\langle \mathit{Term} \rangle$ + $\langle \mathit{Term} \rangle$ | $\langle \mathit{Term} \rangle$ - $\langle \mathit{Term} \rangle$ | $\langle \mathit{Constant} \rangle$ * $\langle \mathit{Term} \rangle$

LIA represents linear functions.

SAT

$\rho(0) > 0 \wedge \rho(1) \leq 0$ is satisfiable by $\rho := \lambda x.\ 1 - x$

UNSAT

$\rho(0) > 0 \wedge \rho(1) \leq 0 \wedge \rho(2) > 0$ is unsatisfiable since all functions in LIA are monotonic.

SE-ESOC

```
size_t search(data, len, pred) {
  size_t i;
  for (i = 0; i < len; i++) {
    if (pred(data[i]))
      return i;
  }
  return len;
}
```

Symbolic inputs (ρ):

```
  search((int[]){0, 1, 2}, 3, ρ);
```

Symbolic execution results:

Path condition	Input (pred)	Output
$\rho(0)$	$\lambda x.\ \mathit{true}$	0
$\neg\rho(0) \wedge \rho(1)$	$\lambda x.\ x > 0$	1
$\neg\rho(0) \wedge \neg\rho(1) \wedge \rho(2)$	$\lambda x.\ x > 1$	2
$\neg\rho(0) \wedge \neg\rho(1) \wedge \neg\rho(2)$	$\lambda x.\ \mathit{false}$	3

Repair via SE-ESOC

```
scanf("%d",&x);
for (i=0;i<10;i++) {
  int t = ρ(i,x);
  if (t>0)
    printf("1");
  else
    printf("0");
}
```

Unsatisfiable path conditions π_i ($\rho \in \mathit{Term}$ are monotonic):

$$\rho(0,5) > 0 \wedge \neg(\rho(1,5) > 0) \wedge \rho(2,5) > 0$$

...

Satisfiable path conditions π_i (less or equal to 20):

$$\rho(0,5) > 0 \wedge \rho(1,5) > 0 \wedge \rho(2,5) > 0 \wedge \rho(3,5) > 0 \wedge ...$$
$$\neg(\rho(0,5) > 0) \wedge \rho(1,5) > 0 \wedge \rho(2,5) > 0 \wedge \rho(3,5) > 0 \wedge ...$$
$$\neg(\rho(0,5) > 0) \wedge \neg(\rho(1,5) > 0) \wedge \rho(2,5) > 0 \wedge \rho(3,5) > 0 \wedge ...$$

Technical challenges of SE-ESOC

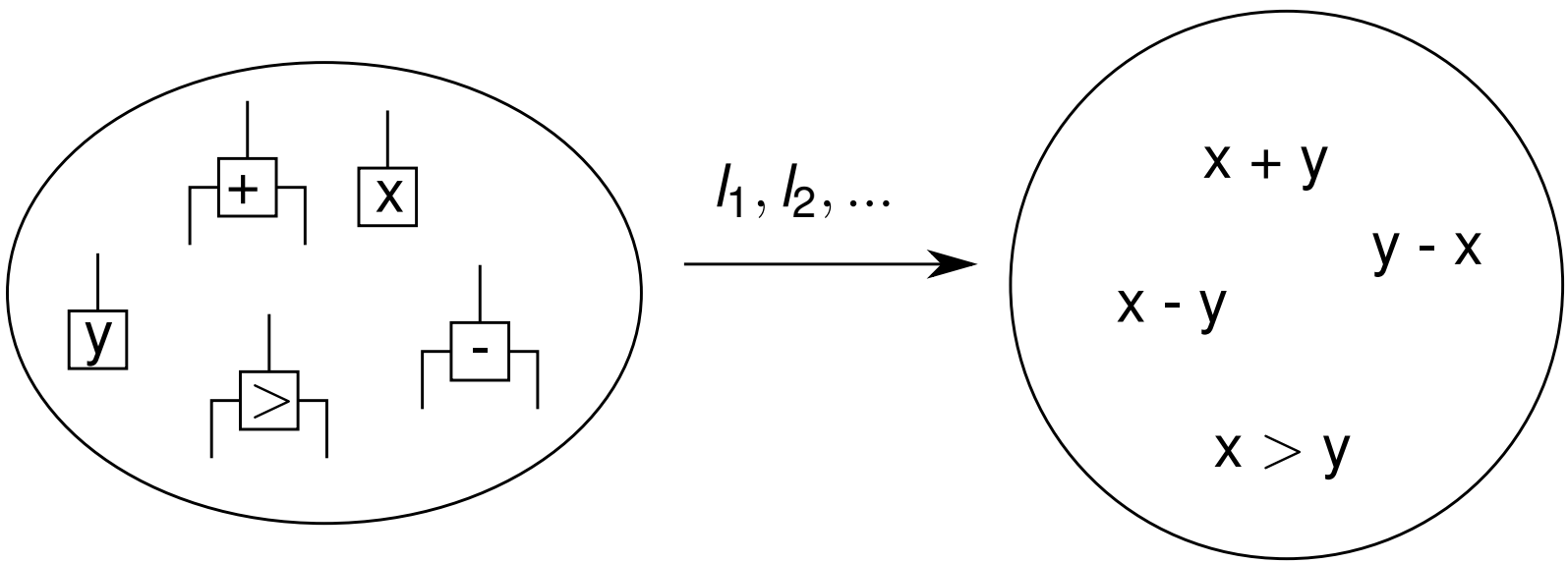
- Unsatisfiability checks are used in symbolic execution to avoid infeasible paths.
- Existing second-order constraint encodings (component-based synthesis by Jha et al.) provide very inefficient proofs of unsatisfiability.

SMT-based component-based synthesis

Oracle-Guided Component-Based Program Synthesis. Jha et al. ICSE'10

Encode second-order synthesis formula through a first order theory (LIA) and integer location variables:

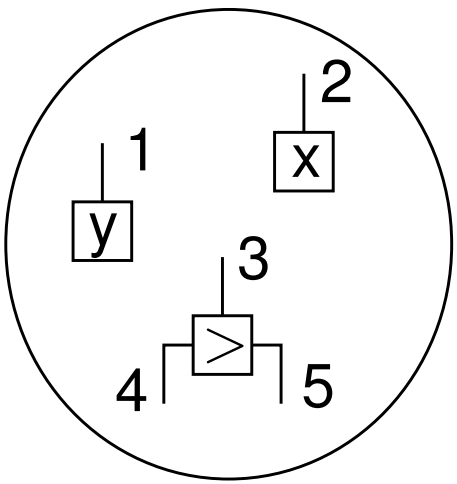
$$\exists p \in P.\ \bigwedge_{(\sigma,o)\in T} [\![p]\!]_\sigma = o \quad \mapsto \quad \exists l_1, l_2, \ldots \bigwedge_{(\sigma,o)\in T} \phi[l_1, l_2, \ldots, \sigma, o]$$



SMT-based component-based synthesis

Oracle-Guided Component-Based Program Synthesis. Jha et al. ICSE'10

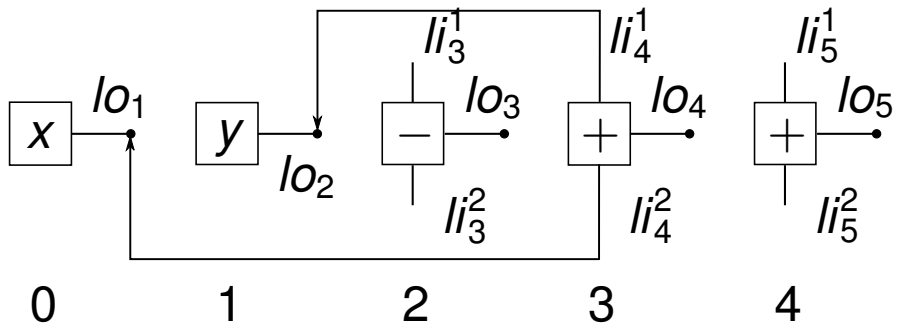
$\exists l_1, l_2, \ldots \bigwedge_{(\sigma,o)\in T} \phi[l_1, l_2, \ldots, \sigma, o]$ → SMT Solver



$l_1 = 0$
$l_2 = 1$
$l_3 = 2$
$l_4 = 1$
$l_5 = 0$

x > y

SMT-based component-based synthesis



$$lo_1 = 0$$
$$lo_2 = 1$$
$$lo_3 = 2$$
$$lo_4 = 3$$
$$li_4^2 = 0$$
$$li_4^1 = 1$$

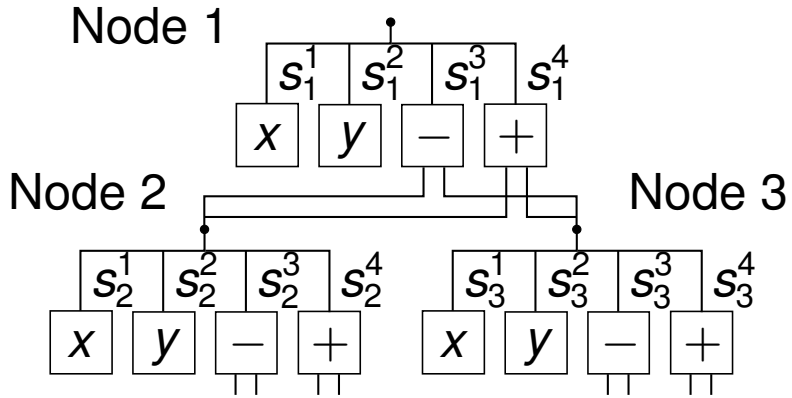
$$\phi_{range} := \bigwedge_{i \in [1,C]} \left(0 \leq lo_i < C \quad \wedge \bigwedge_{j \in [1,N_i]} 0 \leq li_i^j < C \right)$$

$$\phi_{cons} := \bigwedge_{i,j \in [1,C], i \neq j} lo_i \neq lo_j$$

$$\phi_{acyc} := \bigwedge_{i \in [1,C], j \in [1,N_i]} lo_i > li_i^j$$

$$\phi_{conn} := \bigwedge_{i,j \in [1,C], k \in [1..N_i]} lo_i = li_j^k \Rightarrow \mathtt{out}_i = \mathtt{in}_j^k$$

Propositional synthesis encoding



$$s_1^1 \mapsto x$$

$$s_1^3 \wedge s_2^1 \wedge s_3^2 \mapsto x - y$$

$$s_1^4 \wedge s_2^1 \mapsto \{x + T\}_{T \in Term}$$

$$\psi_{node} := \bigwedge_{j \in [1,C]} s_i^j \Rightarrow \mathtt{out}_i = F_j(\mathtt{out}_{i_1}, \mathtt{out}_{i_2}, ..., \mathtt{out}_{i_k})$$

$$\psi_{choice} := exactlyOne(s_i^1, s_i^2, ..., s_i^C)$$

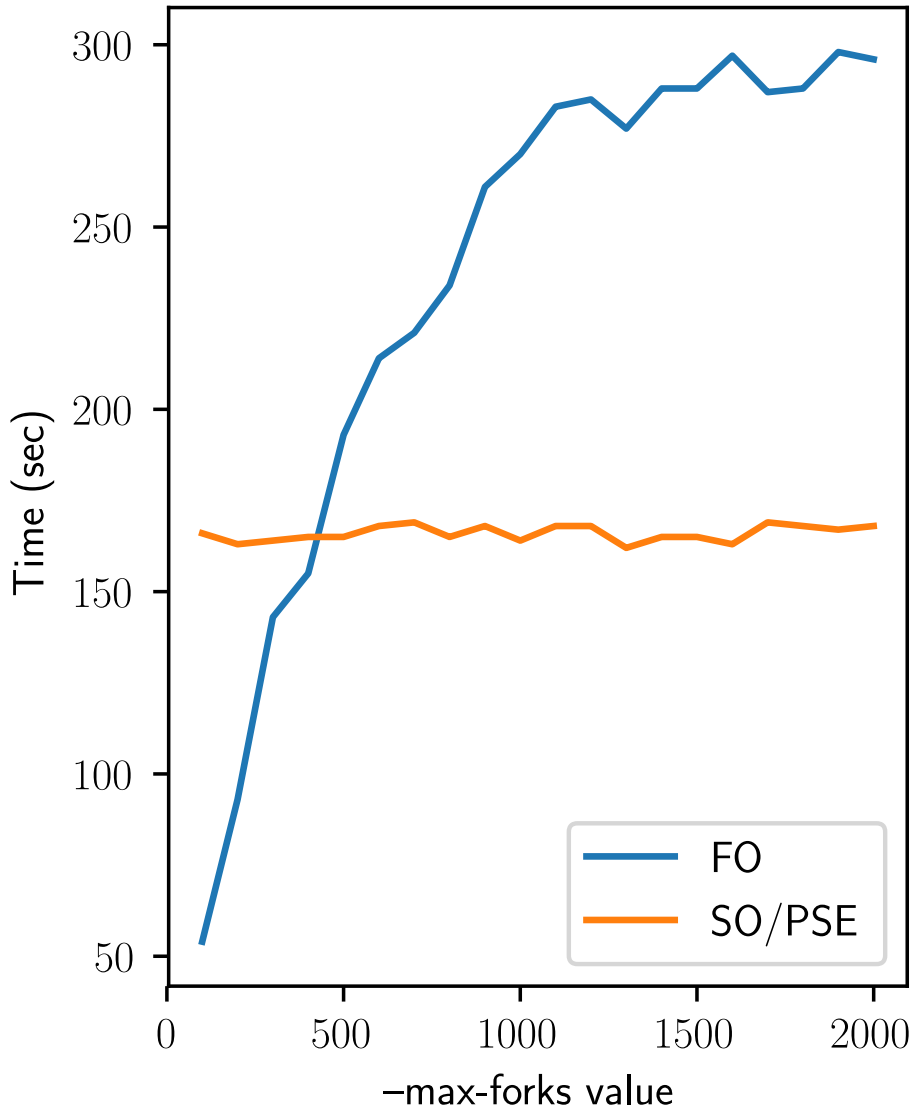
Example: repairing bug in GNU find

```
...
    error_severity(1);
    return;
  }
// ρ(ent->fts_info, ent->fts_errno, prev_depth)
else if (ent->fts_info == FTS_SLNONE)
  {
    if (symlink_loop(ent->fts_accpath))
...
```

More than 2000 paths for first-order SE, and only 16 paths for second-order SE:

ρ_1 :=(4 <= ent->fts_info)
ρ_2 :=!(ent->fts_errno == prev_depth)
ρ_3 :=((4 < ent->fts_info) && (prev_depth <= ent->fts_errno))
ρ_4 :=!(0 == ent->fts_errno)
ρ_5 :=((ent->fts_info == ent->fts_errno) || (9 <= prev_depth))
ρ_6 :=(ent->fts_info == (7 + prev_depth))
ρ_7 :=((prev_depth + ent->fts_errno) == (ent->fts_info - 6))
ρ_8 :=((ent->fts_errno < prev_depth) || (6 == ent->fts_info))
ρ_9 :=(ent->fts_info < (4 + ent->fts_errno))
ρ_{10} :=(ent->fts_info <= (ent->fts_errno + 6))
ρ_{11} :=!(ent->fts_info == 6)
ρ_{12} :=(0 <= prev_depth)
ρ_{13} :=!(4 < ent->fts_info)
ρ_{14} :=!(1 <= prev_depth)
ρ_{15} :=((ent->fts_errno < prev_depth) || (ent->fts_info <= 1))
ρ_{16} :=((ent->fts_errno < 32) || (prev_depth == ent->fts_info))

Time/number of explored paths trade-off



Example: bug in GNU Coreutils cut

GNU Coreutils wrongly interprets the command `-b 2-,3-` as `-b 3-` (extract input bytes staring from the third byte):

```
$ echo -ne '1234' | cut -b 2-,3-
34
```

instead of `-b 2-` (extract input bytes staring from the second byte):

```
$ echo -ne '1234' | cut -b 2-,3-
234
```

Developer tests:

```
echo -ne '1234' | cut -b 2-,3-
echo -ne '1234' | cut -b 3-,2-
```

Example: bug in GNU Coreutils cut

Automatic patch based on developer tests

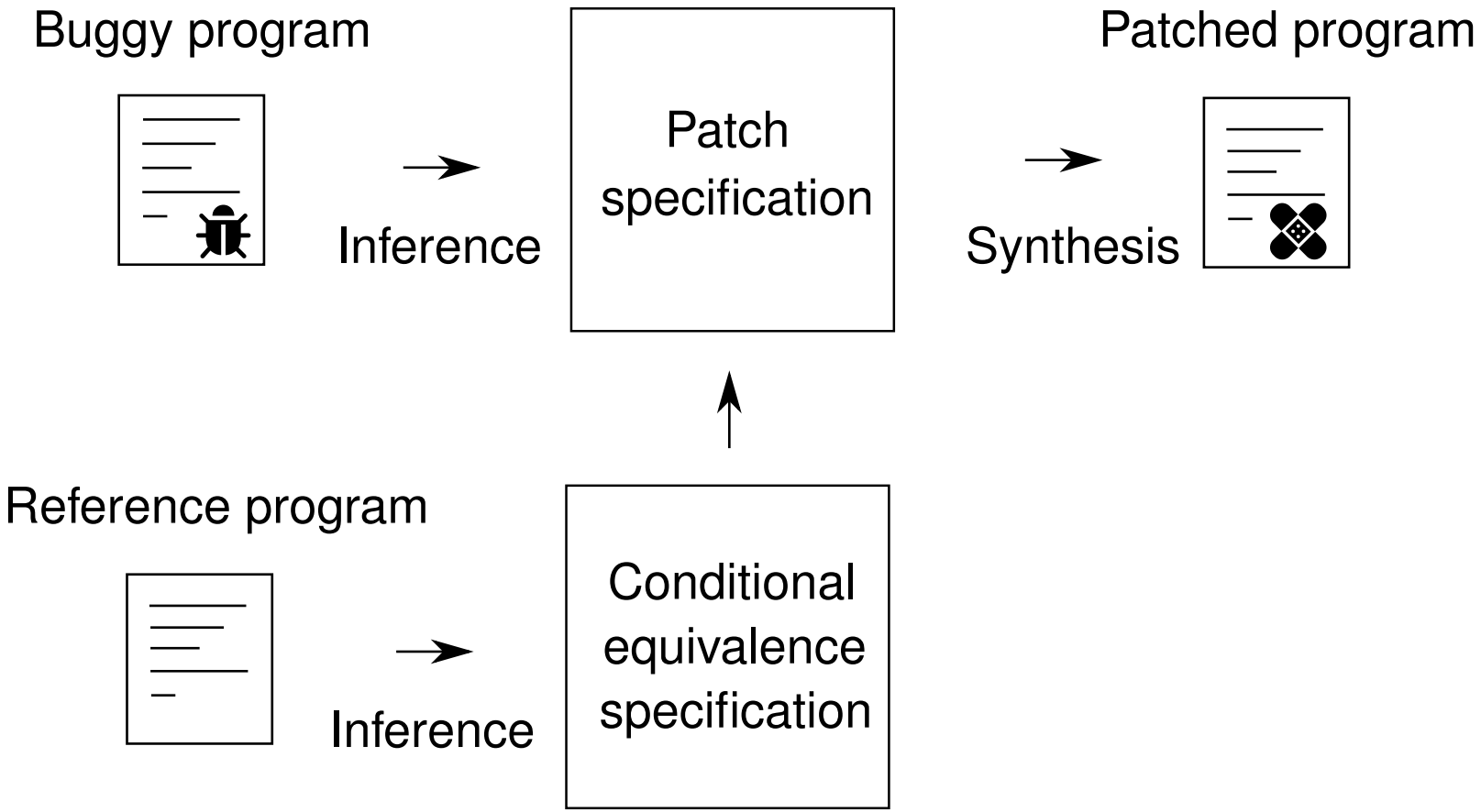
```
if (!rhs_specified)
  {
    if (eol_range_start == 0 || eol_range_start == 3)
      eol_range_start = initial;
    field_found = true;
  }
```

Developer patch

```
if (!rhs_specified)
  {
    if (eol_range_start == 0 || initial < eol_range_start)
      eol_range_start = initial;
    field_found = true;
  }
```

Infer specification from reference implementation

Semantic Program Repair Using a Reference Implementation. Mechtaev et al. ICSE'18



Example: bug in GNU Coreutils cut

Introduce parameters into the test:

```
echo -ne '1234' | cut -b α0-,α1-
```

Infer specification from Busybox cut, and synthesize patch using CEGIS:

patch 1	`eol_range_start` $==$ 3
cex 1	$-$b 3$-$, 4$-$
...	
patch N	`initial` $<$ `eol_range_start`
verified	

`initial` $<$ `eol_range_start` enables equivalence of Coreutils cut and Busybox cut for all values of α_0 and α_1.

Result

1. Test-equivalence is an efficient approach for repairing side-effect free expressions.
2. Second-order symbolic execution to efficiently analyse semantics of changes.
3. Can be used with additional specification (e.g. reference implementation) to address test-overfitting.

Future work

1. Synthesizing patches with side effects.
2. Optimize second-order constraint solving.
3. SMT-based synthesis with cost function.

Community website

`http://program-repair.org`