Automatic Patch Generation by Learning Correct Code
Fan Long, Martin Rinard

Daniel Perez

The University of Tokyo

Nov 21, 2016

Goal

Premises

- All program contain bugs
- We cannot be sure if a program contains bugs

We would like to fix bugs automatically as much as possible

About patch generation systems

Patch generation mostly work in the following way

1. Defect localization
2. Patch generation
3. Patch ranking
4. Patch validation

Defect localization

The approach used here assumes that there is effectively a defect and that automatic tests fail.
It then detect the defect as follow

- Priority on statements frequently on failure
- Priority on statements executed infrequently on success

Then uses heuristics to find the defect localization.

Patch generation

Assumptions

This approach here assumes the following

- The code is **almost** correct
- The program can be fixed by modifying a **single** statement

Patch generation

Assumptions

This approach here assumes the following

- The code is **almost** correct
- The program can be fixed by modifying a **single** statement

Patch anatomy

`if (C) { ... } else { ... }`	$\rightarrow$	`if (C && E){ ... } else { ... }`
`if (C) { ... } else { ... }`	$\rightarrow$	`if (C || E) { ... } else { ... }`
S	$\rightarrow$	`if (E) { S }`
Replace S	$\rightarrow$	S[replace v1 with v2]
Copy and Replace S	$\rightarrow$	Q[replace v1 with v2]; S
Initialize S	$\rightarrow$	`memset(&e, 0, sizeof(e)); S`

Patch ranking

Previous approaches acted as follow

- No sorting
- Random order
- Heuristic ranking

This paper main contribution is a way to sort generated patches efficiently.

Patch validation

After sorting the goal is to get a list of patches which pass the tests

1. Actually patch the source code
2. Drop the patches which do not pass the tests
3. Repeat on until finding *n* patches which pass the tests

Improvements

The main contribution of this paper paper is the improvement of patch ranking.
State-of-the art patch generation systems used heursitics to rank correct patches.
This paper proposes a machine learning based approach to learn from correct human patches.

Insights and assumptions

Assumptions

- The code is **almost** correct
- The program can be fixed by modifying a **single** statement

Insights and assumptions

Assumptions

- The code is **almost** correct
- The program can be fixed by modifying a **single** statement

Insights

- Correct patches share universal features that hold across applications
- These features capture interactions between the patch and the surrounding code
- These features can be learned to recognize correct patches

Probabilistic model

Given a program p and

- $L(p)$: program points that the system can try to modify
- $l \in L(p)$: modification point
- $M(p, l)$: possible transformations at l
- $m \in M(p, l)$: a program modification
- δ: a patch defined as $\langle m, l \rangle$
- $\phi(p, m, l)$: extracted features in p for $\langle m, l \rangle$

the goal is to maximize

$$P(\delta \mid p, \theta) = P(m, l \mid p, \theta)$$

when the patch is correct

Probabilistic model

The probability $P(m, l \mid p, \theta)$ is given using the following model

$$P(m, l \mid p, \theta) = \frac{1}{Z} \cdot A \cdot B$$
$$A = (1 - \beta)^{r(p,l)}$$
$$B = \frac{\exp\left(\phi(p, m, l) \cdot \theta\right)}{\sum_{l' \in L(p)} \sum_{m' \in M(p,l')} \exp\left(\phi(p, m', l') \cdot \theta\right)}$$
$$Z = \text{partition function}$$

In A, $r(p, l)$ is given by the defect localization and $\beta = 0.02$
B is the weight of the patch divided by the sum of the weight of all other patches

Feature extraction

A key point is that features must be universal (i.e. not application dependent)

The extracted features are divided into two types

- Modification features
  - The kind of modification
  - The relationship between the patched statement and the modification kind
- Program value features: how variables are used in the original program and in the patch
  Program specific information is abstracted (e.g. variable names, values)

Program value features

Commutative operators	Is an operand of +, *, ==, !=
Binary operators	Is an operand of −, /, <, >
Unary operators	Is an operand of −, ++, !
Enclosing statements	occurs in an assign/loop statement occurs in a branch condition is a function call parameter
Value traits	Is a local or global variable, is argument

Training process

The training process uses the following steps

1. Collect correct patches written by humans
2. Generate patches from the source before the patch
3. For each patch, compute the AST difference
4. Extract the patch features
5. Compute θ to maximize the probability of the human patches to be true

Repair algorithm

The repair algorithm is the same as for existing patch generation systems

1. Localize the defect
2. Generate the patches for the defect
3. Use the learned parameters θ to rank the patches
4. Validate the patches by running the test suite

The system assumes that the defect is detected by the test suite.

Training set

The system has been trained with a total of 777 patches

pr	12
curl	53
httpd	75
libtiff	11
php	187
python	114
subversion	240
wireshark	85
Total	777

Experimental results

Out of 69 defects, the system finds 39 plausible patches, in which 18 are correct with a timeout of 12h.

App	LoC	Defects	Plausible		Correct	
			Prophet	SPR	Prophet	SPR
libtiff	77k	8	5	5	2,2	1,1
lighthttpd	62k	7	3	3	0,0	0,0
php	1046k	31	17	16	13,10	10,9
gmp	145k	2	2	2	1,1	1,1
gzip	491k	4	2	2	1,1	1,0
python	407k	9	5	5	0,0	0,0
wireshark	2814k	6	4	4	0,0	0,0
fbc	97k	2	1	1	1,1	1,0
Total		69	39	38	18,15	16,11

Conclusion

- Experimental results confirm the hypothesis: correct code share properties across applications
- The learning approaching used outperforms existing heursitics approaches
- Capturing the interaction between the patch and the program is essential to learn meaningful features

References

- Long, Fan and Rinard, Martin, Automatic Patch Generation by Learning Correct Code, POPL'16
- `https://www.youtube.com/watch?v=d-FTp3eXnQ8`