The 38th ACM/SIGAPP Symposium on Applied Computing (SAC 2023)

MCRepair: Multi-Chunk Program Repair via Patch Optimization with Buggy Block

Jisung Kim and Byeongjung Lee

{kimjisung78, bjlee}@uos.ac.kr

Software Engineering Laboratory (SELAB)

Dept. of Computer Science and Engineering

University of Seoul, Republic of Korea

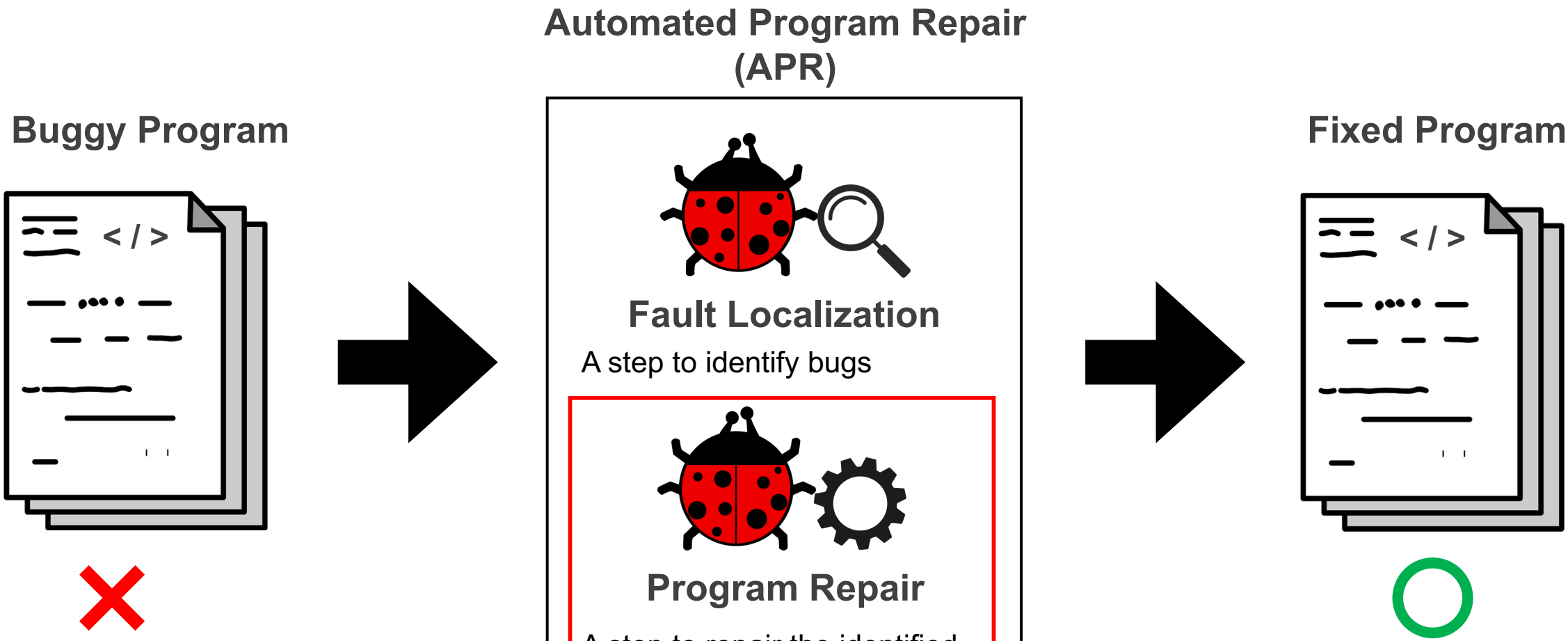
Presented by Jisung Kim

Presented at 28th, March, 2023.

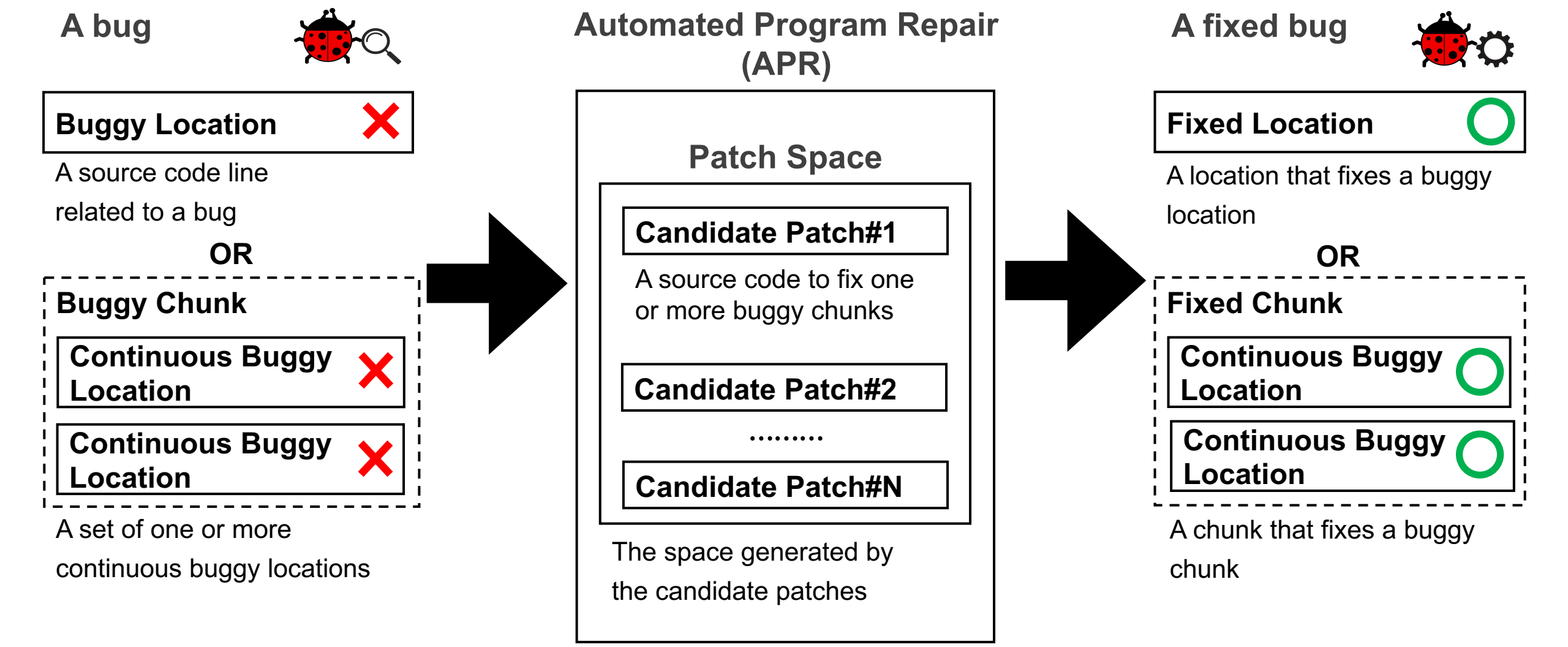
서울시립대학교 UNIVERSITY OF SEOUL

acm SAC2023

1. Introduction

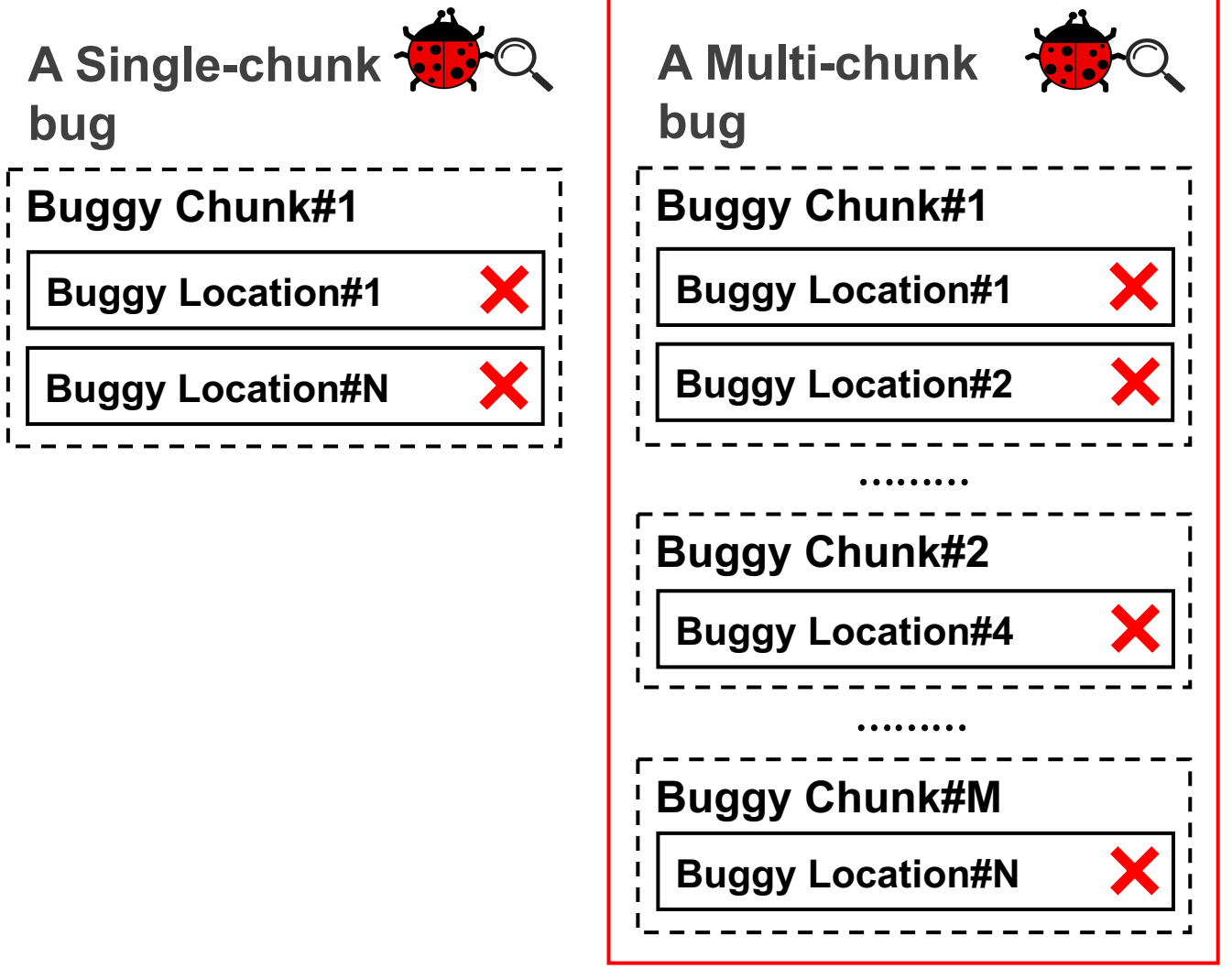


1. Introduction



Kind of Bugs

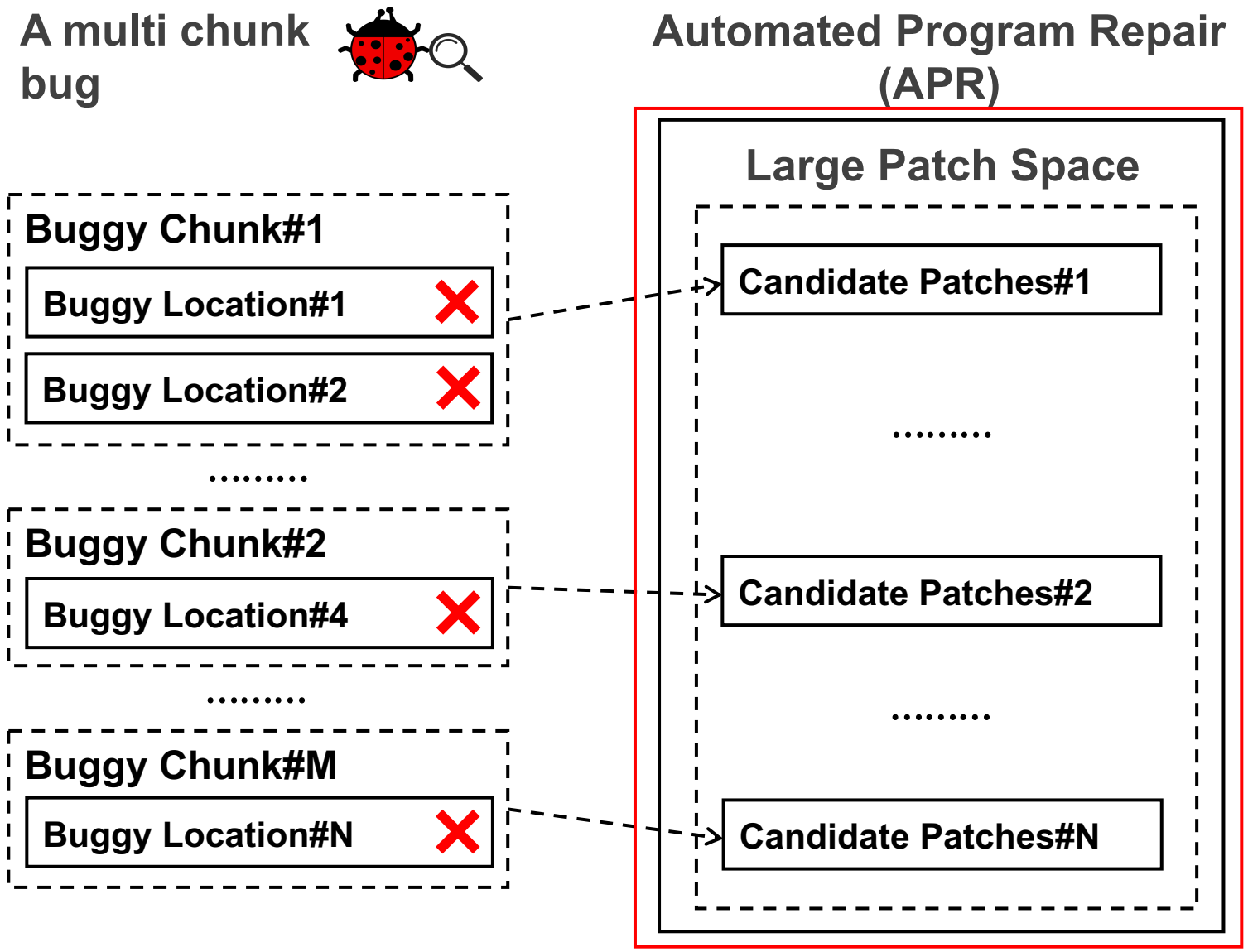
- Bugs are classified into single-chunk and multi-chunk bugs according to the number of buggy chunks.
- **Especially, repairing multi-chunk bugs is a challenging problem due to the three complex problems.**



1st Problem

(Large patch space based on the number of buggy chunks)

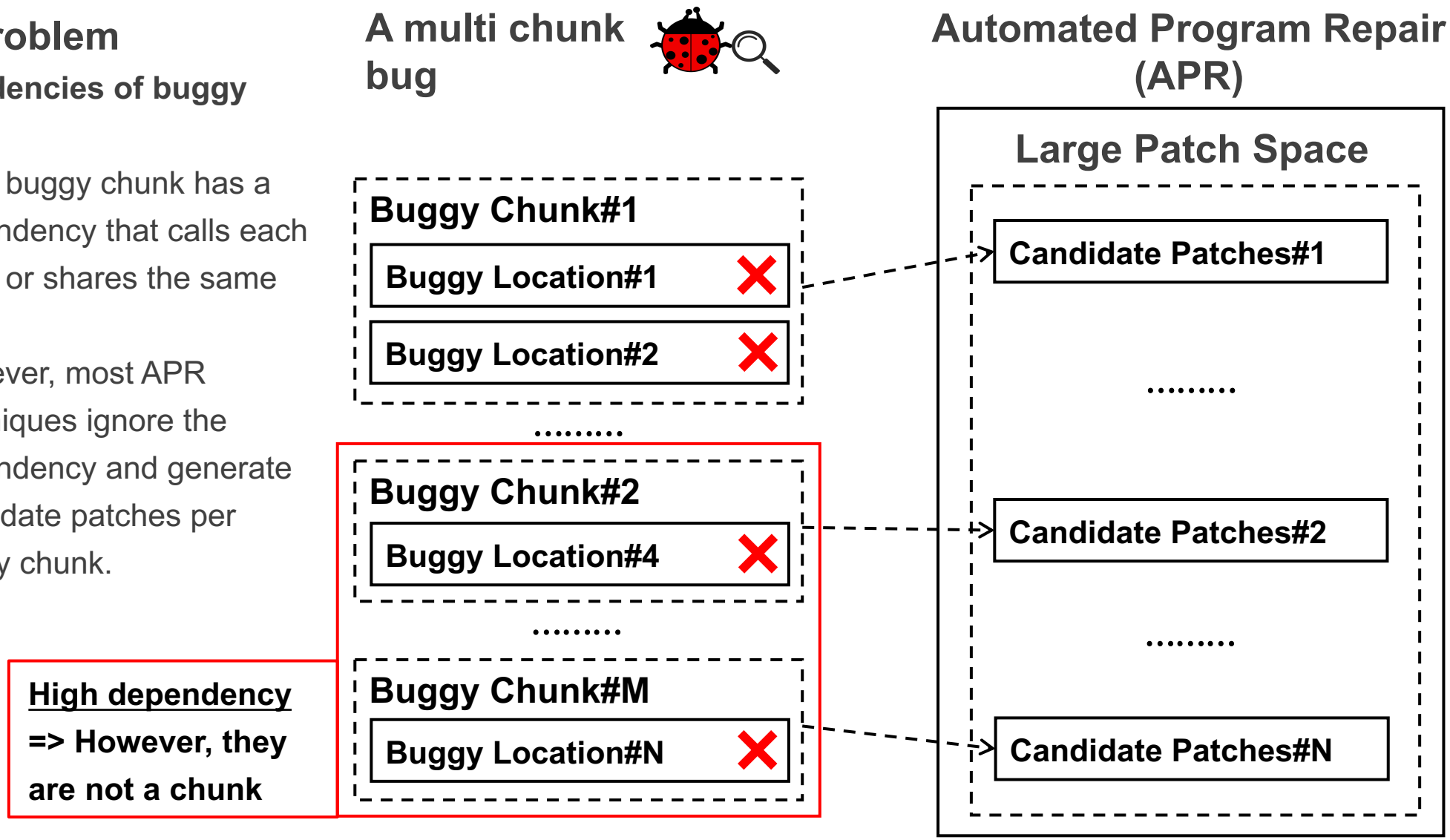
- When candidate patches are generated per buggy chunk, patch space increases exponentially.
- It becomes more time-consuming to find the correct patches in the space.



2nd Problem

(Dependencies of buggy chunks)

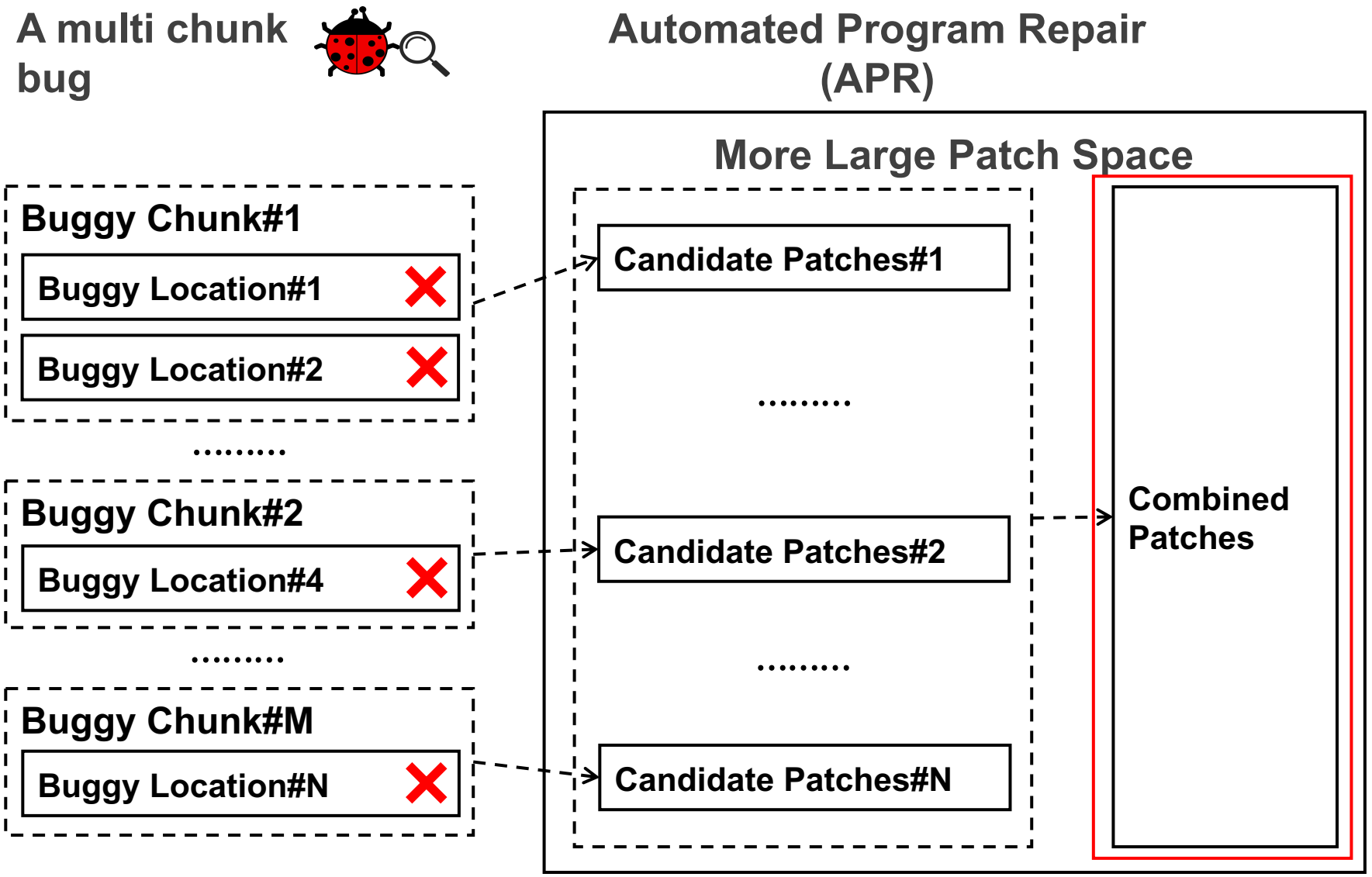
- Each buggy chunk has a dependency that calls each other or shares the same flow.
- However, most APR techniques ignore the dependency and generate candidate patches per buggy chunk.



3rd Problem

(Patch combination)

- When a module has a multi-chunk bug, an APR technique must combine candidate patches and repair the module simultaneously.
- Nevertheless, the patch space also increases exponentially based on the number of combinations.



2. Approach

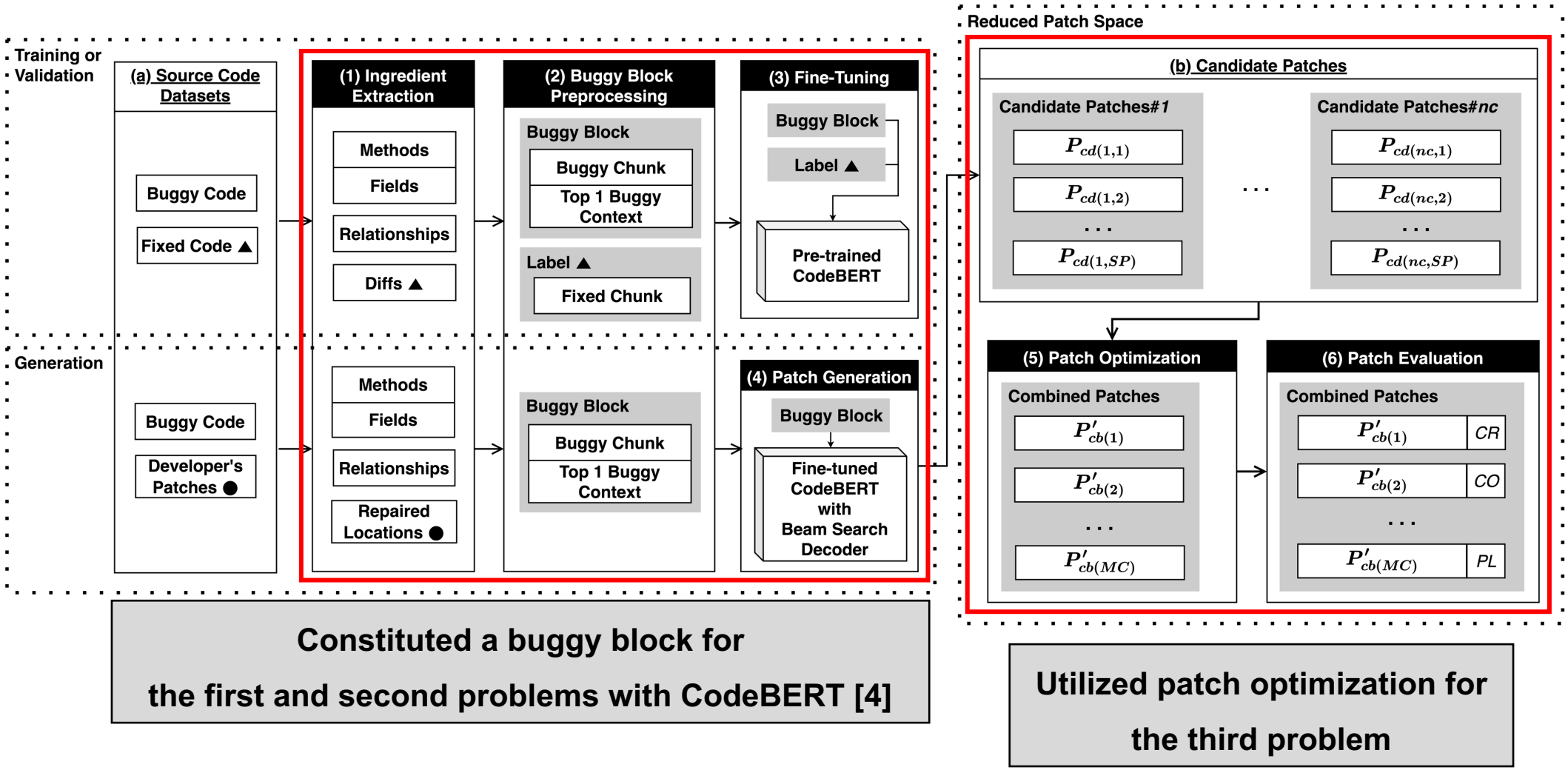
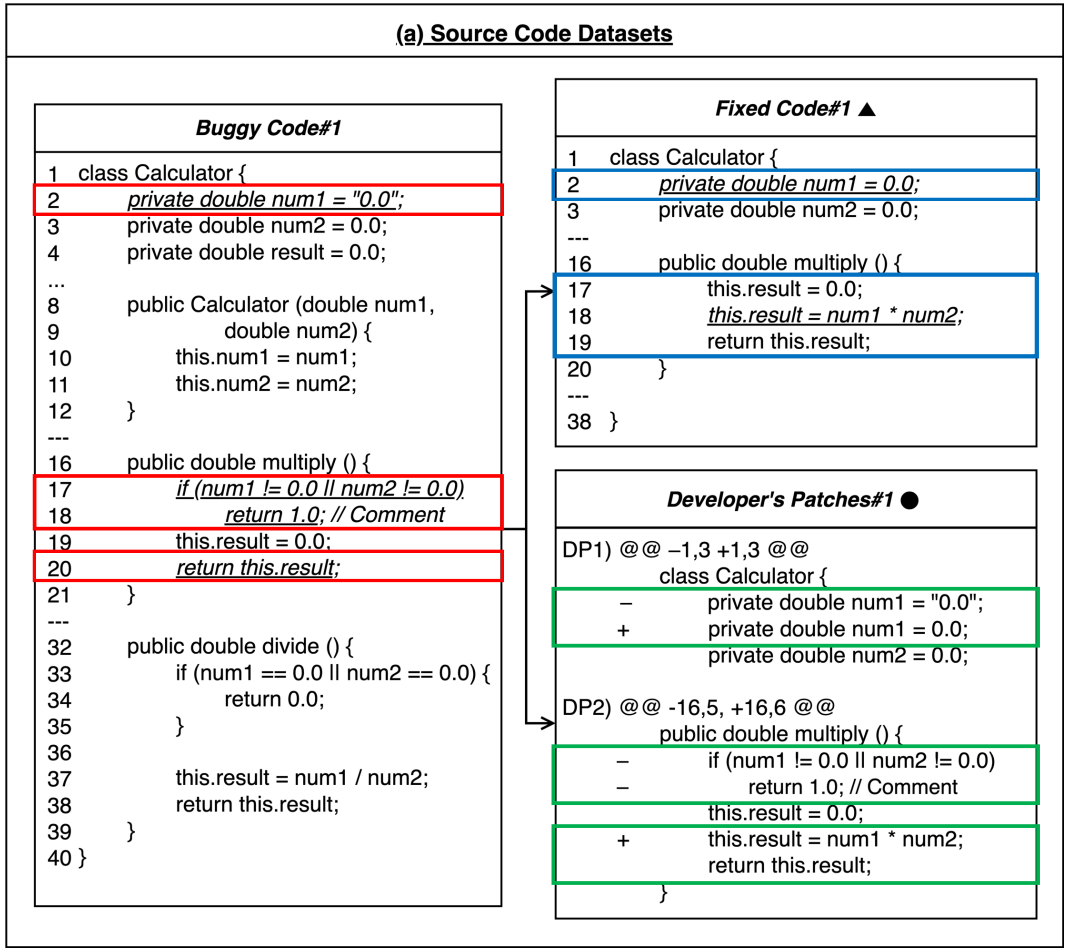


Figure 1: An overview of Multi-Code Repair (MCRepair for short).

(▲: Only use for training or validation, ●: Only use for generation)

An example of (a) Source Code Datasets

- MCRepair splits the datasets into three sets such as training, validation, and generation.
- *Buggy Code#1* has the three buggy chunks as in line 2, lines 17–18, and line 20 (Red boxes of *Buggy Code#1*).
  - To completely fix the code, it must update line 2, delete lines 17–18, and add a line before line 20. (Blue boxes of *Fixed Code#1* or Green boxes of *Developer's Patches#1*) .



An example of the source code datasets in Fig. 2

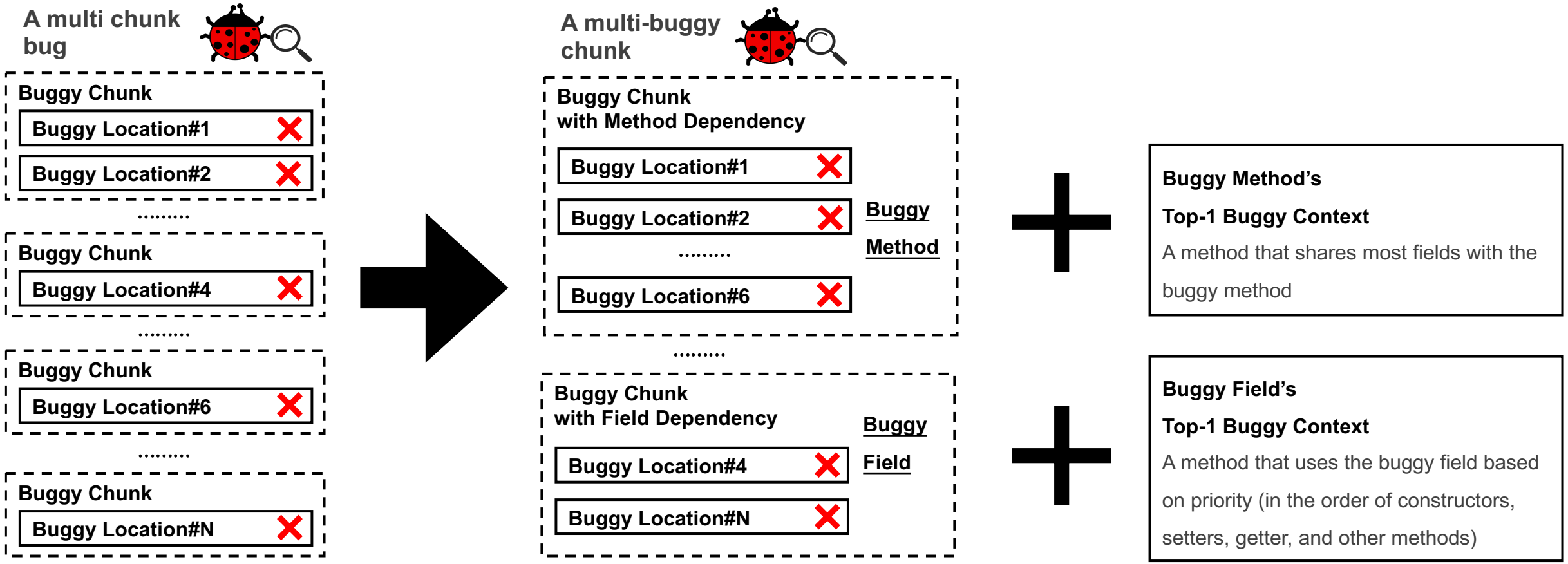
(▲: Only use for training or validation,

●: Only use for generation)

2. Approach

Buggy Block

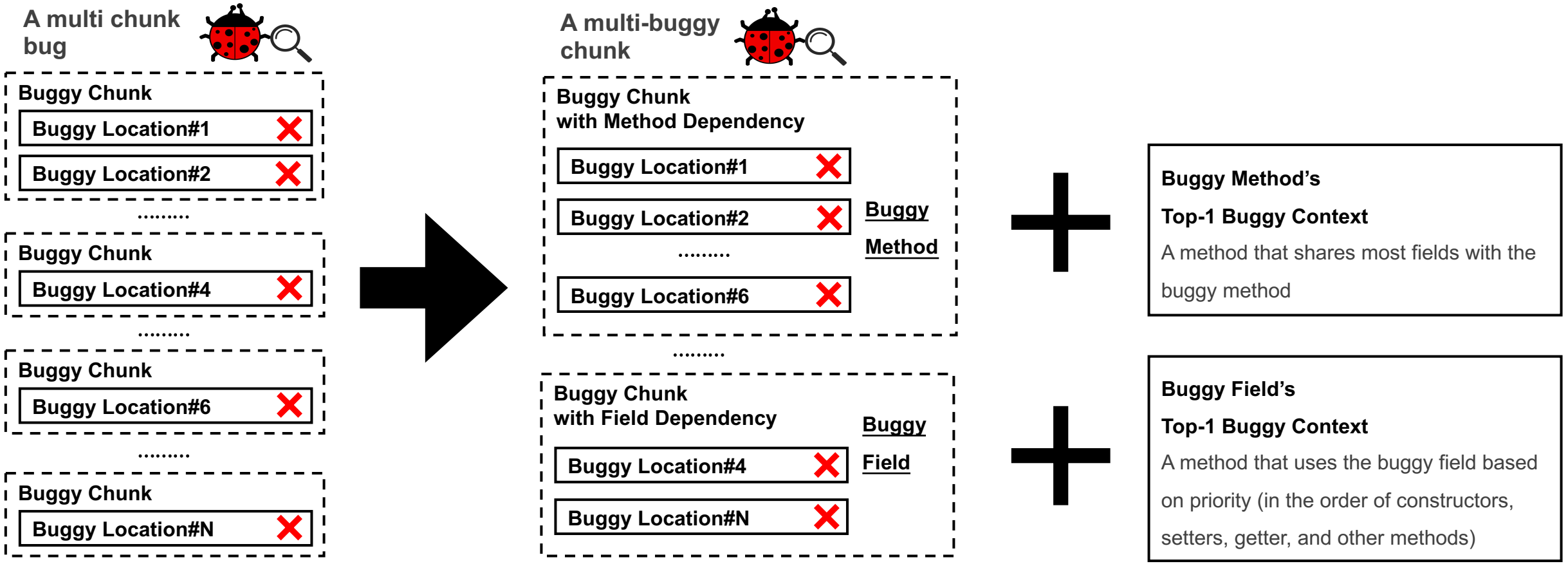
- A method that binds one or more buggy chunks into a “multi-buggy chunk” with method and field dependencies



2. Approach

Buggy Block

- Constitute each data that includes a buggy chunk in a multi-buggy chunk and its “top-1 buggy context” to achieve better performance



2. Approach

CodeBERT [4]

- A specific BERT [5] model for source code datasets
- Be fine-tuned by buggy blocks and generate a set of candidate patches per buggy block

Its pre-trained processes

1) Masked Language modeling (MLM)
2) Replaced token detection (RTD)

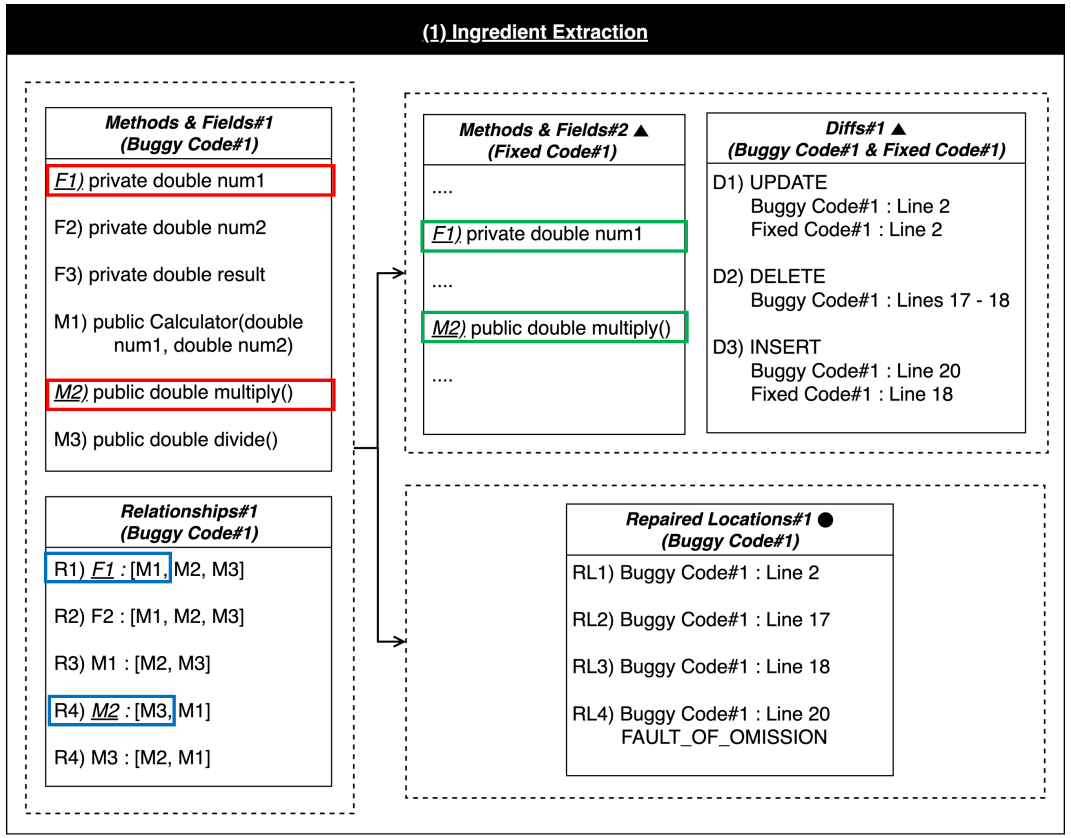
Its pre-trained programming languages

1) C
2) Java
3) Python
4) Javascript
5) PHP
6) Go

Its tokenization method

1) Byte-pair encoding (BPE)
   => A sub-word tokenization method to prevent OOV

2. Approach



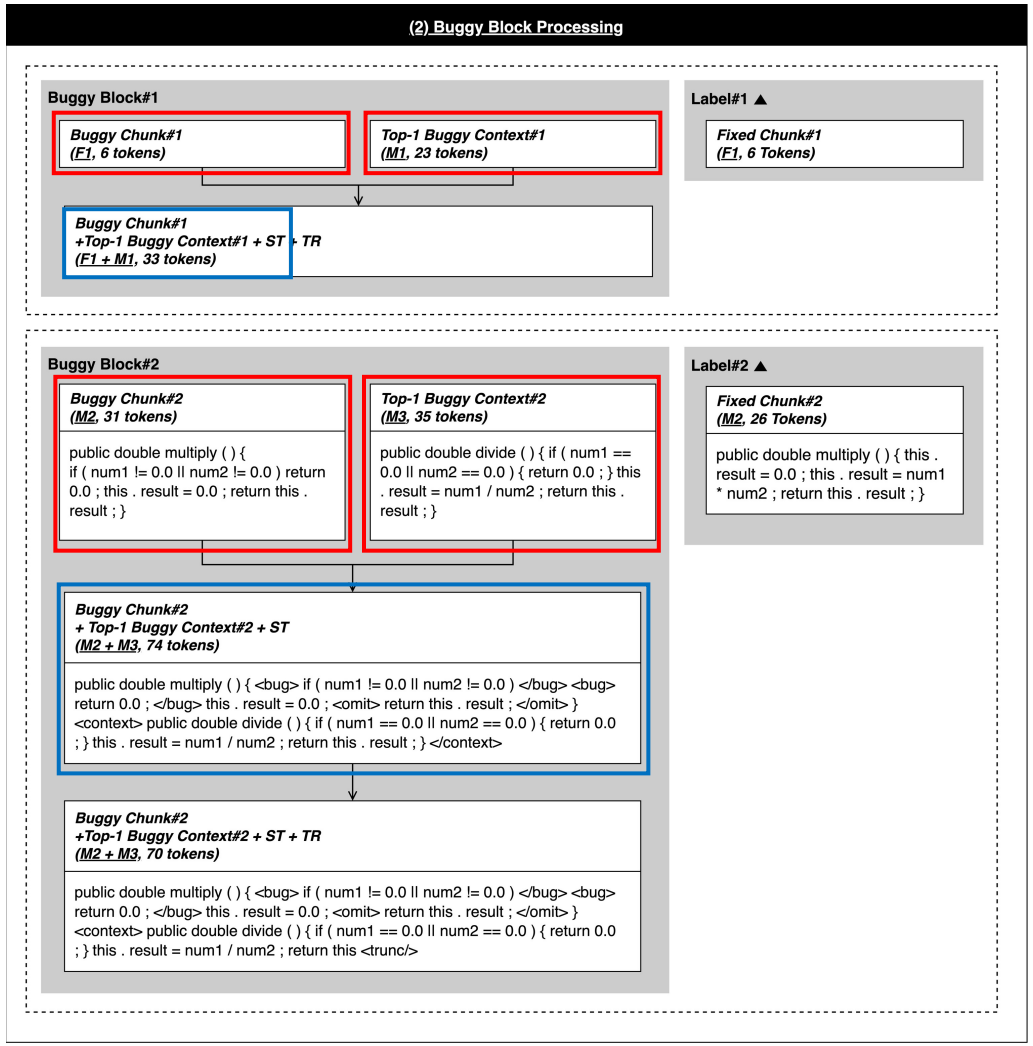
(1) Ingredient Extraction in Fig. 2

(M: Method, F: Field, R: Relationship,

D: Difference, RL: Repaired Location)

Process of Buggy Block

1) Extract buggy chunks (Red boxes), their top-1 buggy contexts (Blue boxes), and their fixed chunks (Green boxes) based on ingredients
   - Because a fixed chunk indicates a fixed version of a buggy chunk, it is labeled for the buggy chunk.
   - The ingredients consist of methods, fields, differences, and repaired locations between buggy code and fixed code.

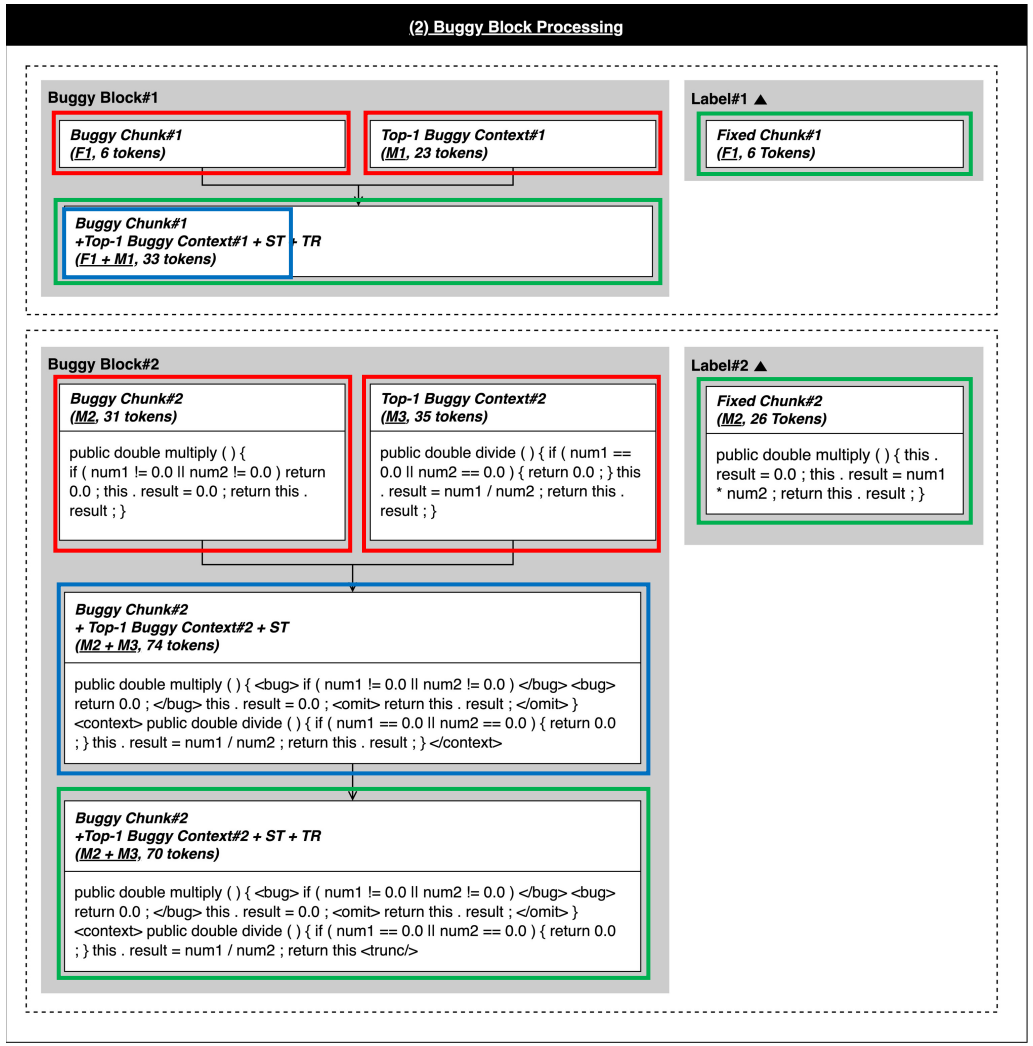


(2) Buggy Block Preprocessing in Fig. 2

(ST: Special Tokens, TR: Truncation)

Process of Buggy Block

2) Construct the tokens of the buggy chunks and their buggy contexts (Red boxes) into buggy blocks (Blue boxes)

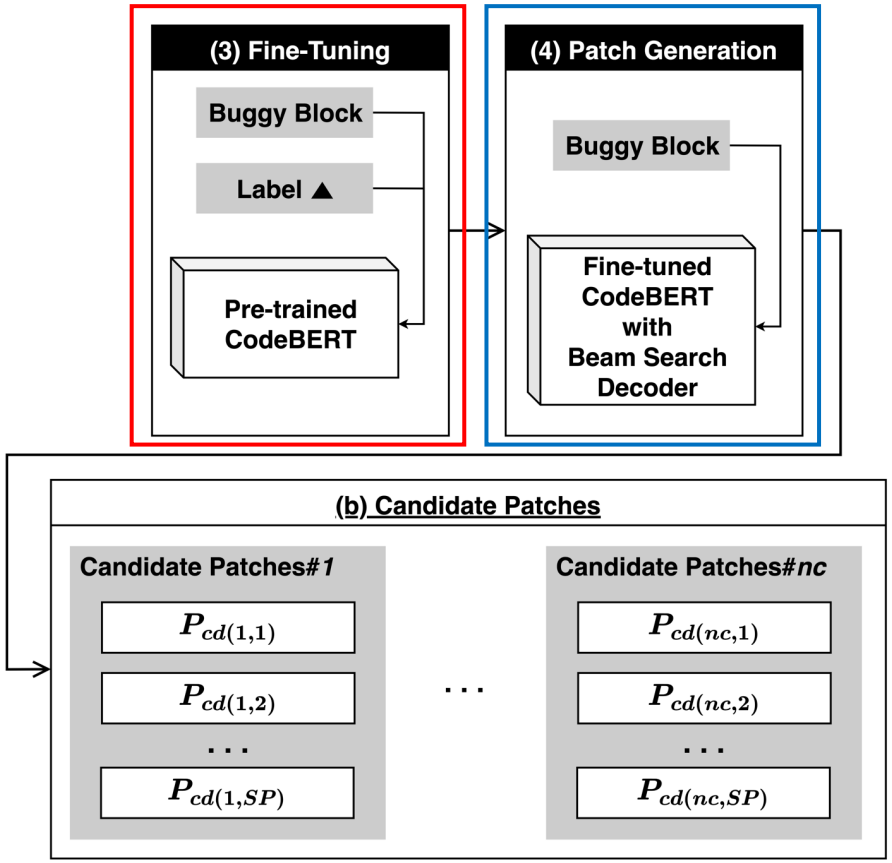


(2) Buggy Block Preprocessing in Fig. 2

(ST: Special Tokens, TR: Truncation)

Process of Buggy Block

3) Add special tokens to each block and label for training, validation, and generation (Green boxes)



(3) Fine-Tuning and (4) Patch Generation in Fig. 1

(ST: Special Tokens, TR: Truncation)

Process of Buggy Block

4) Fine-tune a pre-trained CodeBERT by inserting buggy blocks for training or validation with their labels (Red boxes)
5) Generate candidate patches per buggy block for generation using a fine-tuned CodeBERT and its beam search decoder (Blue boxes)

2. Approach

Buggy Block for the first and second problems

- In this slide, we generate 6 candidate patches per buggy chunk.

The buggy block decreases the numbers of buggy chunks and their candidate patches using field and method dependencies.

Buggy Code#1

```
1   class Calculator {
2       private double num1 = "0.0";
3       private double num2 = 0.0;
4       private double result = 0.0;
---
16      public double multiply () {
17          if (num1 != 0.0 || num2 != 0.0)
18              return 0.0;
19          this.result = 0.0; //comment
20          return this.result;
21      }
---
40 }
```

Original Patch Space for 3 X 6 (18) candidate patches of three buggy chunks

Buggy Code#1 of Fig. 2

Buggy Code#1

```
1   class Calculator {
2       private double num1 = "0.0";
3       private double num2 = 0.0;
4       private double result = 0.0;
---
16      public double multiply () {
17          if (num1 != 0.0 || num2 != 0.0)
18              return 0.0;
19          this.result = 0.0; //comment
20          return this.result;
21      }
---
40 }
```

Reduced Patch Space for 2 X 6 (12) candidate patches of two buggy chunks in a multi-buggy chunk with a buggy field and a buggy method

Buggy Code#1 of Fig. 2

(b) Candidate Patches

Candidate Patches#1

$P_{cd(1,1)}$ $P_{cd(1,5)}$ $P_{cd(1,2)}$ $P_{cd(1,6)}$ $P_{cd(1,3)}$ $P_{cd(1,4)}$

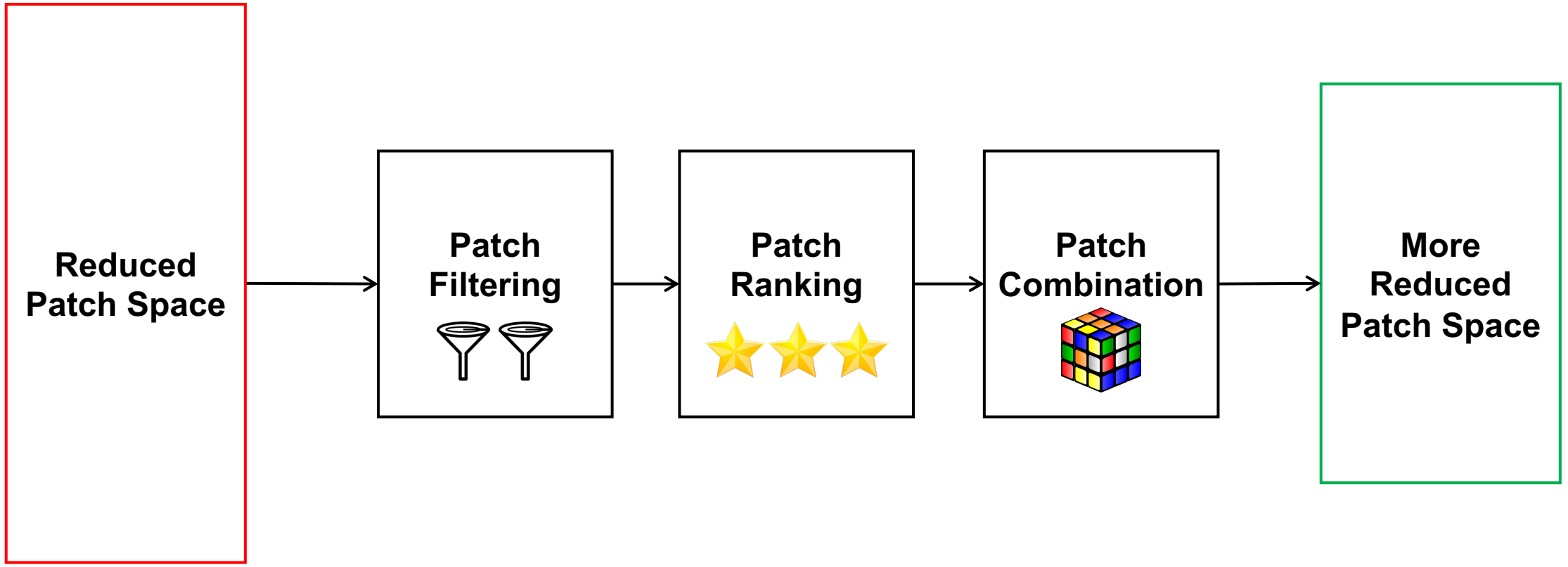
Candidate Patches#2

$P_{cd(2,1)}$ $P_{cd(2,5)}$ $P_{cd(2,2)}$ $P_{cd(2,6)}$ $P_{cd(2,3)}$ $P_{cd(2,4)}$

An example of the candidate patches in Fig. 3

Patch Optimization

- An optimization strategy to effectively combine the generated candidate patches after filtering and ranking

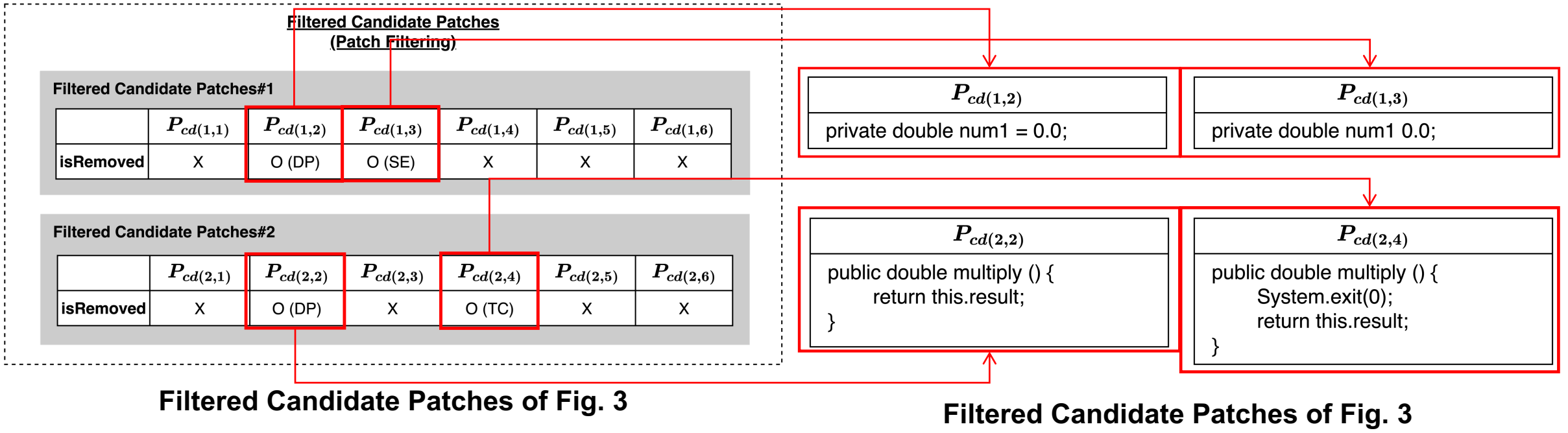


Process of Patch Optimization

1) Patch Filtering

It removes candidate patches that are duplicated, have syntax errors, or include termination code (e.g., System.exit) to minimize the number of candidate patches per buggy block.

- Because the patches that include the termination code cannot evaluate, they are filtered out.



Filtered Candidate Patches of Fig. 3

Filtered Candidate Patches of Fig. 3

(DP: Duplicated, SE: Syntax Error, TC: Termination Code)

Process of Patch Optimization

2) Patch Ranking

It ranks candidate patches using Equation 1.

- According to Reference number [6], a correct patch has a small-edit size compared to its buggy code because the patch minimally changes the buggy code.

$$Sim(P_{cd(i,j)}) = (\alpha \times S_{act}(P_{cd(i,j)})) + (\beta \times S_{ngra}(P_{cd(i,j)}))$$

Action similarity **N-gram similarity**

A weighted sum of the two ranking measures

Ranked Candidate Patches (Patch Ranking)

Candidate Patches#1

	S_{act}	S_{ngra}	Sim	Rank	isSelected
$P_{cd(1,1)}$	1.000	0.333	0.666	1	O
$P_{cd(1,4)}$	1.000	0.000	0.500	4	X
$P_{cd(1,5)}$	1.000	0.142	0.571	2	O
$P_{cd(1,6)}$	1.000	0.142	0.571	2	O

Candidate Patches#2

	S_{act}	S_{ngra}	Sim	Rank	isSelected
$P_{cd(2,1)}$	0.750	0.266	0.508	3	O
$P_{cd(2,3)}$	0.642	0.000	0.321	4	X
$P_{cd(2,5)}$	1.000	0.400	0.700	1	O
$P_{cd(2,6)}$	0.800	0.348	0.574	2	O

Ranked Candidate Patches of Fig. 3

Process of Patch Optimization

2) Patch Ranking – *Action similarity*

- Action similarity considers the expected actions between a buggy code and its candidate patch at “Repaired Locations” using Equation 2.

Obatined miniumum ***Penalized distance***

$$S_{act} = \frac{minTotal + p \times disTotal}{minTotal + disTotal}$$

where,

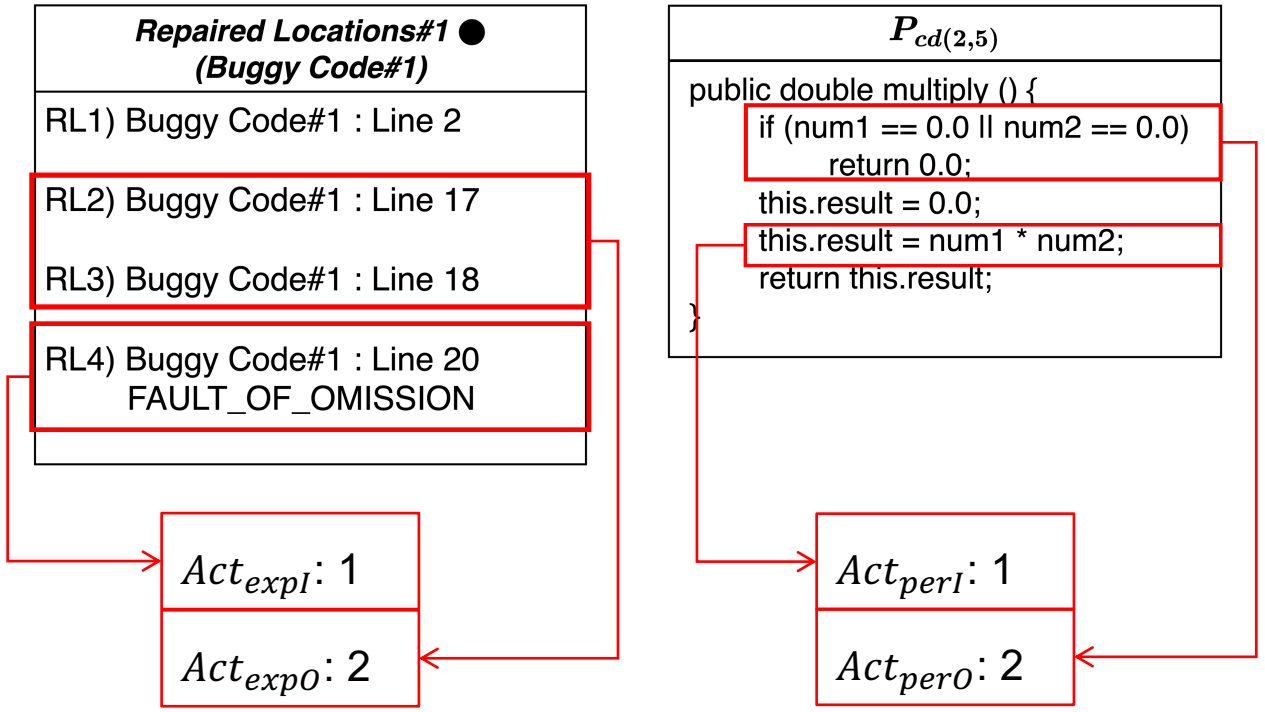
- $minTotal = min(Act_{expI}, Act_{perI}) + min(Act_{expO}, Act_{perO})$
- $disTotal = |Act_{expI} - Act_{perI}| + |Act_{expO} - Act_{perO}|$
- p is a penalty $(0 < p < 1)$.

The expected insertions that a location indicates “FAULT_OF_OMISSION”

The expected other actions including updates and deletions

Process of (5) Patch Optimization

2) Patch Ranking – *Action similarity*'s example



Repaired Location#1 of Fig. 2

A candidate patch of Fig. 3

$$\frac{S_{act}}{1.000} = \frac{minTotal + p \times disTotal}{minTotal + disTotal}$$

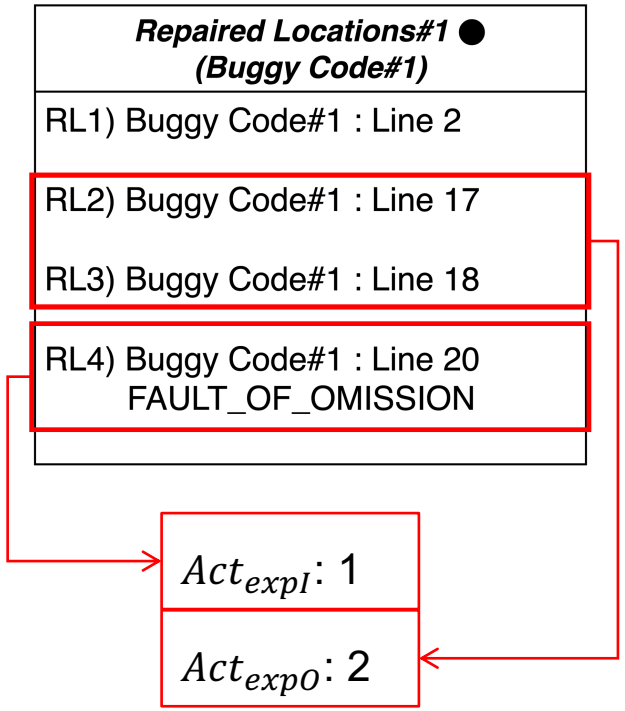
(1 + 2 = 3 for $minTotal$; 0.5 for p; 0 + 0 = 0 for $disTotal$; 1 + 2 for the denominator $minTotal$; 0 + 0 for the denominator $disTotal$)

where,

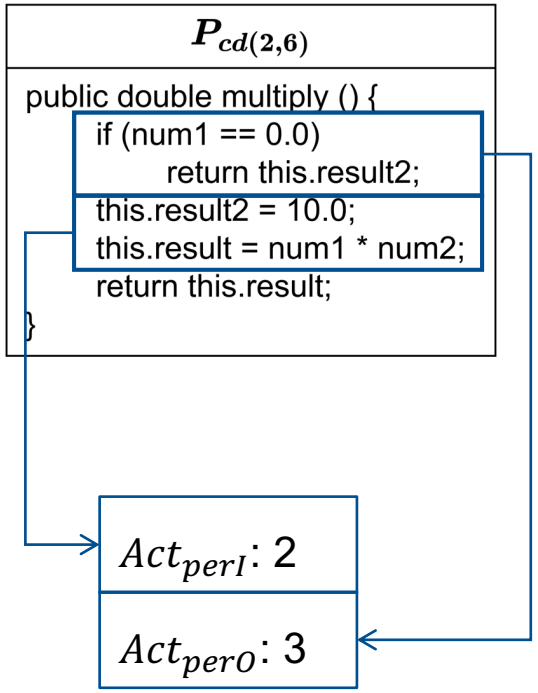
- $minTotal = min(Act_{expI}, Act_{perI}) + min(Act_{expO}, Act_{perO})$
- $disTotal = |Act_{expI} - Act_{perI}| + |Act_{expO} - Act_{perO}|$
- p is a penalty ($0 < p < 1$).

Process of (5) Patch Optimization

2) Patch Ranking – *Action similarity*'s example



Repaired Location#1 of Fig. 2



A candidate patch of Fig. 3

$$\underbrace{S_{act}}_{0.800} = \frac{\overbrace{minTotal}^{1+2=3} + \overbrace{p}^{0.5} \times \overbrace{disTotal}^{1+1=2}}{\underbrace{minTotal}_{1+2=3} + \underbrace{disTotal}_{1+1=2}}$$

where,

- $minTotal = min(Act_{expI}, Act_{perI}) + min(Act_{expO}, Act_{perO})$
- $disTotal = |Act_{expI} - Act_{perI}| + |Act_{expO} - Act_{perO}|$
- p is a penalty $(0 < p < 1)$.

Process of Patch Optimization

2) Patch Ranking – *N-gram similarity*

- N-gram similarity considers the structure and syntax changes between a buggy code and its candidate patch using Equation 3.

$$S_{ngra} = |B \cap P| \,/\, |B \cup P|$$ ***Jaccard Similarity***

- B and P are the sets of the n-gram tokens of $P_{cd(i,j)}$ and $P_{cd(i,j)}$'s buggy chunk.

Process of Patch Optimization

2) Patch Ranking – *N-gram similarity*'s example

Buggy Code#1

```
1   class Calculator {
2         private double num1 = "0.0";
3         private double num2 = 0.0;
4         private double result = 0.0;
---
16        public double multiply () {
17              if (num1 != 0.0 || num2 != 0.0)
18                    return 1.0; // Comment
19              this.result = 0.0;
20              return this.result;
21        }
---
40 }
```

$P_{cd(2,5)}$

```
public double multiply () {
        if (num1 == 0.0 || num2 == 0.0)
                return 0.0;
        this.result = 0.0;
        this.result = num1 * num2;
        return this.result;
}
```

$$S_{ngra} = \frac{|B \cap P|}{|B \cup P|} = \frac{15}{42}$$

0.357

The intersection size of the n-gram sets

The union size of the n-gram sets

- B and P are the sets of the n-gram tokens of $P_{cd(i,j)}$ and $P_{cd(i,j)}$'s buggy chunk.

Repaired Location#1 of Fig. 2

A candidate patch of Fig. 3

Process of Patch Optimization

3) Patch Combination

It combines the top k candidate patches after patch ranking using Equation 4 and builds a set of combined patches.

$$k = \max(\{\, pn \mid pn^{nc} \leq MC,\ np \leq SP \})$$

- nc is the number of chunks in a multi-buggy chunk.
- MC is the number of maximum combined patches in $P'_{cd(i,j)}$ I set.
- np is the number of patches in a chunk to combine after patch ranking.

- A combined patch is evaluated and results in CO(Compilable), PL(Plausible), or CR(Correct).
  - When the patch passes all testcases, it results in PL(Plausible).
- When the evaluation is finished, we measure the performance, which is the number of repaired bugs per evaluated result.

Process of Patch Optimization

3) Patch Combination – The combination's example

Ranked Candidate Patches (Patch Ranking)

Ranked Candidate Patches#1

	Sim	Rank	isSelected
$P_{cd(1,1)}$	0.666	1	O
$P_{cd(1,4)}$	0.500	4	X
$P_{cd(1,5)}$	0.571	2	O
$P_{cd(1,6)}$	0.571	2	O

Ranked Candidate Patches#2

	Sim	Rank	isSelected
$P_{cd(2,1)}$	0.429	3	O
$P_{cd(2,3)}$	0.291	4	X
$P_{cd(2,5)}$	0.678	1	O
$P_{cd(2,6)}$	0.584	2	O

Ranked Candidate Patches of Fig. 3

$$k = \max(\{\, pn \mid pn^{nc} \le MC,\ np \le SP \,\}) \quad \text{with } nc \le 2,\ MC \le 10,\ SP \le 6$$

k: 3 — pn: {1, 2, 3}

- nc is the number of chunks a in a multi-buggy chunk.
- MC is the number of maximum combined patches in $P'_{cd(i,j)}$ I set
- np is the number of patches in a chunk to combine after patch ranking

Combined Patches (Patch Combination)

Combined Patches

	Candidate Patches		Candidate Patches
$P'_{cb(1)}$	$P_{cd(1,1)}$	$P'_{cb(6)}$	$P_{cd(1,6)}$
	$P_{cd(2,5)}$		$P_{cd(2,1)}$
$P'_{cb(2)}$	$P_{cd(1,1)}$	$P'_{cb(7)}$	$P_{cd(1,5)}$
	$P_{cd(2,6)}$		$P_{cd(2,5)}$
$P'_{cb(3)}$	$P_{cd(1,1)}$	$P'_{cb(8)}$	$P_{cd(1,5)}$
	$P_{cd(2,1)}$		$P_{cd(2,6)}$
$P'_{cb(4)}$	$P_{cd(1,6)}$	$P'_{cb(9)}$	$P_{cd(1,5)}$
	$P_{cd(2,5)}$		$P_{cd(2,1)}$
$P'_{cb(5)}$	$P_{cd(1,6)}$		
	$P_{cd(2,6)}$		

3^2 (9) ***combined patches***

Combined Patches of Fig. 3

2. Approach

Patch Optimization for the third problem

The patch optimization decreases the numbers of the generated candidate patches and their combined patches.

(b) Candidate Patches

Candidate Patches#1

$P_{cd(1,1)}$	private double num1 = 0.0;
$P_{cd(1,2)}$	private double num1 = 0.0;
$P_{cd(1,3)}$	private double num1 0.0;
$P_{cd(1,4)}$	<del/>
$P_{cd(1,5)}$	private double num3 = 0.0;
$P_{cd(1,6)}$	final double num1 = 0.0;

Candidate Patches#2

$P_{cd(2,1)}$	public double multiply () { return this.result; }
$P_{cd(2,2)}$	public double multiply () { return this.result; }
$P_{cd(2,3)}$	<del/>
$P_{cd(2,4)}$	public double multiply () { System.exit(0); return this.result; }
$P_{cd(2,5)}$	public double multiply () { if (num1 == 0.0 ll num2 == 0.0) return 0.0; this.result = 0.0; this.result = num1 * num2; return this.result; }
$P_{cd(2,6)}$	public double multiply () { if (num1 == 0.0 ll num2 == 0.0) return 1.0; this.result = 0.0; this.result = num1 * num2; this.result = num1 * num2; return this.result; }

Reduced Patch Space for 6^2 (36) combined patches

Filtered Candidate Patches (Patch Filtering)

Filtered Candidate Patches#1

	isRemoved
$P_{cd(1,1)}$	X
$P_{cd(1,2)}$	O (DP)
$P_{cd(1,3)}$	O (SE)
$P_{cd(1,4)}$	X
$P_{cd(1,5)}$	X
$P_{cd(1,6)}$	X

Filtered Candidate Patches#2

	isRemoved
$P_{cd(2,1)}$	X
$P_{cd(2,2)}$	O (DP)
$P_{cd(2,3)}$	X
$P_{cd(2,4)}$	O (TC)
$P_{cd(2,5)}$	X
$P_{cd(2,6)}$	X

More Reduced Patch Space for 4^2 (16) combined patches

Ranked Candidate Patches (Patch Ranking)

Ranked Candidate Patches#1

	Sim	Rank	isSelected
$P_{cd(1,1)}$	0.666	1	O
$P_{cd(1,4)}$	0.500	4	X
$P_{cd(1,5)}$	0.571	2	O
$P_{cd(1,6)}$	0.571	2	O

Ranked Candidate Patches#2

	Sim	Rank	isSelected
$P_{cd(2,1)}$	0.429	3	O
$P_{cd(2,3)}$	0.291	4	X
$P_{cd(2,5)}$	0.678	1	O
$P_{cd(2,6)}$	0.584	2	O

Combined Patches (Patch Combination)

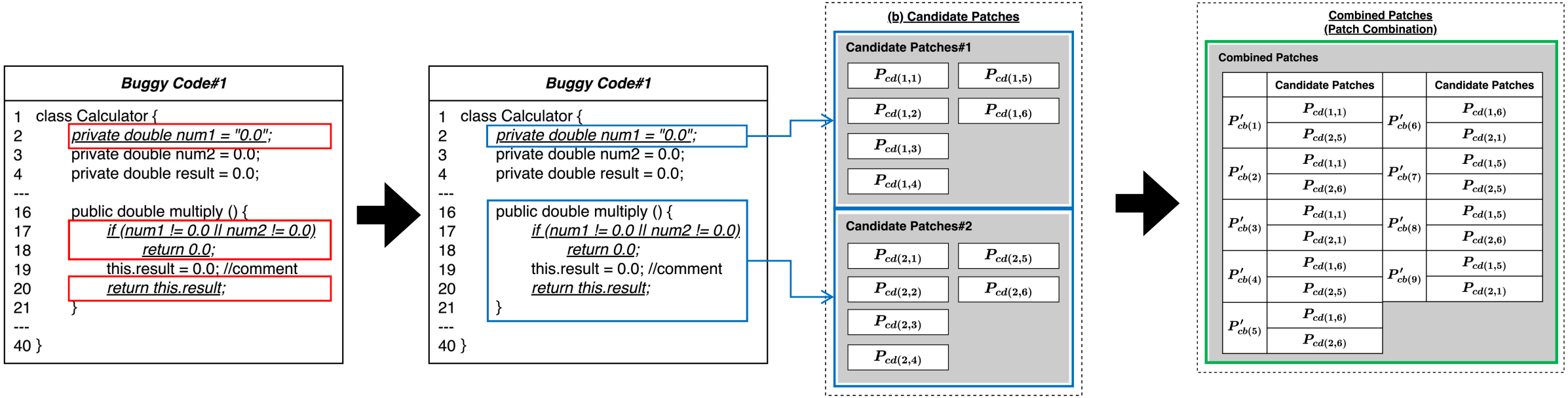
Combined Patches

	Candidate Patches		Candidate Patches
$P'_{cb(1)}$	$P_{cd(1,1)}$	$P'_{cb(6)}$	$P_{cd(1,6)}$
	$P_{cd(2,5)}$		$P_{cd(2,1)}$
$P'_{cb(2)}$	$P_{cd(1,1)}$	$P'_{cb(7)}$	$P_{cd(1,5)}$
	$P_{cd(2,6)}$		$P_{cd(2,5)}$
$P'_{cb(3)}$	$P_{cd(1,1)}$	$P'_{cb(8)}$	$P_{cd(1,5)}$
	$P_{cd(2,1)}$		$P_{cd(2,6)}$
$P'_{cb(4)}$	$P_{cd(1,6)}$	$P'_{cb(9)}$	$P_{cd(1,5)}$
	$P_{cd(2,5)}$		$P_{cd(2,1)}$
$P'_{cb(5)}$	$P_{cd(1,6)}$		
	$P_{cd(2,6)}$		

More Reduced Patch Space for 3^2 (9) combined patches

Buggy Block and Patch Optimization with Patch Space

The proposed buggy block and patch optimization reduce the patch space for multi-chunk bugs because it decreases the numbers of buggy chunks, candidate patches, and combined patches using method and filed dependencies.



Patch Space for 6 X 6 X 6 (108) combined patches by three buggy chunks

Reduced Patch Space for 6 X 6 (36) combined patches by two buggy chunks in a multi-buggy chunk

More Reduced Patch Space for 3 X 3 (9) combined patches

3. Experimental Setup

Datasets and Fine-tuning

1) Datasets
   - Training and validation dataset: Bugs2Fix [7]
   - Generation (benchmark) dataset: Defects4J [8]
2) Fine-tuning
   - max_token_length=512
   - batch_size=16
   - dropout_rate= 0.1

Optimization and Evaluation

1) Optimization
   - SP in Equation 4: 500
     (Five hundred candidate patches were generated per buggy block.)
   - MC in Equation 4: 10,000
     (The generated candidate patches were combined up to 10,000 patches.)
2) Evaluation
   - Timeout : 5.5 hours per module

4. Experimental Results

RQ1. Performance of MCRepair

- The results of RQ1 were the number of correctly fixed bugs.
- Perfect fault localization assumes known buggy locations without fault localization execution excluding its bias problem.

Table 1: RQ1. Comparison with Learning-based and Template-based APR techniques on Defects4J with Perfect Fault Localization.

(C: Chart, CL: Closure, L: Lang, M: Math, MC: Mockito, T: Time)

The highest number of each column is in bold.

Projects	C	CL	L	M	MC	T	Total
FixMiner	7	6	4	12	2	**3**	34 + 32
CoCoNut	7	9	7	16	**4**	1	44 + 22
TBar	**10**	13	**10**	13	3	**3**	52 + 12
CURE	**10**	14	9	19	**4**	1	57 + 9
Recoder	**10**	**21**	**10**	18	2	**3**	64 + 2
MCRepair	5	**21**	9	**24**	**4**	**3**	**66**

Fixed at least two and up to 32 more bugs than the baselines

4. Experimental Results

RQ1. Performance of MCRepair

- The results of RQ1 were the number of correctly fixed bugs.

Table 2: RQ1. Comparison with the state-of-the-art APR techniques on Defects4J with Perfect Fault Localization. (In terms of multi-chunk bugs)

Bug Types	CoCo-Nut	TBar	CURE	MC-Repair
Type 1. Single-chunk Single-location	31	32	**43**	37
Type 2. Single-chunk Multi-location	7	5	**8**	7
Type 3. Multi-chunk Multi-location	6	15	6	**22**
Total	44	52	57	**66**

46–266% relative imporvements

+ 16 (266% ↑) + 7 (46% ↑) + 16 (266% ↑)

Table 2: RQ1. Comparison with the state-of-the-art APR techniques on Defects4J with Perfect Fault Localization. (In terms of total bugs)

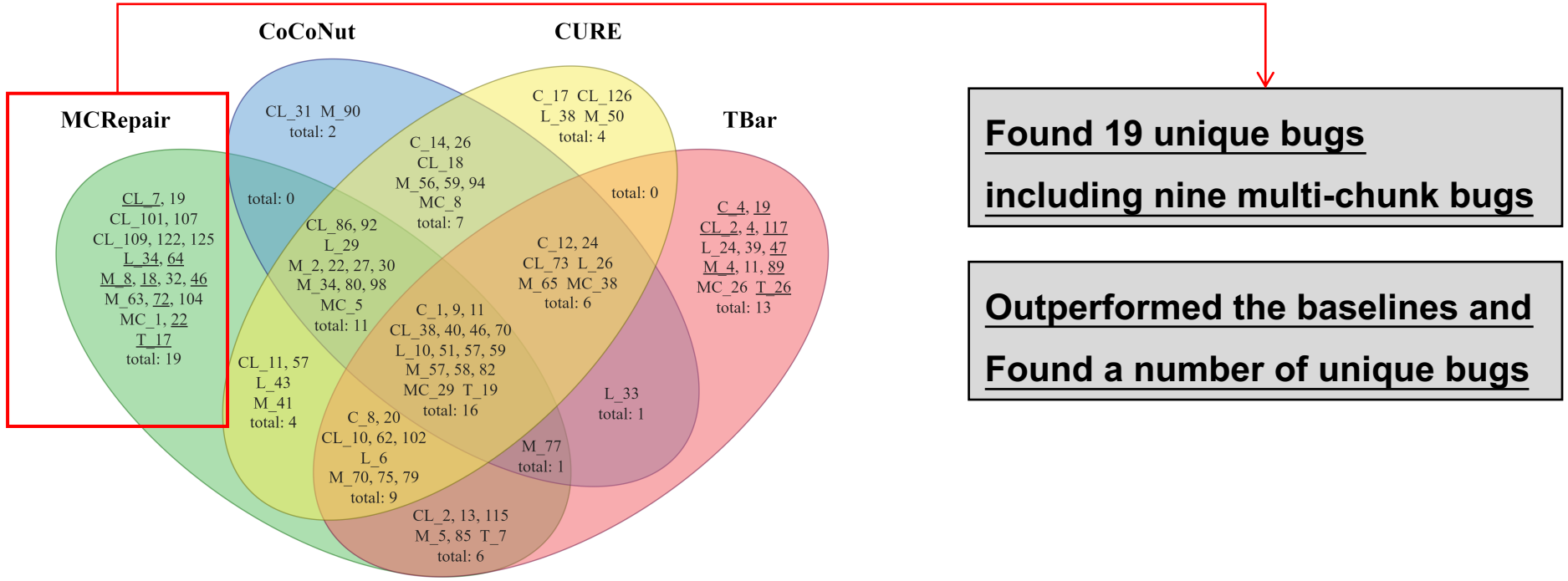
Bug Types	CoCo-Nut	TBar	CURE	MC-Repair
Type 1. Single-chunk Single-location	31	32	**43**	37
Type 2. Single-chunk Multi-location	7	5	**8**	7
Type 3. Multi-chunk Multi-location	6	15	6	**22**
Total	44	52	57	**66**

15–50% relative imporvements

+ 22 (50% ↑) + 14 (26% ↑) + 9 (15% ↑)

RQ1. Performance of MCRepair

- The results of RQ1 were the number of correctly fixed bugs.



Found 19 unique bugs including nine multi-chunk bugs

Outperformed the baselines and Found a number of unique bugs

Figure 4: RQ1. Venn Diagram about Table 2.

(Underlined: Unique Type 3 bugs)

RQ2. Generalizability of MCRepair

- The results of RQ2 were the same as them of RQ1.

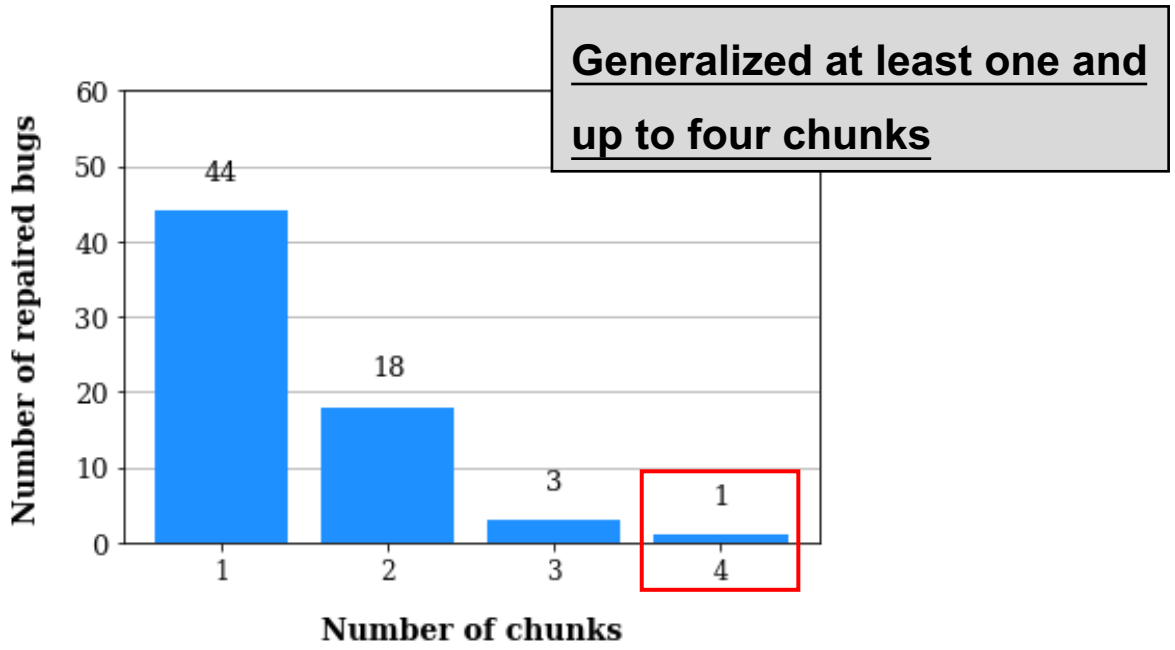


Figure 5: RQ2. Statistics per chunk range on Defects4J with Perfect Fault Localization.

RQ3. Contribution of MCRepair's each component

- The results of RQ3 were also the same as them of RQ1.

Table 3: RQ3. Sensitivity Analysis for MCRepair on Defects4J with Perfect Fault Localization. The results are displayed as x/y. x and y are the number of correctly repaired Type 3 bugs and total bugs. (−: without)

Projects	C	CL	L	M	MC	T	Total
−patch optimization	1/5	8/13	3/6	5/21	0/3	1/3	18/51 ▽ 4 / ▽ 15
−buggy contexts	1/6	8/14	4/10	6/23	0/2	1/4	20/59 ▽ 2 / ▽ 7
MCRepair	1/5	8/21	3/9	8/24	1/4	1/3	22/66

Removed two components such as patch optimization and buggy contexts in buggy blocks

The removal rates decreased by 15 and 7 bugs.

Fixed four Type 3 fewer bugs without patch optimization

The components affected the performance including multi-chunk bugs.

5. Conclusion

- We proposed an APR technique named “MCRepair” that utilized a buggy block, patch optimization, and CodeBERT to target multi-chunk bugs.
  - It addressed the three complex problems such as large patch space, dependencies of buggy chunks, and patch combination.
- On Defects4J, MCRepair correctly repaired 66 bugs, including 22 multi-chunk bugs.
  - We improved 46–266% performance than the baselines in terms of multi-chunk bugs.
  - We found 19 unique bugs, including nine multi-chunk bugs.
- In future work, we will improve MCRepair with its findings and limitations.

5. Conclusion

- The results of MCRepair were opened in the following URL.

https://github.com/kimjisung78/MCRepair

References in our paper

[4] Z. Feng *et al.* 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020. Association for Computational Linguistics*, Stroudsburg, PA, USA, 1536–1547. DOI: https://doi.org/10.18653/v1/2020.findings-emnlp.139

[5] J. Devlin, M.W. Chang, K. Lee, and K. Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL),* Association for Computational Linguistics, Stroudsburg, PA, USA, 4171–4186. DOI:https://doi.org/10.18653/v1/N19-1423

[6] S. Wang et al. 2020. Automated patch correctness assessment: how far are we? In *Proceedings of the IEEE/ACM 35th International Conference on Automated Software Engineering (ASE)*, ACM, New York, NY, USA, 968–980. DOI:https://doi.org/10.1145/3324884.3416590

[7] M. Tufano et al. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, ACM, New York, NY, USA, 832–837. DOI:https://doi.org/10.1145/3238147.3240732

[8] R. Just, D. Jalali, and M.D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 ACM International Symposium on Software Testing and Analysis (ISSTA)*, ACM Press, New York, New York, USA, 437–440.

[15] P. Adam et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*.

[16] T. Wolf et al. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP): System Demonstrations*, Association for Computational Linguistics, Stroudsburg, PA, USA, 38–45. DOI:https://doi.org/10.18653/v1/2020.emnlp-demos.6

Thank you!

Jisung Kim and Byeongjung Lee

{kimjisung78, bjlee}@uos.ac.kr

서울시립대학교
UNIVERSITY OF SEOUL

Q & A

Thank you so much!

Jisung Kim and Byeongjung Lee*

{kimjisung78, bjlee}@uos.ac.kr

서울시립대학교
UNIVERSITY OF SEOUL