Autorepairability

A New Software Quality Characteristic

Pongpop Lapvikai, Mahidol University

Yoshiki Higo, Osaka University

Chaiyong Ragkhitwetsagul, Mahidol University

Morakot Choetkiertikul, Mahidol University

SANER 2024 ERA Track

Automated Program Repair (APR)

is a technique to remove exposed bugs fully
and automatically without
human intervention.

Autorepairability

We introduce a new software characteristic,
autorepairability,
which measures how effective APR techniques
are in a specific project.

Measuring Autorepairability

Step 1: Generates mutants of source code SP by using a mutation testing technique MU. Each mutant μ ∈ M is a source code that is very similar to the given source code

Step 2: Applying APR tool A to mutants μ with test cases TP, a fix s is generated if it passes all TP tests. Set S represents all solutions A can generate over all mutants in M. Mutants passing all TP tests are not subjected to A.

Step 3: The technique calculates the ratio of the number of generated solutions |S| per the number of mutants |M|, i.e., the AR-ability(SP , TP , M, A) score.

$$AR\text{-}ability(S_P, T_P, MU, A) = \frac{|S|}{|M|}$$

Experiment

- We want to investigate whether methods with equivalent functionality have different autorepairability (i.e., whether some implementations are easy to fix bugs with APR techniques, while others are not?)

- We also aim to search for program elements that affect the autorepairability.

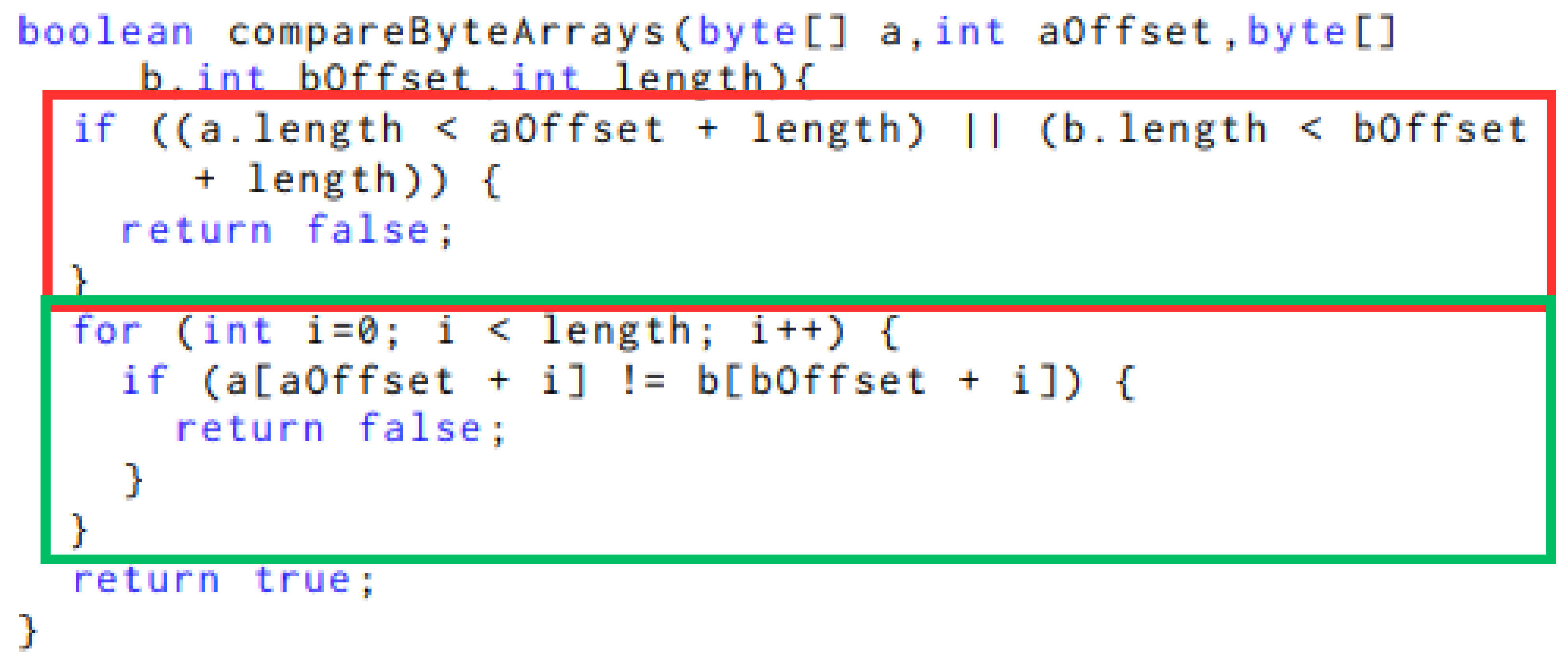
Research Questions

RQ1: What are the autorepairability scores of Java methods with functional equivalence?

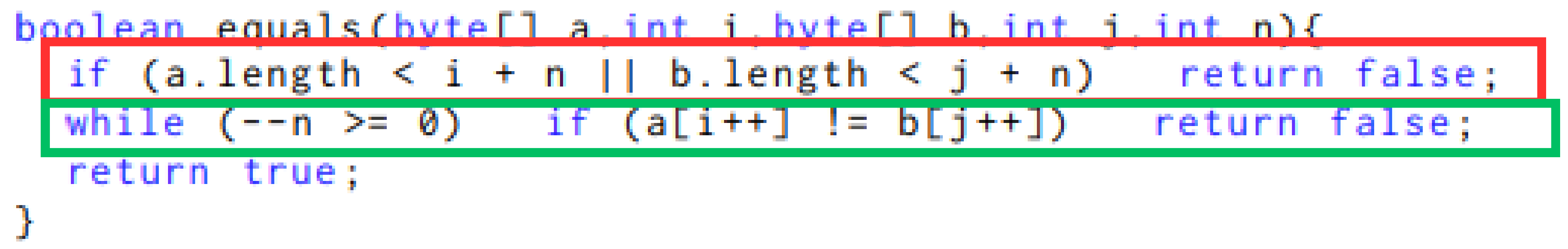
RQ2: What kinds of program elements affect autorepairability?

Methodology

Dataset : A set of 1,342 functional equivalent Java method pairs with test cases from Higo et al. (2022).



```
boolean compareByteArrays(byte[] a,int aOffset,byte[]
      b,int bOffset,int length){
  if ((a.length < aOffset + length) || (b.length < bOffset
      + length)) {
    return false;
  }
  for (int i=0; i < length; i++) {
    if (a[aOffset + i] != b[bOffset + i]) {
      return false;
    }
  }
  return true;
}
```



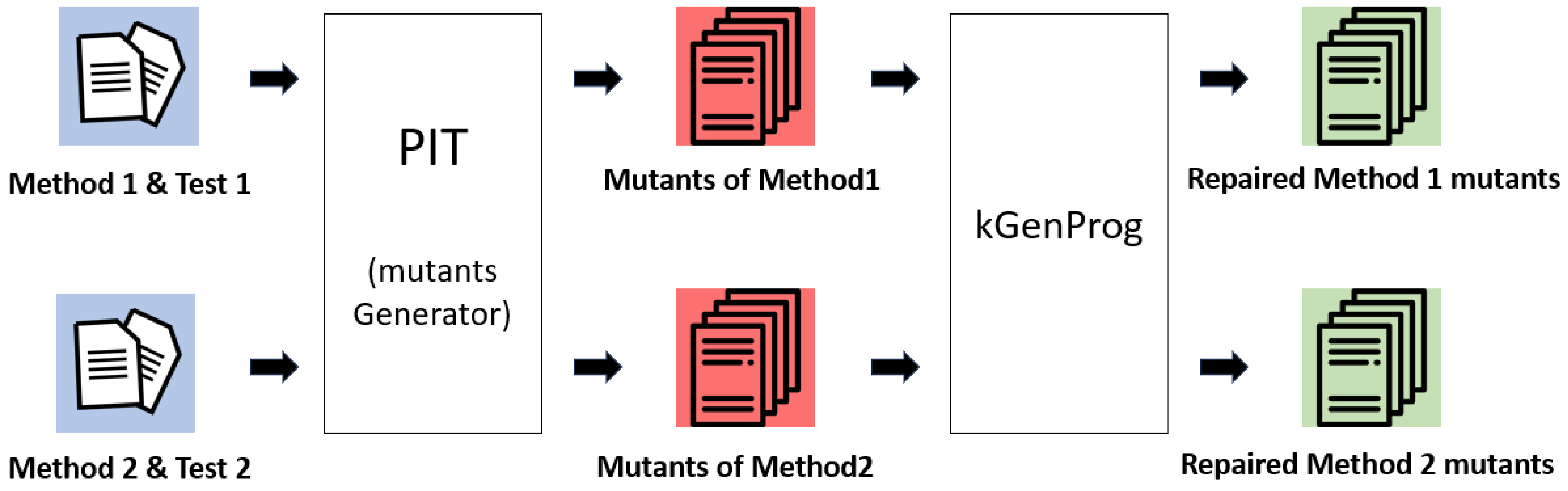
```
boolean equals(byte[] a,int i,byte[] b,int j,int n){
  if (a.length < i + n || b.length < j + n)    return false;
  while (--n >= 0)    if (a[i++] != b[j++])    return false;
  return true;
}
```

Each pair passes the unit test cases of each other.

Methodology (cont.)

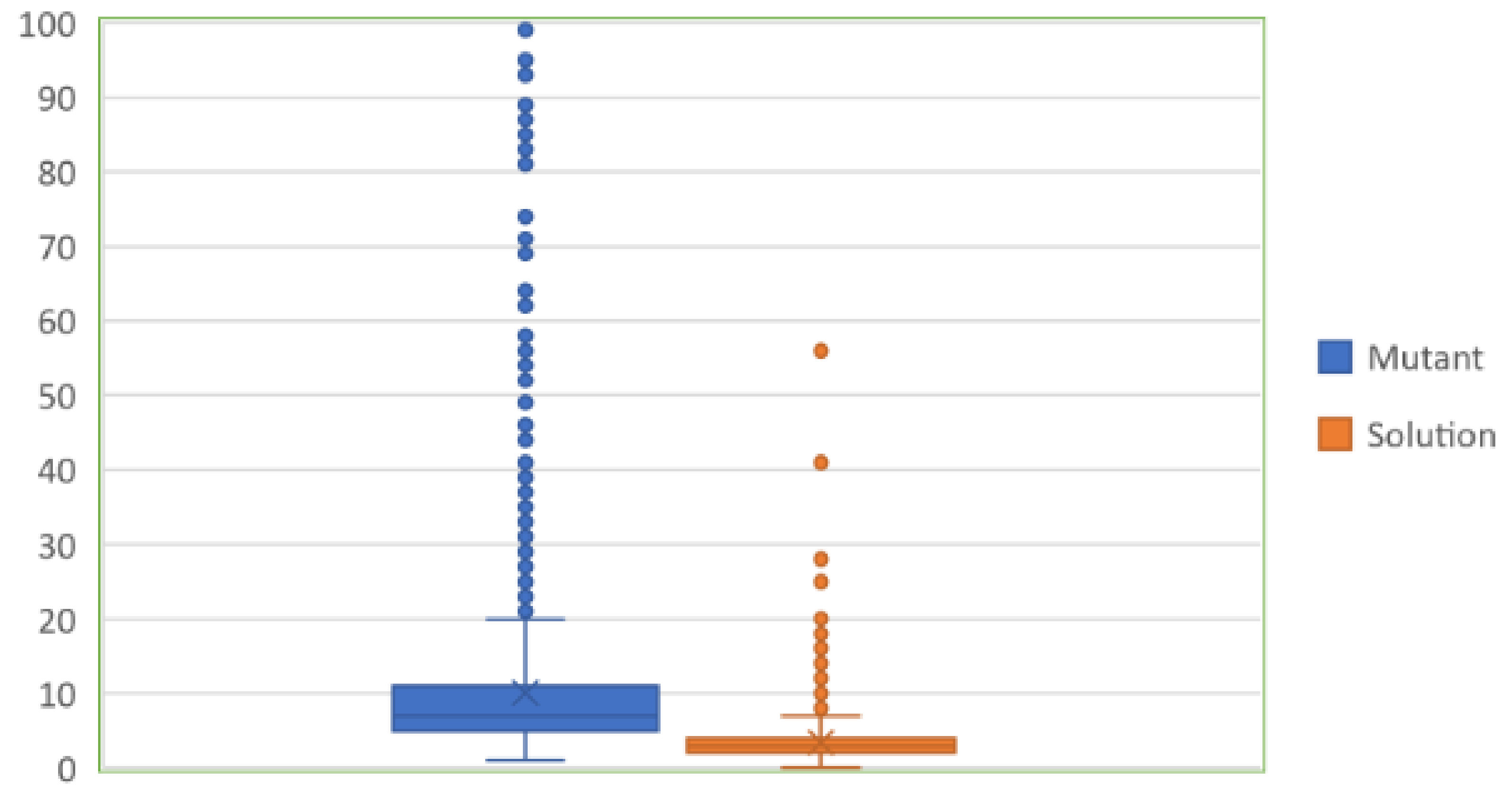
Mutation tool : A developed tool that generates mutants of the source code of a given program based on PIT's mutation operators.

APR tool : kGenProg, an APR tool of the generation-and-validation strategy.

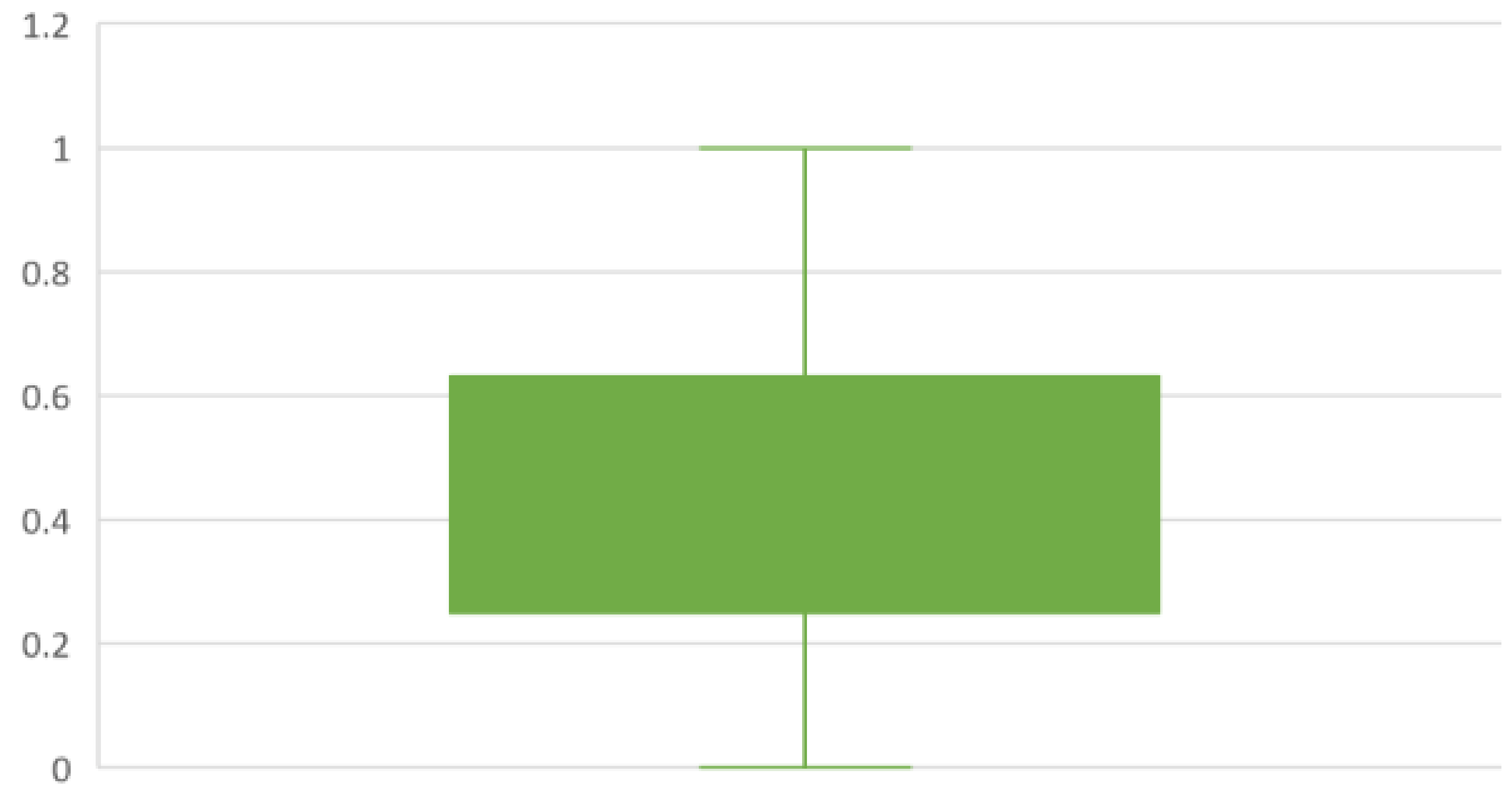


Results: RQ1

Only 1,282 method pairs from Higo et al. (2022) can successfully be executed.

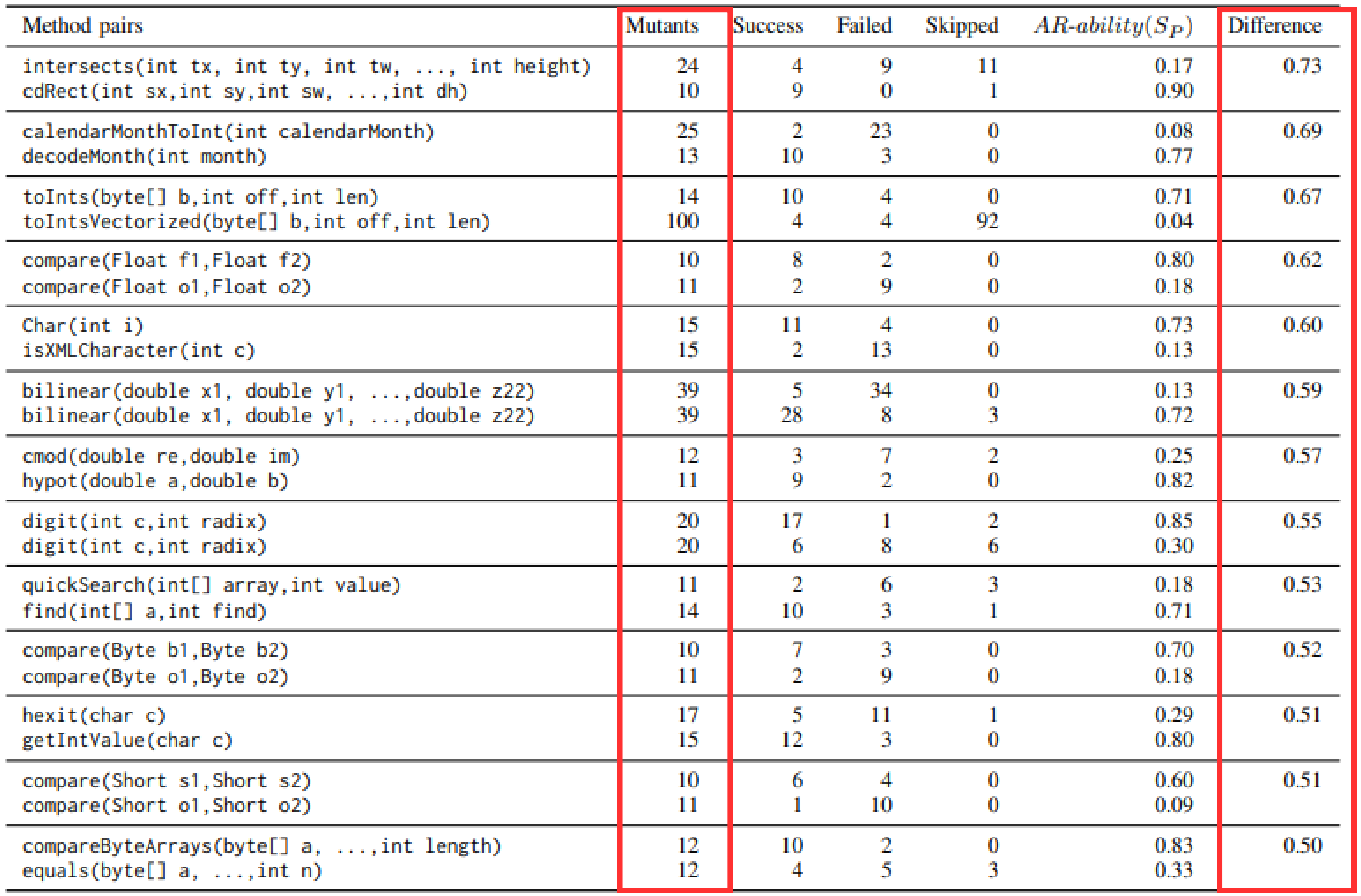


(a) No. of mutants and solutions



(b) $AR\text{-}ability(S_P)$ scores

Manual Investigation



Method pairs	Mutants	Success	Failed	Skipped	$AR\text{-}ability(S_P)$	Difference
`intersects(int tx, int ty, int tw, ..., int height)`	24	4	9	11	0.17	0.73
`cdRect(int sx,int sy,int sw, ...,int dh)`	10	9	0	1	0.90	
`calendarMonthToInt(int calendarMonth)`	25	2	23	0	0.08	0.69
`decodeMonth(int month)`	13	10	3	0	0.77	
`toInts(byte[] b,int off,int len)`	14	10	4	0	0.71	0.67
`toIntsVectorized(byte[] b,int off,int len)`	100	4	4	92	0.04	
`compare(Float f1,Float f2)`	10	8	2	0	0.80	0.62
`compare(Float o1,Float o2)`	11	2	9	0	0.18	
`Char(int i)`	15	11	4	0	0.73	0.60
`isXMLCharacter(int c)`	15	2	13	0	0.13	
`bilinear(double x1, double y1, ...,double z22)`	39	5	34	0	0.13	0.59
`bilinear(double x1, double y1, ...,double z22)`	39	28	8	3	0.72	
`cmod(double re,double im)`	12	3	7	2	0.25	0.57
`hypot(double a,double b)`	11	9	2	0	0.82	
`digit(int c,int radix)`	20	17	1	2	0.85	0.55
`digit(int c,int radix)`	20	6	8	6	0.30	
`quickSearch(int[] array,int value)`	11	2	6	3	0.18	0.53
`find(int[] a,int find)`	14	10	3	1	0.71	
`compare(Byte b1,Byte b2)`	10	7	3	0	0.70	0.52
`compare(Byte o1,Byte o2)`	11	2	9	0	0.18	
`hexit(char c)`	17	5	11	1	0.29	0.51
`getIntValue(char c)`	15	12	3	0	0.80	
`compare(Short s1,Short s2)`	10	6	4	0	0.60	0.51
`compare(Short o1,Short o2)`	11	1	10	0	0.09	
`compareByteArrays(byte[] a, ...,int length)`	12	10	2	0	0.83	0.50
`equals(byte[] a, ...,int n)`	12	4	5	3	0.33	

Results: RQ2

Code structure 1: Usage of ternary operator (3 occurrences)

```
return s1.shortValue()<s2.shortValue() ? -1 :
s1.shortValue()>s2.shortValue() ? 1: 0;
```

```
if (o1.shortValue()<o2.shortValue()) return -1;
if (o1.shortValue()>o2.shortValue()) return 1;
return 0;
```

APR score = 0.60 ✓

>

APR score = 0.09

Results: RQ2 (cont.)

Code structure 2: Combined logical expressions in a conditional statement (2 occurrences)

```
if (i >= 0x20 && i <= 0xD7FF)
return true;
```

APR score = 0.73 ✓

>

```
if (c < 0x20)   return false;
if (c <= 0xD7FF)   return true;
```

APR score = 0.13

Implications and Conclusion

We measured autorepairability of functional equivalent Java methods and compared their autorepairability scores.

We manually investigated the selected pairs with high differences in autorepairability scores and observed code constructs that may affect the autorepairability scores.

Introducing autorepairability to the characteristics of software quality will lead to new research in the future.