APPENDIX A
A BRIEF INTRODUCTION TO COGNITIVE COMPLEXITY

Cognitive Complexity is built on three basic rules that tally with the intuition of developers about the cognitive (or mental) effort required to understand a piece of code, and reflect the relative understanding difficulty of different control flows. Specifically, the three rules are defined as [17]:

Rule$_1$: Ignore readable shorthand structures
Cognitive Complexity ignores the code features that make code easier to read. Thus, it does not increment for method structure itself or the null-coalescing operators like `?.` and `??.` in C#.

Rule$_2$: Increment for breaks in the linear flow
Code features that break the linear flow of code usually require extra effort to understand that code. Thus, *Cognitive Complexity* increments for the code features like *loops* (e.g., `foreach`, `for`, `while`, and `do while`), *conditionals* (e.g., `switch`, ternary operators, `if`, `else if`, and `else`), `catch` statements, `goto|break|continue LABEL` statements, sequences of logical operators, and each method in a recursion cycle.

Rule$_3$: Increment for nested control flow
Intuitively, nested four `for` structures require more effort to understand than the same four structures when they are organized linearly, that is, nesting increases the effort to understand the code. *Cognitive Complexity* accounts for the difficulty increased by the nested control flow structures, and adds an increment for each level of nesting. For example, a nested *loop* structure increases the *Cognitive Complexity* by two.

To illustrate the idea to compute the *Cognitive Complexity* for a specific method, we give a simple code snippet for method `myMethod`, which is shown as follows:

```
public void myMethod(boolean flag) {
  int i = 0;
  int iters = 10;
  int times = 5;
  if (flag) { // +1 (Rule2: +1)
    for (; i < iters; i++) { // +2 (Rule2: +1, Rule3: +1)
      while(i < times) {} // +3 (Rule2: +1, Rule3: +2)
    }
  }
} // Cognitive Complexity of this method is 6
```

Obviously, method `myMethod` contains a nested `for` structure and a nested `while` structure. According to Rule$_1$ to Rule$_3$, *Cognitive Complexity* assigns different values for nested `for` and `while` structures; these values and the way to compute them are shown explicitly in the comments `//`. For example, the `for` structure is nested within the `if` structure, which increases the *Cognitive Complexity* value of `myMethod` by one according to Rule$_3$ (i.e., "Rule3: +1" in the comment). Moreover, the `for` structure breaks the linear flow of the code, which increases the *Cognitive Complexity* value of `myMethod` by one according to Rule$_2$ (i.e., "Rule2: +1" in the comment). Thus, the `for` structure increases the *Cognitive Complexity* value of `myMethod` by two (i.e., 1+1) in total. Then the *Cognitive Complexity* value of method `myMethod` is the sum of the *Cognitive Complexity* value of `for` and `while` structures.

APPENDIX B
A BRIEF INTRODUCTION TO BASELINE TECHNIQUES

In this work, nine existing techniques from the literature are selected as baseline techniques, which are briefly described as follows:

- No Prioritization (`NOP`): `NOP` does not perform any test case ordering, instead executing them in the original order defined by developers.
- Total Method based Prioritization (`TMP`) [36]: `TMP` sorts test cases in descending order of the number of methods they cover and executes them accordingly. Specifically, the priority score of each test case is equal to the number of methods it covers.
- Total Changed Method based Prioritization (`TCM`) [12]: `TCM` sorts test cases in descending order of the number of changed methods they cover and executes them accordingly. Specifically, a method is deemed changed if at least one of its statements has been modified (with comment changes excluded), and the priority score of each test case is equal to the number of changed methods it covers.
- Genetic-Algorithm-based Prioritization (`GAP`) [37]: `GAP` treats all permutations of test cases as candidate solutions and employs heuristics to guide the search for an optimal test execution order. For `GAP`, two key parameters—population size and maximum number of evolutionary generations—are both set to 100.
- Adaptive Random Prioritization (`ARP`) [37]: `ARP` selects the next test case that maximizes the minimum distance to the set of previously scheduled test cases. Specifically, the distance between two test cases is quantified using the *Jaccard Index*—computed based on the sets of methods covered by each of the two test cases.
- Fault Proneness Prediction based Prioritization (`FPP`) [14]: `FPP` sorts test cases in descending order of the sum of fault-proneness values of the methods they cover and executes them accordingly. Specifically, the fault-proneness of each method is predicted using historical fault data and a neural network model, and the priority score of each test case is defined as the sum of fault-proneness values of the methods it covers. As `FPP` relies on historical fault data, for a given version pair (u, v), u starts from the second considered version of the project—since the first considered version lacks any accumulated fault history to support fault-proneness prediction.
- Total Changed Method Prioritization with Quotient Set (`TCM-QS`) [13]: Similar to our proposed `FI-HRR`, `TCM-QS` partitions test cases into groups and adopts a round-robin selection of one test case from each group. However, it differs in two key aspects: (i) test cases are grouped based on the number of changed methods they cover (rather than their *FPDT* values), and (ii) test case

selection within each group is determined by the number of methods they cover.

- Method-Level Tree Kernel Prioritization (MTK) [13]: MTK sorts test cases in descending order of the sum of the *CMM* values of the methods they cover and executes them accordingly. Specifically, the priority score of each test case is defined as the sum of the *CMM* value of the methods it covers.
- Method-Level Tree Kernel Prioritization with Quotient Set ((MTK−QS) [13]: Similar to our proposed FI−HRR, MTK−QS partitions test cases into groups and adopts a round-robin strategy to select one test case from each group. However, it differs in two key aspects: (i) test cases are grouped based on the sum of the *CMM* values of the methods they cover (as opposed to their *FPDT* values), and (ii) test case selection within each group is determined by the number of methods they cover.