Hybrid Regression Test Selection by Integrating File and Method Dependences

Guofeng Zhang*†
National University of Defense Technology
College of Computer
Changsha, China
zhangguofeng16@nudt.edu.cn

Luyao Liu*†
National University of Defense Technology
College of Computer
Changsha, China
lyliu@nudt.edu.cn

Zhenbang Chen‡†
National University of Defense Technology
College of Computer
Changsha, China
zbchen@nudt.edu.cn

Ji Wang‡†
National University of Defense Technology
College of Computer
Changsha, China
wj@nudt.edu.cn

ABSTRACT

Regression Testing Selection (RTS) reduces the cost of regression testing by only running test cases affected by code changes. Due to the bottleneck of single granularity analyses, the latest RTS techniques tend to analyze with mixed granularities. However, a better synergy of the existing RTS techniques is still challenging. Besides, we have found that once existing RTS approaches use static method-level analysis, handling external library callbacks is difficult, leading to the missed selection of affected test cases.

To address these difficulties, we introduce a new hybrid RTS approach, **JcgEks**, which enhances Ekstazi by integrating static method call graphs. It combines the advantages of dynamic and static analyses, improving precision from class-level to method-level and reducing end-to-end time without sacrificing safety. More importantly, **JcgEks** safely handles external library calls. Besides, we propose a new safety metric and implement the checking tool called Checker to evaluate the safety of RTS tools. We compared **JcgEks** with four baseline RTS tools in 1000 revisions across 20 open-source projects. The experimental results demonstrate that, compared with the state-of-the-art RTS tool FineEkstazi, **JcgEks** had the same level of end-to-end testing time and number of selected test classes, while FineEkstazi was confirmed to miss test classes in the experiment. Compared with Ekstazi, **JcgEks** has reduced end-to-end time by 29% and the number of test classes by 30.9% while ensuring safety.

*Guofeng Zhang and Luyao Liu are co-first authors.
†Also with State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology.
‡Zhenbang Chen and Ji Wang are the corresponding authors.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695525

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Regression test selection, regression testing, change-impact analysis

ACM Reference Format:
Guofeng Zhang, Luyao Liu, Zhenbang Chen, and Ji Wang. 2024. Hybrid Regression Test Selection by Integrating File and Method Dependences. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695525

1 INTRODUCTION

Software testing [1, 2] is the industry's current *de facto* method for improving software quality. Regression testing [14, 55] is commonly adopted in software development to check the correctness of the changes for passing the old test cases. However, rerunning all the test cases after each code change can be very expensive and even unacceptable [12, 15, 35, 43, 45, 59], especially for the large-scale software projects, which may contain thousands or even more than tens of thousands of test cases and need a very long time for complete rerunning. To address this problem, regression testing selection (RTS) techniques [7, 13, 18, 27, 32, 48] have been proposed.

RTS aims to improve regression testing efficiency by selectively identifying and executing only a subset of test cases affected by code changes. The typical RTS techniques require building the dependences between tests and program code. When code changes occur, the RTS identifies which tests are affected by these changes and runs just those tests again. A *perfect* RTS technique [47] should be 1) *Safe*: it selects all tests affected by code changes; 2) *Precise*: it does not select any test that is not affected by code changes.

RTS techniques can be divided into two categories: dynamic and static. Dynamic RTS [17, 22, 36, 38, 58] collects dependences by testing previous versions, whereas static RTS [24–26, 44] infers dependencies using static analysis. Besides, according to dependence granularities, RTS can be divided into control flow level [22, 42], method-level [8, 37, 50, 58], file-level [17, 25, 52, 57, 61], package-level [11], and module-level [10, 15, 21, 23, 45, 46] techniques. Many techniques prioritize precision and focus on fine-grained analysis,

such as control flow and method-level analyses, which may result in significant overhead. More recent techniques focus on coarser-grained analysis, such as file and module-level analyses, which can reduce overhead but may compromise precision.

Recent studies have demonstrated that file-level RTS (FRTS) has the shortest end-to-end time compared with the other single-granularity RTS techniques [17, 20, 25, 32, 57]. Although FRTS has several advantages over other methods, it still suffers from imprecision and may select redundant tests unaffected by code changes. Li *et al.* [31] point out that static method-level RTS (MRTS) is superior to static FRTS in reducing tests, fault detection efficiency, and testing cost. On the other hand, fine-grained analysis offers higher precision but is often considered impractical in regression test processes due to the overhead. For example, dynamic MRTS requires instrumentation for each method, which may introduce significant runtime overhead and violate the testing's time constraints. Therefore, an alternative option is static MRTS, which does not impose an additional runtime burden. However, static MRTS has limitations, such as inherited imprecision or unsoundness due to reflection or dynamic dispatch. Consequently, some hybrid approaches combine different RTS technologies to improve RTS's effectiveness, such as the combination of dynamic and static RTS techniques [30, 44] and the combination of the RTS techniques with different analysis granularities [32, 46, 52]. However, the balance between precision and efficiency is still a challenging problem for RTS.

Besides precision and testing efficiency, RTS's safety is also an important problem. RTS'safety [39–41, 53] can also be classified into different levels based on dependence granularities. For example, method-level safety entails selecting all test cases that can access the changed methods during execution. There also exists work on the safety of static RTS [30, 44], which focuses more on the dynamic features of the language, such as reflection and dynamic binding, without discussing the influences of external library callbacks. Furthermore, the expense of conducting the static analysis of all external libraries is unaffordable. As far as we know, we are the first to investigate the impact of external library callbacks on RTS safety and emphasize the importance of considering external library callbacks when using static RTS technology.

To further optimize the trade-off between RTS's precision, end-to-end testing time, and safety, we have proposed a new hybrid RTS method, called **JcgEks**, that integrates dynamic FRTS with static method-level analysis. On the one hand, we use a finer-grained static analysis to refine the selection results of dynamic FRTS further; on the other hand, dynamic analysis not only offers runtime information to compensate for the safety problem of static analysis but also enables an incremental static analysis for static MRTS. To clarify, dynamic and static analyses are well synergized and complement each other. This way, the precision can be improved without extra instrumentation overhead, and irrelevant test classes can also be safely excluded at the method-level.

To tackle the safety problem caused by external library callbacks, we over-approximate the method invocations by types. Besides, to evaluate the safety of RTS tools, we have proposed a new safety metric with respect to missed test classes and implemented a checking tool, Checker. Specifically, Checker inserts output statements into all changed methods at runtime and extracts the relation between test classes and the changed methods from execution logs. This process helps precisely identify the test classes affected by the changes. By comparing the test classes identified by the RTS tool as affected by the changes, we can determine the number of test classes missed by the RTS tools.

We have implemented our method based on Ekstazi [16] and used java-callgraph[1] for the static method-level analysis. We conducted an experimental evaluation from 3 aspects: reducing the testing scale, reducing end-to-end time, and missing tests. The experiment contains 1000 CI versions from 20 projects in the GitHub community. To demonstrate the effectiveness of **JcgEks**, we compared it with four state-of-the-art RTS tools: Ekstazi[2], FineEkstazi[3], HyRTS[4], and STARTS[5]. The reduced testing scale and end-to-end time demonstrate the precision and effectiveness of **JcgEks**. The experiment results indicate that: 1) On average, compared to Ekstazi, FineEkstazi, STARTS, and HyRTS, **JcgEks** can reduce the testing scale by 29.0%, 3.1%, 78.2%, and 73.1%, respectively; 2) **JcgEks** can reduce the end-to-end time by 30.9%, 12.6%, 56.5%, and 86.9%, respectively; 3) On the experimental benchmark, **JcgEks** has the same safety as Ekstazi, while FineEkstazi has been proven to be unsafe by our proposed inspection tool Checker.

The main contributions of this article are as follows:

- We propose a new hybrid RTS method **JcgEks** that synergizes dynamic FRTS and static MRTS to improve RTS's effectiveness further.
- We are the first to investigate the scenarios involving external library callbacks and propose an approach to ensuring RTS's safety. Besides, we propose a metric to evaluate the safety of RTS tools with respect to missed test classes.
- We have implemented a prototype tool for **JcgEks**. In addition, we have developed a safety checking tool called Checker with respect to the safety metric.
- We evaluated **JcgEks**'s effectiveness by conducting a large-scale experiment on 1000 CI revisions of 20 real-world projects. **JcgEks** reduces the number of test classes by 29.0% and end-to-end time by 30.9% compared to the Ekstazi, and Checker confirms the unsafety of FineEkstazi.

2 MOTIVATION EXAMPLE

It is common for developers to utilize libraries or frameworks during development. These libraries often require developers to implement specific interfaces and pass the implemented classes as parameters back to the library for API calls. This scenario is known as external library callbacks. However, existing static analysis techniques face challenges in analyzing external libraries, and there is a lack of research addressing this issue. To our knowledge, we are the first to address handling external library callbacks in RTS.

Besides, traditional RTS techniques often employ dynamic or static analysis at specific levels of granularity. To illustrate this, we provide an example demonstrating the techniques used for regression testing selection at the dynamic file-level and static method-level. We have included two figures that depict the example program and its corresponding dependences. Figure 1 describes a

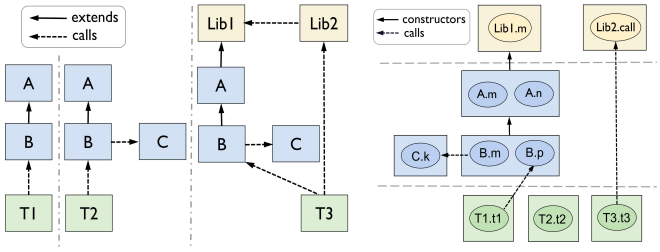
[1] https://github.com/gousiosg/java-callgraph
[2] http://ekstazi.org
[3] https://github.com/EngineeringSoftware/fine-ekstazi
[4] http://hyrts.org
[5] https://github.com/TestingResearchIllinois/starts

```java
//Library classes
interface Lib1{
  void m(); /*abstract method*/
}
class Lib2{
  void call(Lib1 l1){ l1.m(); }
}
```

```java
//Application classes
class A implements Lib1{
  @Override void m(){...}
  void n( ) {...} // changed method
}
class B extends A{
  @Override void m(){ C c = new C(); c.k();}
  void p() {...}
}
class C{
  void k( ) {...} // changed method
}
```

```java
//Test classes
class T1{
  void t1(){ B b = new B(); b.p(); }
}
class T2{
  void t2(){
    Class<?> clz = Class.forName("B");
    Constructor c = clz.getConstrutor();
    Object o = c.newInstance();
    Method m = clz.getMethod("m");
    m.invoke(o);
  }
}
class T3{
  void t3(){
    Lib2 lib2 = new Lib2();
    lib2.call(new B());
  }
}
```

Figure 1: Motivation Example



(a) Dynamic FRTS **(b) Static MRTS**

Figure 2: Different dependence representations

Java program *P* divided into three parts. The interface `Lib1` and class `Lib2` are in the external library that program *P* relies on. The classes `A`, `B`, and `C` are the classes under test (represented as *CUT* later). The classes `T1`, `T2`, and `T3` are the test classes. For the sake of brevity, we omit the constructors of all classes in the subsequent descriptions. *We assume that the methods* `A.n()` *and* `C.k()` *are changed in the revision, which only affects the test classes* `T2` *and* `T3`.

Figure 2 shows the different dependence representations for two RTS technologies. Figures 2a and 2b show the dependence graphs of dynamic FRTS and static MRTS techniques, respectively. In Figure 2a, the dashed lines illustrate the call dependence relationships between classes, while the solid lines represent the inheritance relationships between classes. Both types of lines indicate class-level dependences. In Figure 2b, the dashed lines denote

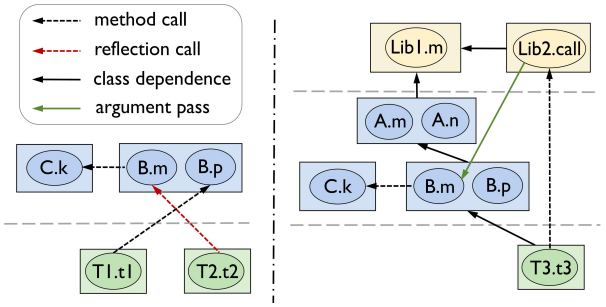


Figure 3: The dependence representation of JcgEks

the call dependence relationships between methods, while the solid lines represent the call dependence relationships between subclass constructors and parent constructors. Both types of lines indicate method-level dependences.

Dynamic FRTS. This technique (the representative tool is Ekstazi) dynamically collects dependent files while executing test cases. Taking `T2` as an example, it calls `B.m()`, where `B.m()` calls `C.k()`. Given that class `B` extends class `A`, the constructors of classes `B`, `A`, and `C` are all executed consecutively. Therefore, `T2` depends on classes `A`, `B`, and `C`. While `T3` relies on classes `B`, `A`, `C`, `Lib1` and `Lib2` due to the upcasting of `B` and the invocation of `Lib2.call()`. Suppose any method or field in the dependence file of these test classes is semantically changed, causing the checksum of the bytecode file to change (*e.g.*, the method body of `A.n()` is changed in the new revision). In that case, although unnecessary, `T1` needs to be selected for re-execution. Therefore, due to its coarse-grained analysis, FRTS may select many unnecessary test classes for retesting, which is safe but imprecise.

Static MRTS. This technique is based on the method call graph but lacks some information in runtime, such as reflection and dynamic binding. As shown in Figure 2b, due to the limitation of static analysis in handling reflections, `T2.t2()` is unable to collect any method dependences on *CUT*. `T3` can only collect the dependences of `Lib2.call()`, which will miss the method `B.m()` that `T3` will actually execute. Therefore, the test classes `T2` and `T3` are missed. There are two reasons for this problem: 1) Static analysis methods face challenges in addressing issues related to dynamic binding; 2) Static analysis methods may only analyze the code within the project but ignore the analysis of the external library code.

Discussion. In general, although dynamic FRTS has advantages over other granularity RTS in end-to-end time, it is still imprecise as it selects many unnecessary test classes. The dynamic MRTS, although much more precise, can incur significant instrumentation overhead. Although static RTS does not have instrumentation overhead, its limitation to tracking the actual execution of code and analyzing external libraries can easily miss necessary tests (*e.g.*, the reflection in `T2.t2()`, dynamic binding, and external library callbacks in `T3.t3()`) and also additional selections. Therefore, our work aims to improve the efficiency and precision of dynamic FRTS through a hybrid RTS technique. Besides, we aim to select more precise test classes without compromising safety while minimizing the end-to-end regression testing time as much as possible.

JcgEks. To handle the situations in the motivation example, Figure 3 shows the dependences constructed by **JcgEks**. Among them, only `B.p()` remains for method `T1.t1()`'s dependence, while

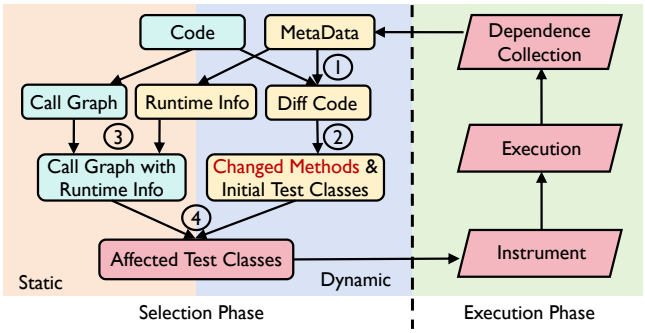


Figure 4: The framework of JcgEks

`T2.t2()` has added a dependence for reflection calling `B.m()`, and `T3.t3()` can handle the scenario of external library `Lib2.call()` callback `B.m()`. These three examples highlight the benefits of our approach. Firstly, in example `T1.t1()`, we showcase the improved precision in dependence for FRTS. Secondly, in `T2.t2()`, we emphasize the enhanced safety achieved by analyzing reflections. Lastly, in `T3.t3()`, we demonstrate the reinforced safety measures when analyzing external library calls.

Then, for the test class `T1`, the changed method `A.n()` does not affect any test classes. In contrast, dynamic FRTS, due to its coarse granularity, will select `T1`. However, **JcgEks** does not choose `T1` because it combines method call graphs to improve the selection's precision. For the test class `T2`, it is difficult for static analysis to handle reflection. As a result, the call graph does not contain an edge from `T2.t2()` to `B.m()`, leading to the missing test class `T2` by static MRTS. In contrast, **JcgEks**, by collecting the reflection-based method call information during runtime, will supplement the call graph with the missing edge from `T2.t2()` to `B.m()`, thereby enabling the selection of the affected test class `T2`. For the test class `T3`, the changed method `C.k()` affects `B.m()` on the method call graph, thereby impacting class `B`. **JcgEks** checks whether the types of each parameter in `Lib2.call()` are affected by the code changes, then **JcgEks** identifies that the parameter's type `Lib1` is a superclass of class `B`. Here, **JcgEks** *over-approximates* that there is a potential callback through `Lib2.call` to `B.m()`, thereby reaching the changed method `C.k()`. Hence, **JcgEks** selects `T3`, which static MRTS misses.

3 APPROACH

Figure 4 shows **JcgEks**'s framework, which follows the typical dynamic RTS technique and consists of two phases: the selection phase and the execution phase. During the execution phase, **JcgEks** instruments and collects two types of runtime information for each test class: reflections and class dependences, which are stored in the metadata used in the next revision's selection phase. In the selection phase, **JcgEks** selects all test classes that are affected by the changes in the revision, which will be executed in the execution phase. To improve the precision of test class selection, **JcgEks** uses the method call graph and refines it with the runtime information collected in the execution phase, reducing the number of test classes affected by the revision's changes.

The primary innovation of **JcgEks** lies in the selection phase, as illustrated in Figure 4. This stage can be described in four steps:

- Calculate the checksum of each class and method in the new revision based on code differences (3.1).
- Comparison of the checksum differences to initially select the affected testing classes at the file-level (3.2).
- Generation of a method call graph enriched with runtime information (3.3).
- Further refinement of the affected test classes(3.4).

In Section 3.5, we propose a metric to evaluate the safety of RTS at the method-level and an approach Checker for obtaining the metric's value, providing a basis for comparing the safety of different RTS tools. Finally, in Section 3.6, we discussed how **JcgEks** handles field changes and the design philosophy of Checker.

3.1 Metadata Computation

A Java project P is a tuple (C, M, E), where C represents the class set of the project, M is the method set of the project, and $E \subseteq M \times M$ is the edge set representing the method calls of P. In addition, we use $\mathcal{T} \subseteq C$ to represent the set of test classes in P. We introduce the following definitions, where ck is the hashing function of the Java bytecode after removing debugging information [32]: $CH = \{c \mapsto ck(c) \mid c \in C \wedge ck(c) \in \mathbb{Z}\}$ maps each class in P to their hash values; $MH = \{m \mapsto ck(m) \mid m \in M \wedge ck(m) \in \mathbb{Z}\}$ is the mapping of all methods in P to their hash values. A method $m = \langle class, name, paras \rangle$, representing the class to which the method belongs, the method name, and the set of parameter types.

JcgEks's metadata differs from that of Ekstazi. The metadata structure consists of four maps: $\langle D, H, R, S \rangle$, where

- $D = \{t \mapsto CH_t \mid t \in \mathcal{T} \wedge CH_t \subseteq CH\}$ stores the class dependence hashes of each test class t.
- $H = \{c \mapsto MH_c \mid c \in C \wedge MH_c \subseteq MH \wedge c.methods = dom(MH_c)\}$ stores the checksums of all methods in each class c, where $c.methods$ represents the method set of class c, and $dom(f)$ represents the domain set of function f.
- $R = \{t \mapsto E_r \mid t \in \mathcal{T} \wedge E_r \subseteq E\}$ records the reflection-based method invocations when executing t, where each invocation pair in E_r is from method m_{src} to m_{dst}.
- $S = \{c \mapsto C_s \mid c \in C \wedge C_s \subseteq C\}$ stores each class's direct superclass and directly implemented interfaces.

Both R and S are obtained at runtime to tackle the limit of static analysis, *e.g.*, missing superclasses in external libraries. We instrument reflection code and construction methods to obtain R and S, respectively. The first step needs to calculate the two checksum maps D and H, denoted as D_{new} and H_{new} in the new revision.

3.2 Test Classes Selection

Algorithm 1 shows the second step of **JcgEks**, which uses the difference between the old and new checksums to get the changed methods and select the test classes affected by the changed classes. The inputs of the algorithm are the new revision checksums D_{new} and H_{new} calculated in the first step, the old revision checksums D_{old} and H_{old} extracted from $MetaData$, and the new revision's all test class set $\mathcal{T}$. The output is the set of modified methods and the preliminarily selected set of test classes $\mathcal{T}'$.

Algorithm 1 checks for any dependent class changes for each test class t (Line 2). We use $c \in dom(D_{new}[t])$ to denote that c is a key of the map $D_{new}[t]$ (Line 3). If the checksums of the class c are different between the old and new revisions(Line 4), the test class t will be added to the affected class set $\mathcal{T}'$ (Line 5), and then traverse all methods of class c in the new revision (Line 6). If method m also

Algorithm 1 Get changed methods and initial test classes

Input: $\mathcal{T}, D_{new}, D_{old}, H_{new}, H_{old}$
Output:
$\mathcal{T}'$: The set of initial selected test classes
$\mathcal{M}'$: The set of changed methods.

```
1:  T', M' ← ∅, ∅
2:  for t ∈ T do
3:      for c ∈ dom(D_new[t]) do
4:          if D_new[t][c] ≠ D_old[t][c] then
5:              T' ← T' ∪ {t}                          ▷ affected test classes
6:              for m ∈ dom(H_new[c]) do
7:                  if m ∈ dom(H_old[c]) then
8:                      if H_new[c][m] ≠ H_old[c][m] then
9:                          M' ← M' ∪ {m}              ▷ changed methods
10:                     end if
11:                 else
12:                     M' ← M' ∪ {m}                  ▷ new methods
13:                 end if
14:             end for
15:         end if
16:     end for
17: end for
18: return (T', M')
```

exists in the old revision, but the checksum changes, it indicates that m is a changed method and is added to the set $\mathcal{M}'$ (Lines 7-9). If m does not exist in the old revision, it means that m is a new method added in the new revision, and it is also added to $\mathcal{M}'$ (Lines 11-12). Please note that if t is a new test class, it will not be filtered through subsequent steps but will be enforced.

Based on the traditional dynamic FRTS step of selecting the initial set of affected test classes $\mathcal{T}'$, our algorithm incorporates method-level checksum checking to calculate the changed methods simultaneously. This step outputs the initially selected set $\mathcal{T}'$ of the affected test classes and the set $\mathcal{M}'$ of the changed methods, which will be used to refine test class selection in the fourth step.

3.3 Call Graph Generation

The third step is to merge the method call graph CG constructed by static analysis with R in $MetaData$ to generate CG'. The definition of CG is as follows: $CG = \{m_{src} \mapsto M_{dst} \mid m_{src} \in M \wedge M_{dst} \subseteq M\}$ is a map whose key represents the caller method, and whose value is a set of callee methods. R records the reflection-based method invocations obtained at runtime. We instrument the $Constructor.newInstance()$ and $Method.invoke()$ to trace the reflection-based invocations of the construction and normal methods, respectively. Note that all caller methods used as keys in the CG belong to the analyzed project, and the values may contain callee methods from the library. Because the key of map R is a test class, each time the test is executed, R only updates the items related to the test class execution, and the other parts are not updated. The data structure of CG' is also a map, where keys are the callers, and values are the sets of callees. When merging R into CG, add the m_{dst} of each pair (m_{src}, m_{dst}) in $R[t]$ to the callee set $CG[m_{src}]$, indicating that m_{src} calls m_{dst} by reflection.

When an object calls a method after its upcasting, static analysis may only obtain a method call to a superclass. However, due to dynamic binding, the corresponding subclass method is executed. Therefore, adding method call edges to the call graph is necessary, from superclasses' to subclasses' methods. Although this approach may result in selecting redundant test cases during the subsequent traversal of the call graph, this over-approximation improves safety.

Algorithm 2 Refine the affected test classes

Input: $\mathcal{T}', \mathcal{M}', CG', S, D_{new}, Lib$.
Output: $\mathcal{T}^*$: The set of affected test classes.

```
1:  T* ← ∅
2:  AM ← {m_c | ∃m_e ∈ M' • m_c can reach m_e in CG'}
3:  AC ← {c | ∃m ∈ AM • c = m.class ∨ c∈Ancestor(S, m.class)}
4:  for t ∈ T' do
5:      W ← t.methods
6:      while W ≠ ∅ ∧ t ∉ T* do
7:          m_src ← W.pop()
8:          if m_src ∈ AM then
9:              T* ← T* ∪ {t}                          ▷ select test classes
10:             break
11:         end if
12:         for m_dst ∈ CG'[m_src] do
13:             if m_dst.class ∈ Lib then
14:                 if m_dst.class ∈ AC ∨ m_dst.paras ∩ AC ≠ ∅ then
15:                     T* ← T* ∪ {t}                  ▷ external library check
16:                     break
17:                 end if
18:             else
19:                 if m_dst.class ∈ dom(D_new[t]) then
20:                     W ← W ∪ {m_dst}                ▷ expand worklist
21:                 end if
22:             end if
23:         end for
24:     end while
25: end for
26: return T*
```

This step builds a method call graph that contains runtime information. We *incrementally* generate a static call graph for the new code version when the code is changed. Besides, using runtime information from metadata can improve the precision and completeness of the call graph. Hence, our call graph construction combines dynamic and static analyses.

3.4 Test Classes Refinement

The final step of our approach is to refine the test classes unrelated to the changed method set $\mathcal{M}'$ from the test classes set $\mathcal{T}'$ produced by the second step (3.2), using the call graph CG' generated in the third step (3.3). Algorithm 2 shows the details of this step. The inputs of the algorithm are $\mathcal{T}'$, $\mathcal{M}'$, CG', and D_{new} obtained in the previous step, as well as the classes inheritance map S from the $MetaData$ and the external libraries Lib. The output is the set of test classes $\mathcal{T}^*$ that are affected by the changed methods $\mathcal{M}'$. Firstly, taking all methods in $\mathcal{M}'$ as the sources, recursively search in a reversed manner for the callers on the call graph CG' to get the set AM of the affected methods (Line 2, where m_c *can reach* m_e *in* CG' represents that there exists a path from m_c to m_e in the call graph CG'). Then, use the inheritance map of classes S and AM to compute the affected superclasses and the implemented interfaces set AC (Line 3), which contains library classes. Since the mapping S records the superclass of all classes, we use all classes of the methods in AM as sources. Then, we recursively query S for the superclasses or interfaces of each source class. Besides all the source classes, the recursively inherited superclasses and implemented interfaces of each source class (denoted by Ancestor(S, $m.class$) in Line 3) will also be added to AC.

Check in sequence whether the test classes in $\mathcal{T}'$ are affected by $\mathcal{M}'$ (Line 4). Add all methods in test class t to the worklist $\mathcal{W}$ as search sources (Line 5) and iteratively check whether the method m_{src} belongs to the AM (Line 8). If it belongs to AM, it indicates that m_{src} may call the $\mathcal{M}'$. Then, add test class t to $\mathcal{T}^*$ and end

the loop (Lines 9-10). Traverse all callees in the set $CG'[m_{src}]$ (Line 12). If the callee m_{dst} belongs to an external library (Line 13), and the affected class appears in the class or parameter types of m_{dst} (Line 14), also add t to the $\mathcal{T}^*$, then end the loop (Lines 15-16). If the $m_{dst}.class$ belongs to the dependent class of the current test class t, add m_{dst} to the $\mathcal{W}$ as the source for subsequent searches (Lines 19-20).

If the call graph CG' is complete, we can determine whether the current test class can reach the changed methods directly by *AM* on Line 8, without the need to expand the worklist as done on Line 20. However, due to external library calls, the call graph CG' may lack the edges of external libraries, which could lead to false negatives in reachability analysis. To ensure the safety of **JcgEks**, the algorithm must continuously expand the worklist and perform an over-approximate analysis on the affected classes using the external library check at Line 14. Specifically, in Line 14, the affected class set *AC* is used to handle the scenario where the affected class is either upcasted to an external library class or passed as an argument to an external invocation, subsequently resulting in a callback. As the behavior of the library is invisible, there is a possibility of the external library calling the changed method set $\mathcal{M}'$. We over-approximate that as long as any element in *AC* appears in the class or parameters of the external library calls, the test class t is considered to be related to the changed methods.

During the execution phase, the checksum of each test class's class and method will be updated. If the test classes with checksum changes are selected in the second step but filtered out in the fourth step, they will not be executed. **JcgEks** will automatically update the checksum of these test classes to avoid affecting the next selection phase.

3.5 Safety Checking

We define method-level RTS's safety as follows: *if a test class executes a changed method during runtime, the test class is affected by the change.* If an RTS tool is safe at method-level, it should select all the test classes affected by the changes. To measure the safety of each RTS tool, we use *LR* to define the proportion of missed test classes for each RTS technique on each revision. Given a revision r, $LR(r)$ is defined in Equation 1, where T_s represents the set of the test classes selected by the RTS technique on r, T_o represents the set of test classes affected by the changes in r, and $\#T_o$ represents the set T_o's size. If no test classes are affected in the current revision, *LR* is defined as NULL.

$$LR(r) = \begin{cases} \frac{\#(T_s - T_o)}{\#T_o} & T_o \neq \emptyset \\ \text{NULL} & Otherwise \end{cases} \quad (1)$$

The higher the score of *LR*, the more likely this RTS tool may miss the test class, resulting in limited safety. Then, given the set $\mathcal{R}$ of all the revisions, we use the *ALR* in Equation 2 to represent the average proportion of missed test classes, where $\mathcal{MR} = \{r_i \mid r_i \in \mathcal{R} \wedge LR(r_i) \neq \text{NULL}\}$ represents the set of the revisions containing test classes affected by the changes in the revision.

$$ALR = \frac{\sum_{r_i \in \mathcal{MR}} LR(r_i)}{\#\mathcal{MR}} \quad (2)$$

To obtain the precise set of test classes affected by changes, we have implemented a tool called Checker, which utilizes the ASM framework to insert output statements into the changed methods and then runs all test classes. By examining the relationship between test classes and output statements in the test execution log, we can get the test classes affected by the revision changes, *i.e.*, T_o in Equation 1.

Checker's basic procedure consists of the following four steps:

- Similar to **JcgEks**, calculate checksums for each method in the project and compare the checksum differences with the previous version to extract the changed methods;
- Based on the ASM framework, dynamically instrument output statements that report the executing test class;
- Execute all the test classes;
- By analyzing the execution logs, we can identify the test classes executing the instrumented statements, which indicates their dependence on the changed methods.

3.6 Discussion

Although **JcgEks** refines the dynamic FRTS's results by method call information, **JcgEks** does support other types of code changes. If the external library version is changed due to modifying the *pom.xml*, **JcgEks** will degrade to Ekstazi for file-level selection as there are no method changes within the project. Besides the code changes in a method, there may be changes to the fields in the revision. There are two cases for field changes: static or non-static field changes. For **non-static** field changes, the changes alter the hash value of the constructor method (*i.e.*, `<ClassName><init>()V`) of the field's class; for **static** field changes, the changes will alter the hash value of the constant pool construction method of the class (*i.e.*, `<ClassName><clinit>()V`). When there are field changes, **JcgEks** will identify these corresponding methods as changed methods. Hence, **JcgEks** will keep all the test classes that depend on the fields. However, **JcgEks** cannot support the situations where the revision does not change any class files but changes the testing's behavior, *e.g.*, modification to the external non-Java programs executed by some test classes.

To identify the changed methods, like **JcgEks**, Checker records the hash values of each method in the current revision. Checker then identifies the methods in the next revision where the hash values have changed as changed ones. Since Checker and **JcgEks** utilize the same way to locate changed methods, Checker can also support field changes. On the other hand, the soundness and precision of Checker and **JcgEks** depend on correctly identifying the set of changed methods. If a changed method is missed, both **JcgEks** and Checker may not be sound, *i.e.*, missing affected test classes; if a non-changed method is identified as changed, Checker and **JcgEks** may become imprecise, *i.e.*, believing some test classes are affected when they actually are not.

Besides, in principle, Checker aims to establish a mapping between changed methods and their affected test classes. At the implementation level, Checker instruments output statements into the changed method to output the names of the currently executed test class and the changed method, which can be used to establish the mappings. There may exist some other implementation methods, such as the one based on the existing code coverage measurement tools (*e.g.*, JaCoCo [51]). Although the basic idea of using instrumentation to establish mappings is similar, Checker describes the mapping from test classes to change methods more accurately

Table 1: Projects Information

Proj	Name	Head	kLOC	Test Class	Test	Time(s)
P1	number	cc76516	36.3	73.7	19727	41.1
P2	imaging	ca8be30	44.7	153	979	45.4
P3	gerrit	675593d	7.8	24.7	117.6	46.3
P4	configuration	8119b6b	53.4	171	2932.8	47.0
P5	codec	7ca5b56	24.1	63.8	1247.8	48.7
P6	statistics	d82aebb	46.6	73.9	27694.1	52.8
P7	JSqlParser	8378ea4	87.4	125.1	1747.7	57.1
P8	lang	552fead	90.0	187	5360.6	58.8
P9	math	a1ac185	148.0	253	2865.6	69.6
P10	net	f019bab	25.7	54.4	432.8	78.7
P11	io	920a132	52.3	201	2534.1	100.7
P12	accumulo	92331ea	441.0	159.7	555.1	102.9
P13	dbcp	c914196	32.2	49	1494.4	106.2
P14	HikariCP	4b796b5	11.7	33.6	129.2	125.0
P15	rng	c109239	47.5	130.1	3747.7	169.5
P16	flink-ml	bc24667	57.7	103	581.3	277.8
P17	email-ext	4740859	46.1	38.2	325	298.7
P18	openmrs	24e4c54	264.6	314	4465.7	317.5
P19	rocketmq	2d66e95	107.3	216.6	764.5	465.1
P20	celeborn	6b64b1d	190.0	28	116.4	735.7
Avg.			90.7	122.7	3890.9	162.2

than code coverage measurement-based approaches, require less instrumentation, and run more efficiently.

4 EXPERIMENT AND EVALUATION

We evaluate the effectiveness of **JcgEks** and answer the following research questions:

- **RQ1:** How much can **JcgEks** reduce *tests* compared to other tools?
- **RQ2:** How much can **JcgEks** reduce *end-to-end time* compared to other tools?
- **RQ3:** How does **JcgEks** compare to other tools in terms of *safety*?
- **RQ4:** How does **JcgEks** compare against other tools when using state-of-the-art RTS checking tools?

4.1 Experimental Setup

Benchmarks. In order to enhance the capability of Ekstazi and **JcgEks** with a broader range of projects, we upgrade both Ekstazi and **JcgEks** to support both JUnit4 and JUnit5 test cases. Table 1 lists 20 projects employed in our experiment. Columns "Proj" and "Name" present the IDs and names of the projects, which are sorted in the ascending order of execution time. All the selected projects are open-source projects on GitHub, and many have performed well in prior RTS research. Similar to the existing work [32], we select 50 consecutive revisions with Java file changes for each project, and column "Head" presents the first revision of the 50 revisions. Column "kLOC" presents the number of thousands of lines of code for each subject. Columns "Test Class", "Test", and "Time" respectively show the average number of test classes, the average number of tests, and the end-to-end testing time to run all the test classes.

Experiment Design. To evaluate **JcgEks**'s effectiveness, we compare it with four baselines: Ekstazi (dynamic FRTS) [17], FineEkstazi (hybrid RTS) [32], STARTS (static FRTS) [26], and HyRTS (hybrid RTS) [57]. To inspect the regression testing time and the safety of RTS, we also run RetestAll and Checker, which run all the test classes without and with Checker's safety instrumentations, respectively. To investigate the impact of external library calls on safety, we conduct a study on **JcgEks** without analyzing external library calls, denoted by **JcgEks_NE**. In total, we run eight tasks

Table 2: Selected test classes

Proj	Ekstazi(%)	**JcgEks**(%)	**JcgEks_NE**(%)	FineEkstazi(%)	HyRTS(%)	STARTS(%)
P1	7.06	4.48	3.8	1.76	N/A	100
P2	5.82	2.61	2.61	2.61	N/A	27.52
P3	19.43	12.55	10.93	10.12	11.74	65.18
P4	5.03	3.68	3.68	0.99	N/A	100
P5	8.15	5.8	5.8	5.33	N/A	12.7
P6	9.07	7.85	7.85	7.44	N/A	100
P7	68.51	68.43	65.87	59.79	N/A	77.46
P8	8.24	1.39	1.34	1.66	N/A	33.96
P9	14.39	11.23	9.8	12.21	N/A	100
P10	1.84	1.29	1.1	56.25	N/A	100
P11	6.67	3.28	3.18	2.89	N/A	13.78
P12	91.36	91.23	91.23	52.29	N/A	N/A
P13	18.16	12.04	11.22	9.39	N/A	36.94
P14	60.71	30.06	27.08	26.49	N/A	N/A
P15	18.75	12.14	12.07	3.69	N/A	25.6
P16	24.47	17.18	16.7	16.99	N/A	33.59
P17	59.42	13.09	12.57	56.02	N/A	N/A
P18	19.9	15.06	13.18	7.58	N/A	61.59
P19	20.91	13.2	13.11	3.65	100	44.23
P20	39.29	33.57	32.86	N/A	100	55
Avg.	25.36	18.01	17.3	17.74	70.58	58.09

Table 3: End-to-end testing time

Proj	Ekstazi(%)	**JcgEks**(%)	**JcgEks_NE**(%)	FineEkstazi(%)	HyRTS(%)	STARTS(%)
P1	37.71	30.41	29.68	35.04	N/A	91.48
P2	35.9	22.25	23.79	20.7	N/A	51.54
P3	37.8	28.51	26.57	23.76	55.51	152.92
P4	29.79	21.28	21.06	23.19	N/A	87.87
P5	20.33	17.66	18.89	17.66	N/A	20.33
P6	23.48	21.4	21.21	22.73	N/A	87.69
P7	90.54	91.59	90.02	84.41	N/A	98.42
P8	32.31	20.41	19.39	16.5	N/A	46.09
P9	37.5	31.03	28.02	32.9	N/A	85.92
P10	9.02	6.99	6.99	87.29	N/A	94.41
P11	15.89	10.82	10.82	11.12	N/A	27.21
P12	86.98	84.55	80.47	46.65	N/A	N/A
P13	48.96	37.76	36.63	36.06	N/A	94.35
P14	74.64	38.96	36.4	34.56	N/A	N/A
P15	41	25.25	24.13	17.46	N/A	36.76
P16	58.21	39.42	34.49	46.11	N/A	35.42
P17	117.34	30.36	26.65	107.1	N/A	N/A
P18	69.17	39.4	31.84	33.92	N/A	60.5
P19	43.35	22.21	21.41	12.47	668.89	55.54
P20	68.45	55.59	55.77	N/A	86.33	74.41
Avg.	48.92	33.79	32.21	37.35	270.24	70.64

of RetestAll, Checker, Ekstazi, **JcgEks**, **JcgEks_NE**, FineEkstazi, STARTS, and HyRTS on each of the 50 revisions in the 20 projects.

All experiments are conducted on an 80-core server with Intel(R) Xeon(R) Platinum 8269CY CPU @ 2.50GHz CPU and 194GB of memory, running Ubuntu Linux 20.04. For the tools HyRTS and STARTS, we execute all projects on the Oracle Java 64-Bit Server version 1.8.0_11, while for other tools, we run all projects on the Oracle Java 64-Bit Server version 11.0.14. Besides, due to the implementation problem, FineEkstazi did not select any test classes to run on program P20, so the relevant data is missing on this project. In addition, because HyRTS supports only JUnit4 and JDK8, while STARTS only supports JDK8, they cannot support all the projects in the experiment.

4.2 RQ1: Reduction of Test Selection

Table 2 shows the average proportion of test classes selected (%) for each project by the six tools. Projects not supported by FineEkstazi, HyRTS, and STARTS are marked with "N/A". It can be seen that among all the projects, the comparison of the test class ratios selected by these six tools is: HyRTS > STARTS > Ekstazi > FineEkstazi > **JcgEks** > **JcgEks_NE**. On average, Ekstazi, **JcgEks**, **JcgEks_NE**, FineEkstazi, HyRTS, and STARTS selected 25.4%, 18.0%, 17.3%, 17.7%, 70.6%, and 58.1% of all test classes, respectively. Compared with Ekstazi, **JcgEks_NE**, FineEkstazi, HyRTS, and STARTS,

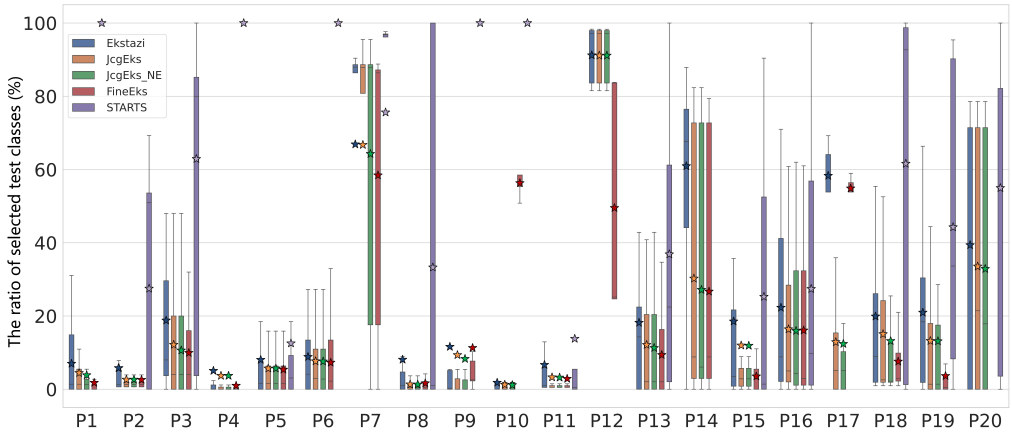


Figure 5: The ratio of selected test classes

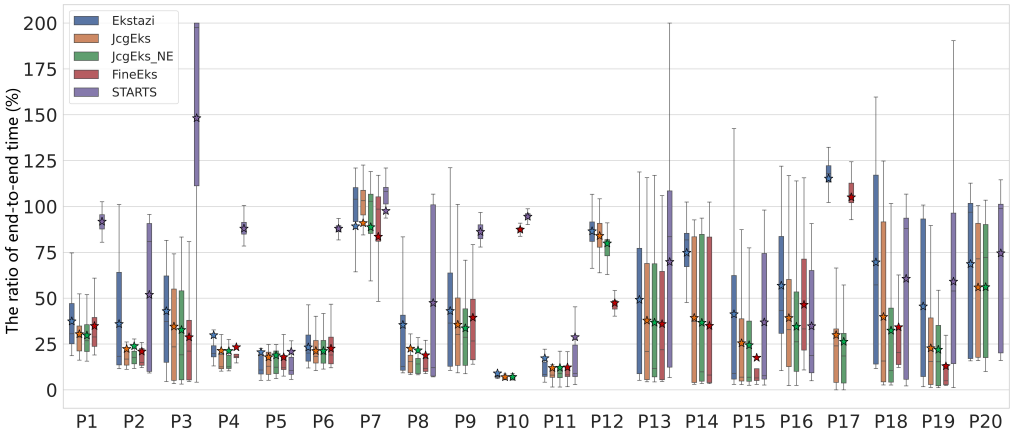


Figure 6: The ratio of end-to-end time

JcgEks can reduce the selection of test classes by 29.0%, −4.1%, 3.1%, 73.1%, and 78.2% on the projects that *both tools can support* (rather than the average of all projects), respectively. The above data indicates that our tool **JcgEks** is superior to the state-of-the-art RTS tool FineEkstazi in reducing the number of test classes.

Compared to FineEkstazi, which also uses call graph-based filtering, **JcgEks** filtered out 3.1% more test classes. This is because FineEkstazi does not finely handle reflections and external library calls, and once dynamic dependences are not a subset of static dependences, method call graph filtering cannot be used. On the contrary, **JcgEks** handles external library calls more safely, improving selection precision further.

Figure 5 illustrates a boxplot of the proportion of the test classes selected from these five tools among 20 projects to the RetestAll. Due to the limited available data, we do not display data about HyRTS in the boxplot but instead list the relevant data in subsequent tables. The x-axis displays all projects, while the y-axis shows the proportion of test classes each tool selects in each project. The median value is marked as a line, while the average value is marked as a pentagram. The same representations apply to Figure 6 below.

As indicated by Figure 5, compared with the percentage of selected test classes, the static FRTS tool STARTS performs the worst. In contrast, the dynamic RTS tools (Ekstazi, **JcgEks**, **JcgEks_NE**, and FineEkstazi) have higher selection precision and can filter out more test classes. Among the four dynamic RTS tools, Ekstazi has only file-level analysis. The other three tools have added further analysis and filtered more test classes, indicating Ekstazi's lower selection precision than the other three tools. The proportions of test classes selected by **JcgEks**, **JcgEks_NE**, and FineEkstazi are not significantly different, indicating that the selection precision of the hybrid RTS has reached a bottleneck.

4.3 RQ2: Reduction of End-to-End Times

An important metric for evaluating RTS techniques is the end-to-end time, which includes not only the test execution time but also the overhead brought by extra analysis. Table 3 illustrates each project's average end-to-end time proportion (%) by the six tools. As indicated by the table, the comparison of the end-to-end time for these 6 tools is: HyRTS> STARTS> Ekstazi> FineEkstazi> **JcgEks**> **JcgEks_NE**. On average, Ekstazi, **JcgEks**, **JcgEks_NE**, FineEkstazi, STARTS and HyRTS need 48.9%, 33.8%, 32.2%, 37.4%, 70.6%, and 270.2% of the RetestAll time, respectively. Without dealing with external libraries, **JcgEks_NE** can reduce end-to-end time by 4.7% compared with **JcgEks**, which indicates that the end-to-end time can be reduced more if we do not consider the dependences of external libraries. However, the RTS's safety may have problems, which will be studied in the next subsection.

Table 4: Results of missed test classes

Proj	**JcgEks_NE**		FineEkstazi		HyRTS		STARTS	
	#	ALR(%)	#	ALR(%)	#	ALR(%)	#	ALR(%)
P1	2	3.09	85	43.67	N/A	N/A	0	0
P2	0	0	1	3.57	N/A	N/A	0	0
P3	0	0	4	6.09	1	3.85	0	0
P4	0	0	3	1.21	N/A	N/A	0	0
P5	0	0	1	0.31	N/A	N/A	0	0
P6	0	0	40	17.97	N/A	N/A	0	0
P7	1	0.64	33	1.99	N/A	N/A	0	0
P8	0	0	0	0	N/A	N/A	0	0
P9	0	0	76	19.49	N/A	N/A	0	0
P10	0	0	1	4.76	N/A	N/A	0	0
P11	0	0	4	7.24	N/A	N/A	0	0
P12	0	0	0	0	N/A	N/A	N/A	N/A
P13	21	10.02	35	10.72	N/A	N/A	8	2.9
P14	0	0	3	0.99	N/A	N/A	N/A	N/A
P15	0	0	221	41.74	N/A	N/A	0	0
P16	8	4.18	73	23.59	N/A	N/A	1	1.52
P17	0	0	0	0	N/A	N/A	N/A	N/A
P18	41	4.38	275	52.6	N/A	N/A	14	4.6
P19	0	0	383	68.71	0	0	0	0
P20	0	0	N/A	N/A	1	0.71	1	0.57
Avg.	3.65	1.12	65.16	16.03	0.67	1.52	1.41	0.56

Figure 6 illustrates the percentage of end-to-end time to RetestAll for all projects executed by each tool. From the overall boxplots, STARTS still has the longest end-to-end time because it selects more test classes and spends more extra time on static analysis. As shown in P3, the average end-to-end time of STARTS is almost 1.5 times that of RetestAll, mainly due to the short testing time of P3 and the overhead resulting from static analysis. On P2 and P3, **JcgEks**, which also used static analysis, has a much lower end-to-end time than STARTS, indicating the efficiency of **JcgEks** due to the incremental building of the static call graph. The end-to-end time of the three hybrid RTS methods is still lower than the FRTS tool Ekstazi, and there is almost no significant difference between these three tools, indicating that **JcgEks** is consistent with state-of-the-art tools in reducing the end-to-end time of regression testing. It should be noted that compared to Ekstazi, **JcgEks** does not reduce end-to-end time in each revision. On some revisions, **JcgEks** may take longer because the extra analyses of **JcgEks** (*e.g.*, call graph building and instrumentations) do not reduce the test classes but introduce overhead.

Furthermore, we found that **JcgEks** can reduce more end-to-end time compared to Ekstazi on projects with longer testing times. The reason is that the proportion of static analysis time is smaller, and improving analysis accuracy from file-level to method-level brings higher benefits. In addition, we calculated the average time spent on static analysis in different revisions of each project, with static analysis time not exceeding 5% of end-to-end time. Moreover, the static analysis time does not increase exponentially on larger-scale projects. This can be attributed to our incremental construction of static call graphs, significantly reducing the time required for static analysis.

4.4 RQ3: Safety Comparison

Table 4 shows the results of missed test classes. Checker calculates the results. Because Ekstazi and **JcgEks** do not miss any test classes in all projects, they are not listed in the table, proving **JcgEks**'s safety in our experiment. The table's symbol "#" represents the total number of missed test classes on the project. It will accumulate if the same test class is missed in different revisions. The *ALR* in the head of the table corresponds to the *ALR* metric

```
public class GerritConnection
    extends Thread implements Connector{
  @Override
  void run(){
-     if (linecount > 0) {
+     if (readCount == 0 || linecount > 0) {
       sleep(SSH_RX_SLEEP_MILLIS);
```

(a) The changed method

```
public class StreamWatchdogTest {
  public void testFullTimeoutFlow(){
    GerritConnection c = new GerritConnection();
    Thread cThread = new Thread(c);
    cThread.start();
```

(b) A test class affected by the changed method

Figure 7: External library call in a revision

defined in Equation 2, representing a project's average missing selection rate. For example, in the P1 project, Ekstazi and **JcgEks** did not miss any test classes, **JcgEks_NE** missed 2 test classes, and FineEkstazi missed 85.

JcgEks_NE, FineEkstazi, STARTS, and HyRTS have varying degrees of missed test classes. The *ALR* score of FineEkstazi is significantly higher than other tools, while the *ALR* score of **JcgEks_NE**, STARTS, and HyRTS is similar. We have the following results by manually analyzing the missed test classes of different tools.

- **JcgEks_NE** missed test classes due to external library calls. As indicated by the table, the occurrence of missed test classes is relatively dense. Checking external library calls can improve the safety of RTS techniques.
- There are two main reasons why STARTS missed test classes in 4 projects. One reason is that static methods cannot handle reflections, and the other is that external library callbacks are not handled. In projects P13, P16, and P18, although both STARTS and **JcgEks_NE** miss test classes, due to the coarser analysis granularity of STARTS (static file-level analysis), some test classes missed by **JcgEks_NE** were selected through STARTS's coarse-grained analysis, resulting in a lower number of missed test classes.
- HyRTS missed test classes in 2 projects because HyRTS instruments and collects dependences for each method, and the missed test classes are affected by the changed methods implemented in Scala code, on which the instrumentation failed.
- The *ALR* score of FineEkstazi is much higher than other RTS tools, mainly due to 3 reasons. The first one is the implementation issue. FineEkstazi divides code changes into 13 types, and some test classes affected by changes are filtered based on change semantics. Our tool Checker found that when code changes contain multiple types of changes, FineEkstazi may miss the test classes due to imprecise semantic perception. For example, in one revision of P1, both the constructor and the "throw" statement were changed. Based on FineEkstazi's classification, changing only the "throw" statement will not select any test class, while changing the constructor will select the affected test class. However, FineEkstazi did not keep the test classes affected by the changed constructor. The second reason is that dynamic binding issues were incorrectly handled in FineEkstazi. The last one is that external library calls were not handled.

Figure 7 shows a case caused by external library callbacks from the gerrit-events project (P3) revision ef2078c. As shown in Figure

7(a), the code change in the revision only includes one line for the `run()` method in the class `GerritConnection`. It should be noted that the changed class is a subclass of the external library class `Thread`, while the changed method is to override the method of the superclass. In the test class `StreamWatchdogTest` as shown in Figure 7(b), upcasting the `GerritConnection` object and using it as a parameter to initialize the `Thead` object. In line 5, the `start()` method of the `Thread` object is called, and it will call back the `run()` method of the class `GerritConnection` in the Java native method `starts0()`. Even worse, native methods do not contain Java bytecode, so existing RTS technologies cannot statically analyze method-level dependences. **JcgEks** over-approximates to handle external library calls safely.

4.5 RQ4: Evaluation by RTSCheck

To further evaluate **JcgEks**, we used RTSCheck [60], a framework for testing RTS tools, to test the six tools: Ekstazi, **JcgEks**, **JcgEks_NE**, FineEkstazi, HyRTS, and STARTS. RTSCheck consists of 3 components: AutoEP, DefectsEP, and EvoEP. Since the four RTS tools being tested operate in a JDK11 environment, but DefectsEP is designed for JDK8 projects, we cannot use the DefectsEP component. Additionally, the experimental design of EvoEP is similar to that of Checker and achieves the same safety checking effects, so we do not use the EvoEP component either. AutoEP has continuously developed programs through automatic code generation, test generation, and code evolution. We only used AutoEP in this experiment and adapted it to the JDK version required by each tool.

Unlike Checker, RTSCheck uses seven rules to classify the quality of RTS tools into three violation categories: R1 and R2 generate safety violations; R3, R4, and R5 generate precision violations; R6 and R7 generate generality violations. Only violating one of R1, R5, R6, or R7 can prove the existence of a bug in the RTS tool. Violating the remaining rules does not indicate the RTS tool has a bug. Table 5 shows the number of RTSCheck violations found in Ekstazi, **JcgEks**, **JcgEks_NE**, FineEkstazi, HyRTS, and STARTS. The results show that these six tools only violated R2 and R3 rules. A violation of R2 means that the tool selects zero test classes, but all the other tools select all the test classes, indicating a possible safety violation on the revision. A violation of R3 means that the tool selects all the test classes on the revision.

As indicated by R2's results, all the RTS tools except FineEkstazi do not have any violations. FineEkstazi does not select any test classes on two revisions. We further inspected the reason for the violations. These two revisions contain method overloading, *i.e.*, adding a new method with the same name as an existing method but different parameter types. For FineEkstazi, adding new methods does not change the method call relations, so FineEkstazi does not select any test classes. However, the program calls the newly added method in the new version. For example, suppose a test class calls `Class1`'s method `foo(long x)` by `c.foo(2)`. When a new method `foo(int x)` is added to `Class1` in the revision, method `foo(int x)` will be called instead of `foo(long x)` with respect to Java's specification of method overloading. These violations indicate FineEkstazi's unsafety in handling this situation. **JcgEks** and **JcgEks_NE** can correctly update the method call graph and select the affected test classes. As shown by R3's results, all the hybrid RTS tools (**JcgEks**, **JcgEks_NE**, FineEkstazi, and HyRTS) select fewer test classes than file-level RTS tools (Ekstazi and STARTS), indicating the better precision of the hybrid RTS techniques. Since the program generated by AutoEP does not involve external libraries, **JcgEks** and **JcgEks_NE** have the same number of R3 violations.

Table 5: Number of violations detected by RTSCheck

Rule	Ekstazi	**JcgEks**	**JcgEks_NE**	FineEkstazi	HyRTS	STARTS
R1	0	0	0	0	0	0
R2	0	0	0	2	0	0
R3	2370	331	331	249	253	2378
R4	0	0	0	0	0	0
R5	0	0	0	0	0	0
R6	0	0	0	0	0	0
R7	0	0	0	0	0	0

Regarding safety checking, there are two differences between RTSCheck and Checker. First, RTSCheck does not provide detection rules for external library callbacks, so it cannot detect such safety violations. Second, Checker can provide the *Oracle* set of affected test classes. RTSCheck's R2 is a voting-based approach that performs differential testing by running multiple RTS tools to test the difference in the numbers of selected test classes. RTSCheck's R2 results can indicate a possible safety violation but cannot directly prove the existence of a safety violation due to the lack of Oracle. Here, the safety checking results of FineEkstazi indicated by RTSCheck's R2 results are consistent with those of Checker, *i.e.*, both RTSCheck and Checker indicate FineEkstazi is unsafe. Besides, the results of **JcgEks_NE**, HyRTS and STARTS are different from those of Checker because RTSCheck does not consider the scenario of external library callbacks.

5 DISCUSSION

Other factors affecting RTS's safety. In this paper, we propose an approach to ensure RTS's safety in the case of external library callbacks and support the analysis of reflection and dynamic binding. However, it is important to acknowledge that various factors can impact the safety of RTS techniques. These factors include the changes in file artifacts [61], the changes to environment variables, the alterations in database contents [54], and more. Currently, mainstream RTS tools do not fully analyze these dependences. Therefore, there is a need for future exploration to address these challenges and improve the overall safety analysis of RTS tools. By considering these additional factors and incorporating them into the analysis, we can enhance the effectiveness and comprehensiveness of measuring RTS's safety.

Trade-off between efficiency and safety. Our study shows that a trade-off between efficiency and safety should be achieved when using RTS techniques. We found that the safe **JcgEks** approach for handling external library calls is less efficient than the unsafe **JcgEks_NE** approach. However, it should be noted that although **JcgEks_NE** is unsafe, it does not result in missing test cases on some projects, which may be due to structural differences between different projects. In some specific scenarios, sacrificing safety for efficiency may be acceptable. For instance, Memon *et al.* [35] propose an unsafe RTS approach to avoid the huge cost of precisely calculating dependences in testing, based on code changes frequency rather than relying on the dependences between changed codes and tests. Machalica *et al.* [33] use a large dataset of historical test results to train a model that precisely predicts test failures.

Mehta *et al.* [34] propose a lightweight statistical model for test selection that reports over 99% of errors. Zhang *et al.* [56] combines ML-based RTS with analysis-based RTS, thereby improving the precision and efficiency of RTS. These examples show that balancing the safety and efficiency of RTS technology for different scenarios will be interesting work in the future.

Finer grained combination. During our experiment, we calculated the time spent on the static analysis part of **JcgEks** and found that it accounted for an average of 5% of the end-to-end time. This suggests that fine-grained static analysis could potentially support large-scale system analysis. Another tool, RETEST [22], combined dynamic class-level instrumentation and static control-flow level analysis, which also shows promise for fine-grained static analysis. HyRTS [32], based on dynamic FRTS, not only studied the integration of method-level analysis but also further investigated the integration of more fine-grained dynamic analysis. The conclusion is that the basic block-level dynamic analysis is not cost-effective. However, this conclusion was based on dynamic RTS, and there is no experimental evidence of integrating fine-grained static analysis with dynamic FRTS. Therefore, it is also interesting to investigate hybrid fine-grained static analysis with dynamic FRTS.

6 RELATED WORK

Hybrid RTS. Hybrid RTS techniques can be categorized into two main groups: dynamic and static mixing and different granularity mixing. TestTube [10] combines static and dynamic analysis to test C-implemented software systems selectively. Bible *et al.* [6] conducted empirical research comparing coarse-grained TestTube and fine-grained DejaVu [42] and found that the hybrid dynamic RTS combining the two approaches can outperform TestTube in terms of precision equivalent to DejaVu. RETEST [22] utilizes dynamic and static analysis while collecting dependences at different granularities to handle Java's object-oriented characteristics. Dejavoo [36] employs a two-stage static RTS approach, conducting preliminary analysis at the class-level and refining it at the control-flow level to enhance analysis capabilities for large systems. DeFlaker [4] uses a mixture of static class-level and dynamic statement-level analysis to monitor code changes' coverage. HyRTS [57] is the first hybrid dynamic RTS technique, combining method-level and file-level analysis to outperform state-of-the-art dynamic FRTS techniques. However, analyzing at a more fine-grained basic block level is not cost-effective. Shi *et al.* [44] utilized dynamic analysis to track reflections and studied the impact of reflection on FRTS to enhance static FRTS's safety. MEST [30] also addressed the reflection problem in RTS by combining dynamic analysis with static MRTS to ensure the safety of static analysis. In the latest research, FineEkstazi [32] claims that RTS approaches have reached the "performance wall". Hence, it classifies the semantics of code changes to identify semantic changes that do not require re-running test cases, essentially combining static analysis and dynamic FRTS.

Compared to the related work, our tool **JcgEks** falls under the category of hybrid RTS. However, it differs from HyRTS [57] of multi-granularity mixing in dynamic RTS. We combine both dynamic and static analysis, as well as different levels of granularity. In contrast to MEST [30] emphasizing dynamic analysis to enhance the safety of static MRTS, our approach focuses on leveraging static analysis to improve the effectiveness of dynamic FRTS. Additionally, while FineEkstazi [32] proposed static rules for combining dynamic FRTS with semantic changes, our approach, combined with static method call graphs, safely handles the issue of external library calls and proposes metrics and tools for RTS safety.

Safety of RTS. To ensure the safety of RTS, it is important to select all tests affected by code changes. Reflection in Java is a significant challenge for static analysis. There has been a lot of related work [3, 5, 9, 19, 28, 29, 49], Bodden *et al.* [9] is the first to study the impact of reflection on static analysis security. They highlighted the limitation of Java's static analysis in supporting reflection and proposed a solution by inserting runtime checks for reflective calls. Soetens *et al.* [50] conducted empirical research on RTS tools that implement static binding and dynamic binding, respectively. They found that static binding occasionally fails to select certain test cases, while dynamic binding offers better safety. Shi *et al.* [44] comprehensively addressed static RTS's unsafety caused by reflection. They employed a similar approach as Bodden's work [9] to track reflection calls using dynamic analysis during test execution. MEST [30] also recognized the problem of imprecise reflection and dynamic binding in static MRTS and improved its safety through dynamic analysis.

Compared with the above work, our work goes beyond by not only supporting dynamic collection of reflection and dynamic binding dependences, building upon the research of Shi *et al.* [44] and MEST, but also examining the influence of external library calls on RTS safety. To our knowledge, no prior research has specifically addressed handling external libraries in static MRTS scenarios. We have thoroughly analyzed and discussed the occurrence of this issue and proposed a solution to address it.

7 CONCLUSION

This paper proposes a new hybrid RTS method **JcgEks**. It integrates dynamic FRTS and static MRTS, combining dynamic and static analysis advantages at different granularities. Based on this, we discuss the unsafety of static RTS analysis caused by external library calls and propose a solution. Furthermore, to measure the safety of the RTS tool, we propose a metric and design an approach to statistically count the number of missed test classes at the method-level granularity. Experimental results demonstrate that **JcgEks** surpasses other state-of-the-art RTS tools in precision, end-to-end time, and safety. We believe this work can serve as the first step in research on external library calls, and the RTS safety checking tool we provide promotes the development of RTS safety research.

Data Availability. Our artifact is available at the following URL: https://github.com/zbchen/JcgEks.

ACKNOWLEDGMENTS

This research was supported by the National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Programs (No. 62172429, 62032024, and 62102442). We want to thank the anonymous reviewers for their valuable comments and suggestions, especially Shepherd, whose suggestions greatly improved the paper's quality.

REFERENCES

[1] Luciano Baresi and Mauro Pezze. 2004. An Introduction to Software Testing. In *Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques, FoVMT 2004, Dagstuhl, Germany, May 3-7, 2004 (Electronic Notes in Theoretical Computer Science)*, Reiko Heckel (Ed.), Vol. 148. Elsevier, 89–111. https://doi.org/10.1016/J.ENTCS.2005.12.014

[2] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[3] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. 2015. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 669–679. https://doi.org/10.1109/ASE.2015.69

[4] Jonathan Bell, Owolabi Legunsen, Michael C Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), 433–444.

[5] John Bible, Gregg Rothermel, and David S. Rosenblum. 2001. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.* 10, 2 (2001), 149–183. https://doi.org/10.1145/367008.367015

[6] Jon M. Bible, Gregg Rothermel, and David S. Rosenblum. 2001. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.* 10 (2001), 149–183.

[7] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. 2011. Regression Test Selection Techniques: A Survey. *Informatica (Slovenia)* 35 (2011), 289–321.

[8] Vincent Blondeau, Anne Etien, Nicolas Anquetil, Sylvain Cresson, Pascal Croisy, and Stéphane Ducasse. 2016. Test case selection in industry: an analysis of issues related to static approaches. *Software Quality Journal* 25 (2016), 1203 – 1237.

[9] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. *2011 33rd International Conference on Software Engineering (ICSE)* (2011), 241–250. https://api.semanticscholar.org/CorpusID:15200755

[10] Yih-Farn Robin Chen, David S. Rosenblum, and Kiem-Phong Vo. 1994. TESTTUBE: a system for selective regression testing. *Proceedings of 16th International Conference on Software Engineering* (1994), 211–220.

[11] Edward Dunn Ekelund and Emelie Engström. 2015. Efficient regression testing based on test history: An industrial evaluation. *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2015), 449–457.

[12] Sebastian G. Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014).

[13] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A systematic review on regression test selection techniques. *Inf. Softw. Technol.* 52 (2010), 14–30.

[14] Emelie Engström, Mats Skoglund, and Per Runeson. 2008. Empirical evaluations of regression test selection techniques: a systematic review. In *International Symposium on Empirical Software Engineering and Measurement*.

[15] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's Distributed and Caching Build Service. *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)* (2016), 11–20.

[16] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 713–716.

[17] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (2015).

[18] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam A. Porter, and Gregg Rothermel. 2001. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.* 10 (2001), 184–208.

[19] Neville Grech, George Kastrinis, and Yannis Smaragdakis. 2018. Efficient Reflection String Analysis via Graph Coloring. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands (LIPIcs)*, Vol. 109. 26:1–26:25. https://doi.org/10.4230/LIPICS.ECOOP.2018.26

[20] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem. *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)* (2018), 112–122.

[21] Mark Harman and Peter W. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2018), 1–23.

[22] Mary Jean Harrold, James A. Jones, Tong Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, Steven Alexander Spoon, and Ashish M. Gujarathi. 2001. Regression test selection for Java software. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*.

[23] Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Soder, Agneta Nilsson, and Magnus Castell. 2015. Supporting Continuous Integration by Code-Churn Based Test Selection. *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering* (2015), 19–25.

[24] David Chenho Kung, Jerry Zeyu Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. 1995. Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs. *J. Object Oriented Program.* 8 (1995), 51–65.

[25] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016).

[26] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), 949–954.

[27] Hareton K. N. Leung and Lee J. White. 1989. Insights into regression testing (software testing). *Proceedings. Conference on Software Maintenance - 1989* (1989), 60–69.

[28] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: taming reflection to support whole-program analysis of Android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. ACM, 318–329. https://doi.org/10.1145/2931037.2931044

[29] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. 2016. Reflection-aware static analysis of Android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. ACM, 756–761. https://doi.org/10.1145/2970276.2970277

[30] Yingling Li, Junjie Wang, Yun Yang, and Qing Wang. 2019. Method-Level Test Selection for Continuous Integration with Static Dependencies and Dynamic Execution Rules. *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)* (2019), 350–361.

[31] Yingling Li, Junjie Wang, Yun Yang, and Qing Wang. 2020. An extensive study of class-level and method-level test case selection for continuous integration. *J. Syst. Softw.* 167 (2020), 110614.

[32] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. More Precise Regression Test Selection via Reasoning about Semantics-Modifying Changes. *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (2023).

[33] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2018. Predictive Test Selection. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2018), 91–100.

[34] Sonu Mehta, Farima Farmahinifarahani, Ranjita Bhagwan, Suraj Guptha, Sina Jafari, Rahul Kumar, Vaibhav Saini, and Anirudh Santhiar. 2021. Data-driven test selection at scale. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021).

[35] Atif M. Memon, Zebao Gao, Bao-Ngoc Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)* (2017), 233–242.

[36] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *SIGSOFT '04/FSE-12*.

[37] Ali Parsai, Quinten David Soetens, Alessandro Murgia, and Serge Demeyer. 2014. Considering Polymorphism in Change-Based Test Suite Reduction. In *XP Workshops*.

[38] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia C. Chesley. 2004. Chianti: a tool for change impact analysis of java programs. *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2004).

[39] Gregg Rothermel and Mary Jean Harrold. [n. d.]. Empirical Studies of a Safe Regression Test Selection Technique. *IEEE Trans. Software Eng.* 24 ([n. d.]).

[40] Gregg Rothermel and Mary Jean Harrold. 1993. A Safe, Efficient Algorithm for Regression Test Selection. In *Proceedings of the Conference on Software Maintenance, ICSM 1993, Montréal, Quebec, Canada, September 1993*, David N. Card (Ed.). IEEE Computer Society, 358–367. https://doi.org/10.1109/ICSM.1993.366926

[41] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *IEEE Trans. Software Eng.* 22, 8 (1996), 529–551. https://doi.org/10.1109/32.536955

[42] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.* 6 (1997), 173–210.

[43] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: an empirical study. *Proceedings IEEE International Conference on Software Maintenance - 1999.* (1999), 179–188.

[44] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-aware static regression test selection. *Proceedings of the ACM on Programming Languages* 3 (2019), 1 – 29.

[45] August Shi, Suresh Thummalapenta, Shuvendu K. Lahiri, Nikolaj S. Bjørner, and Jacek Czerwonka. 2017. Optimizing Test Placement for Module-Level Regression Testing. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), 689–699.

[46] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)* (2019), 228–238.

[47] Min Kyung Shin, Sudipto Ghosh, and Leo R. Vijayasarathy. 2021. An empirical comparison of four Java-based regression test selection techniques. *J. Syst. Softw.* 186 (2021), 111174.

[48] Mats Skoglund and Per Runeson. 2005. A case study of the class firewall regression test selection technique on a large scale distributed software system. *2005 International Symposium on Empirical Software Engineering, 2005.* (2005), 10 pp.–.

[49] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science)*, Vol. 9458. Springer, 485–503. https://doi.org/10.1007/978-3-319-26529-2_26

[50] Quinten David Soetens, Serge Demeyer, Andy Zaidman, and Javier Pérez. 2016. Change-based test selection: an empirical evaluation. *Empirical Software Engineering* 21 (2016), 1990–2032.

[51] JaCoCo Team. [n. d.]. Java Code Coverage Library.

[52] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. 2017. File-level vs. module-level regression test selection for .NET. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017).

[53] David Willmor and Suzanne M. Embury. 2005. A Safe Regression Test Selection Technique for Database-Driven Applications. In *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*. IEEE Computer Society, 421–430. https://doi.org/10.1109/ICSM.2005.15

[54] David Willmor and Suzanne M. Embury. 2005. A safe regression test selection technique for database-driven applications. *21st IEEE International Conference on Software Maintenance* (2005), 421–430.

[55] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verification Reliab.* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430

[56] Jiyang Zhang, Yu Liu, Milos Gligoric, Owolabi Legunsen, and August Shi. 2022. Comparing and Combining Analysis-Based and Learning-Based Regression Test Selection. In *IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2022, Pittsburgh, PA, USA, May 21-22, 2022*. ACM/IEEE, 17–28. https://doi.org/10.1145/3524481.3527230

[57] Lingming Zhang. 2018. Hybrid regression test selection. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 199–209. https://doi.org/10.1145/3180155.3180198

[58] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. *2011 27th IEEE International Conference on Software Maintenance (ICSM)* (2011), 23–32.

[59] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: A large-scale empirical study. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), 60–71.

[60] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 430–441. https://doi.org/10.1109/ICSE.2019.00056

[61] Ahmet Çelik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression test selection across JVM boundaries. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017).