Hybrid Regression Test Selection

Lingming Zhang
The University of Texas at Dallas
lingming.zhang@utdallas.edu

ABSTRACT

Regression testing is crucial but can be extremely costly. Regression Test Selection (RTS) aims to reduce regression testing cost by only selecting and running the tests that may be affected by code changes. To date, various RTS techniques analyzing at different granularities (e.g., at the basic-block, method, and file levels) have been proposed. RTS techniques working on finer granularities may be more precise in selecting tests, while techniques working on coarser granularities may have lower overhead. According to a recent study, RTS at the file level (FRTS) can have less overall testing time compared with a finer grained technique at the method level, and represents state-of-the-art RTS. In this paper, we present the first hybrid RTS approach, HyRTS, that analyzes at multiple granularities to combine the strengths of traditional RTS techniques at different granularities. We implemented the basic HyRTS technique by combining the method and file granularity RTS. The experimental results on 2707 revisions of 32 projects, totalling over 124 Million LoC, demonstrate that HyRTS outperforms state-of-the-art FRTS significantly in terms of selected test ratio and the *offline* testing time. We also studied the impacts of each type of method-level changes, and further designed two new HyRTS variants based on the study results. Our additional experiments show that transforming instance method additions/deletions into file-level changes produces an even more effective HyRTS variant that can significantly outperform FRTS in both *offline* and *online* testing time.

ACM Reference Format:
Lingming Zhang. 2018. Hybrid Regression Test Selection. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180198

1 INTRODUCTION

Regression testing has been widely used to ensure that software evolution does not break existing functionalities. However, simply rerunning entire regression test suites can be extremely time consuming, e.g., some real-world test suites take weeks to run [38]. Besides, regression testing can also consume a lot of computing resources, e.g., Google has over 100 Million tests running each day, occupying various machines and clusters [7, 8, 20, 33]. As a result, shown in a prior survey by Yoo and Harman [47], various approaches have been proposed to reduce the costs of regression testing, including regression test selection [15, 21, 23, 26, 27, 34, 35, 37, 46, 50], regression test-suite reduction [17, 18, 24, 28, 41, 43, 51, 52], regression test-case prioritization [19, 25, 32, 38, 40, 48, 49].

Regression Test Selection (RTS) [13, 15, 21, 23, 26, 34, 35, 37, 46, 50] aims to select and run only the tests that are affected by code changes, since the tests not affected by code changes should have the same results with prior runs. In this way, RTS can greatly save the regression testing efforts, and has been widely used in practice [23, 30, 42]. A typical RTS technique requires two dimensions of information: (1) the test dependency information (i.e., the program elements that can be executed during each test execution) on an old program version, (2) the changed program elements. Then, a *safe* RTS technique selects any test whose dependencies overlap with the changed program elements as the affected tests, since missing any of those tests may fail to detect some regression bugs.

Depending on how the test dependencies are collected, RTS techniques can be categorized as dynamic [23, 26, 34, 35, 37, 50] and static [29, 30, 36] techniques; depending on the granularities of program elements in test dependencies and program changes, RTS techniques can be categorized as basic-block-level [26, 34, 37], method-level [35, 50], file-level [23, 30], and even module-level [42, 44] techniques. Since static RTS uses static analysis to overapproximate the test dependencies and thus may select more tests than necessary, dynamic RTS techniques at different granularities have been largely studied in the literature. According to a recent study [23], RTS at the file granularity can have less end-to-end time (i.e., including both RTS overhead and actual testing time) compared with a finer grained technique at the method level due to its lower overhead, and represents state-of-the-art RTS. Actually, various open-source projects have already adopted file-level RTS in their daily development, e.g., Apache Camel [9], Math [10], and CXF [11].

Although the file-level RTS has been demonstrated to be cost-effective, it may select more tests than finer-grained RTS. For example, prior study showed that file-level RTS may select twice as many tests as method-level RTS on five GitHub Java projects [23]. Actually, dynamic RTS techniques at different granularities have there own strengths – while techniques working on coarser granularities may have lower overhead, RTS techniques working on finer granularities may be more precise in selecting tests. Our insight is to **combine the strengths of RTS at different granularities** to design a hybrid RTS approach that can be more cost-effective than any of the existing RTS techniques.

In this paper, we propose the first hybrid RTS approach that analyzes test dependency and change information at multiple granularities. We then implement a basic hybrid RTS technique, HyRTS, that combines method-level and file-level analysis for more cost-effective RTS for modern Java programs. The basic idea is to perform the method-level analysis just for the class files with only finer-grained method-level changes, while performing file-level analysis for all the other cases, e.g., file-level additions/deletions, or class file

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180198

header changes. To evaluate the proposed technique, we compare it against state-of-the-art file-level RTS (denoted as FRTS) on 2707 revisions of 32 GitHub Java projects, totalling 124,764,602 LoC. The experimental results demonstrate that HyRTS significantly outperforms state-of-the-art FRTS technique in terms of both selected test ratio and the *offline* testing time (when the test dependencies are collected offline) – HyRTS/FRTS selects 18.35%/27.18% tests (i.e., 8.83*pp* more precise) and costs 31.60s/40.06s *offline* testing time (i.e., 21.1% faster) on average across all subjects. We also performed an extensive study on the impacts of each type of method-level changes on HyRTS, and designed two new HyRTS variants based on the study results. Additional experimental results show that: (1) further including the basic-block-level analysis in HyRTS does not pay off; (2) further transforming instance method additions/deletions into file-level changes actually produces an even more cost-effective variant that significantly outperforms FRTS in both the *offline* and *online* modes.

The paper makes the following contributions:

- The design and implementation of the first hybrid RTS approach, HyRTS, that combines method and file level RTS.
- Experimental results demonstrating the effectiveness and efficiency of the HyRTS technique on 2707 revisions of 32 projects, totalling 124,764,602 LoC.
- An extensive study on the impacts of each type of method-level changes on RTS during real-world software evolution.
- Two additional HyRTS variants designed based on the study results, and experimental results demonstrating that further transforming instance method additions/deletions to file-level changes can make RTS even more cost-effective.

2 BACKGROUND AND EXAMPLE

Traditional Regression Test Selection (RTS) techniques [23, 26, 34, 35, 37, 50] usually apply at certain level of code granularity. Pioneer RTS techniques [26, 34, 37] usually apply at the level of program basic blocks. Such techniques collect dynamic test dependencies for old program versions at the basic-block level, and compute detailed program change information by traversing Control-Flow Graphs (CFGs) using Depth-First Search (DFS). Then, the tests whose dependencies overlap with the computed changes are selected for execution. Although precise, such techniques need to compute detailed test dependency and change information, and can incur non-trivial overhead [23, 34].

To reduce the RTS overhead, researchers also proposed RTS at the method level. Such techniques (e.g., FaultTracer [50] and Chianti [35]) compute program changes at the method level (denoted as *atomic changes*). For example, a CM atomic change denotes a change to a method body. Besides normal atomic changes, they also capture changes of instance method overriding hierarchy to detect dynamic dispatch changes, denoted as LC (i.e., *look-up* changes). A LC change is usually formulated as ⟨`X,Y.m()`⟩, which denotes that an invocation to `Y.m()` with `X` as the runtime object may be resolved into a different target method due to software changes. Such LC changes are additionally generated whenever instance methods are added (AM) or deleted (DM). Then, the tests whose method-level dependencies may overlap with the atomic changes are selected. To illustrate, Figure 1 presents an example program

```
1 // source code classes
2 class A {
3   A(){...}
4   int m1(){...}
5   static int m2(){...}
6 }
7 class B extends A {
8   B(){...}
9   static int m2(){...}
10 }
```

```
1 // test classes
2 class T1 {
3   void t(){A a=new A(); a.m1();}}
4 class T2 {
5   void t(){A.m2();}}
6 class T3 {
7   void t(){A b=new B(); b.m1();}}
8 class T4 {
9   void t(){B.m2();}}
```

Figure 1: Example

Table 1: Example test dependencies

Test	Method dependency	File dependency
T1	T1.t(), A.A(), A.m1()	T1, A
T2	T2.t(), A.m2()	T2, A
T3	T3.t(), A.A(), B.B(), A.m1()	T3, A, B
T4	T4.t(), B.m2()	T4, B

together with its tests and Table 1 presents the corresponding test dependencies at both the method and file levels. When `A.m2()` is changed in the next revision (denoted as CM: `A.m2()`), only test `T2` needs to be selected and re-run, since all the other tests cannot execute the change at all. However, when `B.m1()` is added, a naive method-level RTS technique fails to select any test since no test directly executed the added tests in the prior revision. Therefore, a safe method-level technique [35, 50] additionally annotates each instance method dependency element with both runtime and static receiver object types (Note that such additional information can incur extra overhead during the dependency collection). For example, the method-level dependency `A.m1()` for `T3` will be annotated with ⟨`B,A.m1()`⟩ to indicate that the invocation was to `A.m1()` with runtime object type of `B`. Therefore, the method-level RTS will be able to match the annotation with the corresponding LC change to select the truly impacted test `T3`.

Recently, researchers have also proposed Ekstazi [23], an even coarser-grained RTS technique at the binary class file level. Such file-level RTS collects test dependencies and computes program changes both at the file level. Although the coarse granularity makes the technique select more tests than the method-level RTS, operating at the file level offers much lower overhead: (1) collecting test dependencies at the file level can be faster than the method level; (2) computing the program changes at the file level can be directly achieved by computing binary file *checksums*, which can be extremely fast; (3) file-level RTS also does not need to track dynamic dispatch changes [23]. For example, according to the file-level dependency shown in Table 1, when `B.m2()` is added, file-level RTS will directly be able to detect that `T3` (and also `T4` due to the imprecision of file-level RTS) is impacted without collecting expensive runtime type information, since file `B` is already accessed by the test. Although file-level RTS selects more tests than method-level RTS, it has much lower overhead. Overall, the file-level RTS has been shown to save the end-to-end testing time for real-world projects, and significantly outperform the method-level RTS. Similar with recent advances in flaky test detection using hybrid coverage [14], this work aims to further advance RTS using hybrid RTS analysis.

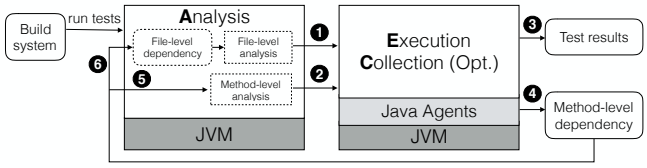

Figure 2: HyRTS design overview

3 TECHNIQUE AND IMPLEMENTATION

In this section, we present our hybrid RTS approach, HyRTS, to combine the strengths of traditional RTS techniques at different granularities. We first introduce a basic HyRTS technique that simply computes detailed method-level changes only when the corresponding files are modified, while simply computing file-level changes for file deletions and additions (Section 3.1). Then, we introduce various HyRTS extensions to further study/improve RTS cost-effectiveness (Section 3.2). Finally, we summarize the basis of HyRTS – change transformation, and discuss its safety (Section 3.3).

3.1 Basic HyRTS

The overall design of our basic HyRTS technique is shown in Figure 2. The technique can be invoked whenever the underlying build system fires the `run test` command. The analysis phase of RTS performs two levels of RTS analysis: (1) file-level analysis takes the file dependencies and selects all the tests that cover file-level changes for the execution phase (❶), (2) method-level analysis takes the method dependencies and selects all the tests that overlap with the detailed method-level changes for actual execution (❷). Then, both sets of selected tests will executed. During the execution phase, the user can choose to also collect the test dependencies for future RTS, or collect the dependencies after the execution phase. The execution phase results will be stored as test results (❸), while the collection phase results will be method-level dependencies (❹). Then for the next software revision, the method-level dependencies will be directly used for the method-level analysis (❺). In addition, file-level dependencies can also be derived from method-level analysis (e.g., a dependency on class `A` can be derived from a dependency on method `A.m()`) for file-level analysis (❻).

Note that the HyRTS tool has been implemented as a Maven plugin for testing Java programs with JUnit tests, and is publicly available on our HyRTS homepage [2]. HyRTS currently supports test-class level RTS for both single-module and multi-module Maven projects, JUnit 3 and JUnit 4 tests, as well as unit (via Maven Surefire [6]) and integration (via Maven Failsafe [5]) tests. We next describe the three key components for HyRTS:

3.1.1 Change Computation. We could have built HyRTS based on the existing method-level RTS tool, FaultTracer [50], or file-level RTS tool, Ekstazi [23]. However, Ekstazi is not open-source, while FaultTracer computes program changes at the source-code level based on the Eclipse Java Development Tools (JDT) [4], which can incur large overhead for large projects [23]. Therefore, we build our hybrid RTS tool from scratch, following the design decisions of state-of-the-art Ekstazi tool [22]. For example, we also compute the *checksums* of bytecode files to efficiently detect file changes. Furthermore, we also use *smart checksums* to compute bytecode file contents without debugging information (e.g., line number information). Different from Ekstazi, we also need to trace method-level changes. To efficiently store method-level information for the prior revision and compute method-level changes, we compute the smart checksum for each method. The detailed HyRTS change computation is shown in Algorithm 1. Shown in Lines 5-9, for the unchanged files, all the method-level checksums are directly copied from prior revision without further detailed analysis. During the actual file-level change computation (Line 11), HyRTS computes 4 types of changes shown in the top half of Table 2. Note that we introduce CH to model the global changes to a file, e.g., interface/super class changes or recompiled bytecode for a newer JDK version, to avoid returning all enclosing methods as changed. In case of AF/DF/CH changes, any test executing the corresponding class files will be directly selected based on file-level RTS analysis; in case of other file changes (CF), the basic HyRTS does not keep file-level changes, and performs method-level detailed change computation (Line 13). HyRTS supports all the method-level changes supported by prior method-level RTS [35, 50], as shown in the bottom part Table 2. Following existing safe method-level RTS [35, 50], we also compute LC changes in case of instance method additions or deletions. Note that field changes do not need to be traced, since all field changes will reflect in the corresponding method-level changes (e.g., initializer changes) at the bytecode level [35]. Also, we split non-initializer method changes into instance and static method changes (e.g., AM to ASM and AIM) to study the impact of each detailed type of changes. In this work, we denote all changes computed by HyRTS as $\Delta = \{\delta_m, \delta_f\}$, where δ_m denotes the method-level changes, such as DIM, and CSI, while δ_f denotes the file-level changes, such as DF and CH.

Algorithm 1: HyRTS change computation

```
   Input  : V1: Old program revision, V2: New program revision
   Output : Δ: The hybrid code changes
   /* Read old checksums, and initialize new ones                    */
1  Map<File, CSUM> oldFileCSUM, Map<File, Map<Meth,CSUM»
     oldMethCSUM=deserializeCSUM(V1)
2  Map<File, CSUM> newFileCSUM, Map<File, Map<Meth,CSUM»
     newMethCSUM=∅
3  for File f in V2 do
       /* Compute new file checksums                                 */
4      newFileCSUM[f]=computeFileCSUM(f)
       /* Compute new method checksums when necessary                */
5      if newFileCSUM[f]==oldFileCSUM[f] then
6          newMethCSUM[f]=oldMethCSUM[f]
7      else
8          for Meth m in f do
9              newMethCSUM[f][m]=computeMethCSUM(m)
   /* Serialize new checksums for next RTS                           */
10 serializeCSUM(newFileCSUM, newMethCSUM)
   /* File-level change computation [23]                             */
11 AF,DF,CH,CF=FileDiff(oldFileCSUM, newFileCSUM)
12 δf =AF∪ DF∪ CH
   /* Meth-level change computation for changed files[35]            */
13 δm=⋃f∈CF MethDiff(oldMethCSUM[f],newMethCSUM[f])
14 return {δf, δm}
```

3.1.2 Dependency Collection. Our test dependency collection component is implemented based on the ASM bytecode manipulation framework [12] with the Java Agent [3] support for JVM load-time code instrumentation. We override `ClassVisitor` and `MethodVisitor` to record the method dependency information together with the runtime and static object types for each instance method invocation for safe method-level RTS. Following recent advances on RTS [23, 30], we focus on *test-class* level test selection since test methods can be hard to isolate in practice, e.g., test methods may be parameterized or depend on other methods within the test class. We also create another Java Agent to dynamically wrap the corresponding runner classes (e.g., `org.junit.runner.Runner` for JUnit 4) to capture the test class start/end events for both JUnit3 and JUnit4 tests in order to trace the per-test dependency information. We also use the same Java Agent to exclude the unselected tests for test execution. In this work, we denote all the test dependencies used in HyRTS as $\mathbb{TD} = \{\mathcal{TD}_m, \mathcal{TD}_f\}$, where $\mathcal{TD}_m$ represents the method-level test dependencies (e.g., methods invoked as well as runtime type information for receiver objects during test execution) while $\mathcal{TD}_f$ is the file-level dependencies (i.e., the set of class files accessed during test execution). Note that HyRTS only collects $\mathcal{TD}_m$ during runtime for sake of efficiency, and $\mathcal{TD}_f$ can be derived offline from $\mathcal{TD}_m$ via skipping detailed method information, e.g., accessing method `A.m()` can be converted into accessing file `A`.

3.1.3 RTS and Application Modes. With the change information (Δ) and test dependencies collected from the prior revision ($\mathbb{TD}$), the selected tests $\mathcal{T}_s$ for the current revision can be computed as the tests whose dependencies on the prior revision have overlap with the changes, i.e., $\mathcal{T}_s = \Delta \Cap \mathbb{TD} = \{\delta_m \cap_m \mathcal{TD}_m\} \cup \{\delta_f \cap_f \mathcal{TD}_f\}$, where $\cap_m$ denotes the method-level test selection rules [35], while $\cap_f$ denotes the file-level test selection rules [23]. As shown in Figure 2, the time costs during the RTS process can be categorized as the Analysis, Execution, and Collection time. In practice, the users usually can choose two different RTS modes – (1) the *offline* mode that collects test dependencies offline (i.e., after running the selected tests), and (2) the *online* mode that collects test dependencies online during the RTS process. Note that the *offline* mode costs more overall CPU time, but the users can get faster test feedback and then prepare the test dependencies for the next RTS run after obtaining the test results, while the *online* mode returns both the test results and test dependencies at the same time [23, 30]. Our HyRTS technique supports both modes – dependency collection will be triggered only during the *online* mode, while test selection will be triggered for both modes. In the experimental study section, we evaluate HyRTS under both modes, i.e., measuring the AE (Analysis and Execution) time for the *offline* mode and the AEC (Analysis, Execution, and Collection) time for the *online* mode.

3.2 HyRTS Extensions

3.2.1 HyRTS Study Variants. Besides the basic HyRTS, we further study the impacts of each type of method-level changes by transforming it into file-level changes to explore the directions for further improving HyRTS. For example, when studying the impact of CIM changes, HyRTS treats the entire file as having a CF change (CF changes are kept and analyzed in the file-level RTS analysis for these variants) whenever there is any CIM change within a file, while still tracing all the other changes in the same way as the basic HyRTS. In this way, HyRTS does not need trace CIM changes anymore, since they are already subsumed by the corresponding file changes. Clearly, such HyRTS variant may select more unnecessary tests and be more imprecise, since any test accessing the CF file will be selected. However, such HyRTS variant can show the impact of each type of fine-grained method-level changes, and provide guidelines for more cost-effective RTS. The detailed experimental results studying the impact of each fine-grained method-level change type can be found in Section 5.2.

Table 2: Supported change types

Name	Description
DF	Delete a class file
AF	Add a class file
CH	Change file head
CF	Change a class file (not kept in basic HyRTS)
DSI	Delete a static initializer
ASI	Add a static initializer
CSI	Change a static initializer
DI	Delete an instance initializer
AI	Add an instance initializer
CI	Change an instance initializer
DSM	Delete a static non-initializer method
ASM	Add a static non-initializer method
CSM	Change a static non-initializer method
DIM	Delete an instance non-initializer method
AIM	Add an instance non-initializer method
CIM	Change an instance non-initializer method

3.2.2 HyRTS$_B$. According to the study results in Section 5.2, CIM and CSM changes tends to have the highest impact on HyRTS effectiveness, i.e., transforming CIM and CSM changes into file-level changes incurs HyRTS to select many more tests, indicating that CIM and CSM changes require even finer-grained analysis instead of coarser-grained analysis. Therefore, we further propose HyRTS$_B$ to extend basic HyRTS to perform finer-grained basic-block level analysis in the case of CIM and CSM changes to investigate the cost-effectiveness of more precise HyRTS (while keeping the basic HyRTS method-level and file-level analyses for other cases). We strictly follow prior work on basic-block-level RTS [26] in implementing the basic-block-level analysis based on Control-Flow Graph (CFG) analysis, and also analyze the try-catch constructs to handle Java Exceptions. Note that HyRTS$_B$ requires to also collect detailed test dependencies at the basic-block level. We implement both the CFG analysis and the basic-block level dependency collection using the ASM framework [12].

3.2.3 HyRTS$_F$. The study results in Section 5.2 also demonstrate that AIM (i.e., instance method addition) and DIM (i.e., instance method deletion) changes do not have high impacts on HyRTS effectiveness, i.e., transforming either AIM or DIM into file-level changes do not incur much test selection imprecision. Therefore, we further propose HyRTS$_F$ to extend basic HyRTS by further transforming both AIM and DIM changes into file-level changes (CF). Although HyRTS$_F$ may select more tests than basic HyRTS, without AIM and DIM changes, HyRTS$_F$ does not need to consider the class inheritance hierarchy changes (i.e., computing LC changes) or trace

the runtime type information for each instance method invocation, since the LC changes and runtime type information are both used to handle safety issues in case of instance method additions or deletions [23, 35, 50]. For example, when instance method `A.m()` is added, test `T` is affected due to method dynamic dispatch change although it does not have `A.m()` in its old dependency. If `T` does not execute other changes, `T` must invoke `m()` with a receiver object `obj` of type `A` or subtype of `A` in the old revision (otherwise only adding `A.m()` cannot impact `T`). No matter `obj` is of type `A` or subtypes of `A`, according to JVM specifications, `A`'s initializer(s) must be invoked first to create `obj`. Therefore, `A` is accessed by `T`, and simply recording file `A` as changed will be sufficient to select `T`. Without tracing LC changes and runtime type information, HyRTS$_F$ may be much faster than the basic HyRTS in end-to-end time.

3.3 Hybrid RTS Safety

Our HyRTS approach transforms code changes into finer/coarser grained changes to implement different RTS variants. Before talking about HyRTS safety, we first define two types of transformations.

Definition 3.1 (Basic Transformation). *When all fine-grained elements under a coarse-grained element are changed, a basic transformation can simply mark the coarse-grained element as changed to lower RTS overheads, i.e.,* $[\forall e_i \in e,\ \delta(e_i)] \Rightarrow \delta(e)$*, where* $\delta(.)$ *denotes the corresponding element is treated as changed.*

To illustrate, if class `B` in Figure 1 is completely deleted in the new revision, traditional method-level techniques still need to dig into `B` to detect all the deleted methods, while file-level RTS can directly return a file deletion change. The actual selection phase for the file-level RTS can also be faster due to the small number of file-level test dependencies to analyze. Therefore, we can easily come up with hybrid RTS techniques that keep changes at the coarse granularity when all the fine-grained elements under a coarse-grained element are changed, while keeping changes at the fine granularity at the other cases. Note that our basic HyRTS and HyRTS$_B$ are example techniques in this category.

Besides the basic change transformations that will not suffer from accuracy lost, this work also investigates transformations that may incur accuracy lost, e.g., HyRTS variants shown in Section 3.2.1 and Section 3.2.3. To illustrate, when only `B.m2()` in Figure 1 is changed, we can still mark the entire class `B` as changed. Although such *aggressive* transformation may select more tests than necessary, the RTS overhead and end-to-end testing time may be further lowered:

Definition 3.2 (Aggressive Transformation). *Aggressive transformation marks a coarse-grained element as changed when only part of its fine-grained children elements get changed, i.e.,* $[\exists e_i \in e,\ \delta(e_i)] \Rightarrow \delta(e)$.

Note that when applying hybrid RTS using either basic or aggressive change transformations, the test dependencies should be traced at the fine granularity (e.g., method level for basic HyRTS, while basic-block level for HyRTS$_B$) since the coarse-grained dependencies can be derived from the fine-grained ones. Then, the hybrid changes at different granularities can be matched against corresponding test dependencies to select tests. Despite the fact that hybrid RTS via change transformation may incur imprecision, we next discuss that hybrid RTS will not incur new safety issues.

Theorem 3.1. *Hybrid RTS via change transformation cannot introduce new safety issues for dynamic RTS.*

Proof. In general, hybrid RTS collects test dependencies at different levels, denoted as $\mathbb{TD}$=$\{\mathcal{TD}_1, \mathcal{TD}_2, ..., \mathcal{TD}_n\}$, where $\mathcal{TD}_i$ ($1 \le i \le n$) denotes the test dependencies at level i. For example, for HyRTS$_B$, $\mathbb{TD}$=$\{\mathcal{TD}_b, \mathcal{TD}_m, \mathcal{TD}_f\}$, which includes basic-block, method, and file level test dependencies. Furthermore, hybrid RTS also transforms all the changes into different levels, denoted as Δ=$\{\delta_1, \delta_2, ..., \delta_n\}$, where δ_i denote the changes at level i. For example, for HyRTS$_B$, Δ=$\{\delta_b, \delta_m, \delta_f\}$, which includes basic-block, method, and file level changes. Then, the selected test set $\mathcal{T}_s$ can be computed as $\mathcal{T}_s = \mathbb{TD} \Cap \Delta = \bigcup_{1 \le i \le n} \{\mathcal{TD}_i \cap_i \delta_i\}$, wherre $\cap_i$ denotes the RTS rules at level i. If hybrid RTS introduces new safety issues, then the test selection must made some unsafe selection at some level, e.g., there should exist level i ($1 \le i \le n$), such that $\{\mathcal{TD}_i \cap_i \delta_i\}$ failed to select some tests that should have been selected by safe RTS.

Then, we can easily construct a revision $\mathcal{V}_2'$ between the original old version $\mathcal{V}_1$ and the original new version $\mathcal{V}_2$, with only changes within δ_i. Then, applying traditional RTS at level i between $\mathcal{V}_1$ and $\mathcal{V}_2'$ will perform $\{\mathcal{TD}_i \cap_i \delta_i\}$, and thus making unsafe selection. However, our hybrid RTS is built on *safe* RTS techniques at different levels, and thus the traditional RTS at level i also cannot make unsafe test selection. Contradiction. □

4 STUDY

4.1 Research Questions

In this study, we are interested in the following research questions:

- **RQ1:** How does our basic hybrid RTS technique (HyRTS) compare with state-of-the-art file-level RTS (FRTS)?
- **RQ2:** How do different types of method-level changes impact the RTS results?
- **RQ3:** Can we further transform method-level changes of HyRTS into finer-grained or coarser-grained changes for even more cost-effective RTS?

Note that we do not compare HyRTS with the method-level RTS that compares the detailed contents of all methods regardless of file-level changes (e.g., FaultTracer [50]), since prior work [23] has shown that method-level RTS is much slower than FRTS, making it sufficient to compare HyRTS against state-of-the-art FRTS.

4.2 Subjects

Table 3 presents all the subject systems for this study. For a fair comparison with state-of-the-art FRTS, we used all the single-module Maven projects from recent studies on RTS [23, 30]. Following prior work [30], for each project, we started from the Head revision, and selected 100 revisions before it (Note that if fewer than 100 revisions are available, we just use all of them). Then, we use all the 2,707 revisions (of the studied 32 projects) that can pass all the regression tests in our evaluation. In the table, all the subjects are sorted in the ascending order of their test execution time. Column 1 presents all the projects used as the subject systems. Note that following prior RTS work [23], we categorize the subjects into two subsets, i.e., short-running subjects (with test execution time below 60s) and long-running subjects (with test execution time above 60s).

Subs	Head	Revs	Tests (Head)		Size (LoC)	
			#	Time (s)	Head	Total
invokebinder	004d2fd	100	107	2	3,036	214,867
compile-testing	e4269a6	73	176	4	6,071	309,745
logstash-logback-encoder	4336fdc	95	208	4	9,435	829,731
commons-cli	b486fbd	96	371	4	6,601	611,755
joda-time	d7d1620	100	4,203	5	85,847	8,557,599
commons-dbutils	633749d	66	300	5	6,763	367,111
commons-validator	e36fc4b	97	527	5	15,635	1,446,859
commons-fileupload	f542f18	94	72	6	4,289	405,347
asterisk-java	61ecf80	59	220	7	43,102	2,538,060
commons-functor	3da1a4b	40	1,079	7	18,174	710,725
la4j	db20416	99	801	8	13,414	1,411,825
commons-jxpath	e48043d	87	411	9	24,910	2,131,230
commons-email	4ad899d	57	138	10	6,756	372,836
commons-compress	d5f3062	100	614	10	34,347	3,358,789
commons-codec	1a4d9cc	79	847	11	19,530	1,523,899
jfreechart	54eeb32	100	2,261	12	140,671	14,071,394
commons-collections	3c1867e	46	16,069	23	61,637	2,811,369
commons-lang	0136218	100	3,946	24	73,781	7,292,816
commons-imaging	0aec9fd	100	441	28	38,020	3,784,160
commons-configuration	4239889	99	2,739	31	67,461	6,647,286
commons-net	2b0f338	100	276	61	27,525	2,726,739
closure-compiler	e5ca4a7	100	11,309	62	297,130	29,637,121
java-apns	a7d1e9f	48	111	73	5,626	253,403
commons-io	593de77	99	1,309	75	29,267	2,887,098
commons-math	79c4719	72	6,008	79	182,030	12,871,938
commons-dbcp	6a65042	100	560	81	20,547	1,995,908
log4j	7be00ee	57	344	95	30,287	1,922,801
stream-lib	a13064c	99	147	105	8,492	819,463
HikariCP	980d8dc	92	109	117	10,283	916,531
OpenTripPlanner	cc4dc2e	87	388	244	78,696	6,847,618
commons-pool	e35320b	96	272	400	13,567	1,264,314
mapdb	ad7102c	70	5,168	867	48,239	3,224,265
Total	–	2,707	61,531	2,472	1,431,169	124,764,602

Table 3: Subject statistics

Columns 2 and 3 present the short SHA-1 hash for the Head revision and the number of revisions for each studied project. Columns 4 and 5 present the test size (i.e., number of test methods) and test execution time for the first executable revision of each studied project. Finally, Columns 6 and 7 present the lines of code (LoC) information (computed by SLOCCount [1], excluding comments and spaces) for the Head revision and all the revisions for each studied project. In total, our experimental study involves 2,707 revisions of 32 projects, totalling over 124 Million LoC, and has a significantly larger scale than prior studies on RTS [23, 30].

4.3 Experimental Setup

For each studied RTS technique, we compute the following widely used RTS metrics to measure its effectiveness:

Selected Test Ratio The ratio of selected tests directly reflects the precision of RTS techniques, and has been widely used in RTS evaluation since the first proposal of RTS [23, 26, 30, 34, 35, 37, 39, 50]. We also use this metric to study the selection precision of different studied RTS techniques.

End-to-End Testing Time Although the ratio of selected tests can tell how precise a RTS technique is, it does not show the overhead incurred by the RTS technique. Actually, a RTS technique that is extremely precise but costs even more than re-executing the entire test suite can be useless in practice. Therefore, recent work on RTS [23, 30] begins to consider the actual time savings of RTS techniques. Following recent RTS work [23, 30], we measure the end-to-end testing time for both modes supported by HyRTS, i.e.,

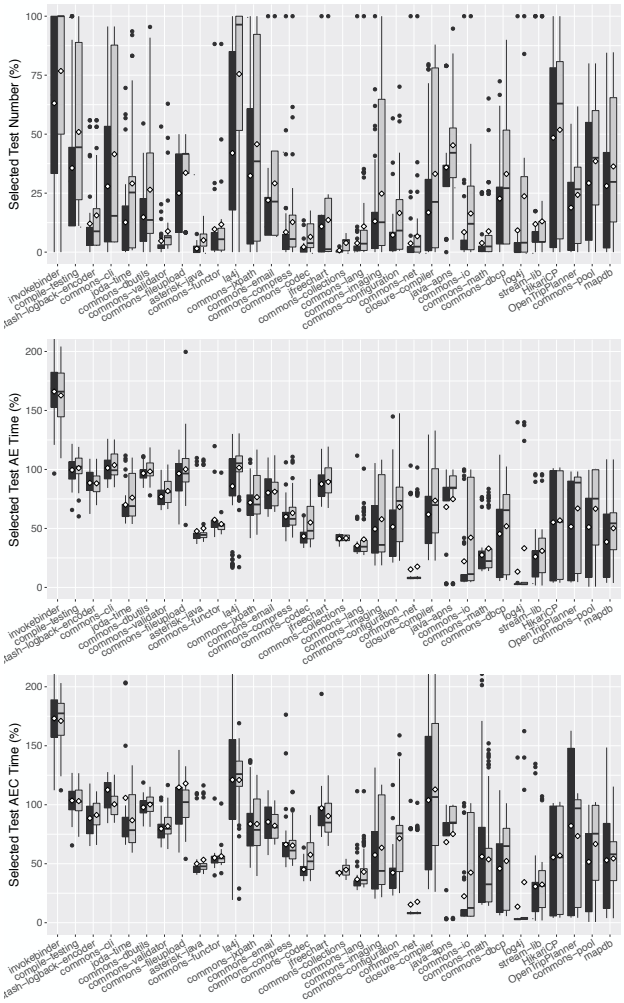

Figure 3: Comparison between basic HyRTS and FRTS

the AE (Analysis and Execution) time for the *offline* mode and the AEC (Analysis, Execution, and Collection) time for the *online* mode.

All our experiments are performed on a 3.70GHz Intel(R) Xeon(R) E5-1620 V2 machine with 128GB of RAM, running Ubuntu Linux 14.04.5 LTS and Oracle Java 64-Bit Server version 1.8.0_101. We apply each studied RTS variant on each code revision with actual code changes (otherwise RTS should not be applied). Our experimental data and implementation are available online [2].

5 RESULT ANALYSIS

5.1 RQ1: Basic Hybrid RTS vs. File-level RTS

Figure 3 presents the comparison results between our basic HyRTS and state-of-the-art FRTS. The three sub-figures present the results in terms of selected test ratio, the *offline* end-to-end time (i.e., the AE time), and the *online* end-to-end time (i.e., the AEC time), respectively. Note that the two time metrics have been normalized into ratios with respect to the original test execution time for the ease of presentation. In each sub figure, the *x*-axis presents all the subjects, while the *y*-axis presents the corresponding metric distributions

using boxplots (with median values marked as lines and mean values marked as hollow diamonds). From the first sub-figure, we can observe that HyRTS consistently selects fewer tests compared with FRTS across *all* the studied subjects. On average, HyRTS and FRTS select 18.35% and 27.18% of all the tests, respectively (i.e., HyRTS is 8.83*pp* more precise). This is expected, since the finer-grained analysis within HyRTS can select tests more precisely in case of CF changes. From the second sub-figure, we found that the HyRTS can outperform FRTS on the *majority* of subjects in terms of the *offline* testing time, demonstrating the effectiveness of the HyRTS technique. For example, on average, FRTS and HyRTS cost 54.35% and 42.87% of the original test execution time, respectively (i.e., HyRTS is 21.1% faster). Different from the observations on selected test ratio, we also found that HyRTS tends to outperform FRTS more on subjects with long running time, since the overhead incurred by the method-level analysis may take a larger ratio for short-running subjects. From the last sub-figure, we find that the performance difference between HyRTS and FRTS is the least on *online* AEC time. For example, the average AEC time across all subjects is 58.67% and 53.66% of the original testing time for FRTS and HyRTS, respectively. Actually, HyRTS may even cost more than FRTS for several subjects with long-running time (e.g., OpenTripPlanner). The main reason is that the method-level dependency information (including the runtime type information for each instance method invocation for safe RTS) required by HyRTS can be more costly to collect than the file-level dependency information required by FRTS.

5.2 RQ2: Impacts of Fine-grained Changes

In this section, we further study the impacts of each type of method-level changes (by transforming it into file-level changes shown in Section 3.2.1) to explore the potential directions for further improving HyRTS. Tables 4 and 5 present the results in terms of the selected test ratio and *offline* AE time, respectively. Note that the impacts for AEC time are quite similar to those for AE time, therefore, we skip those results due to the space limitation. In each table, Column 1 presents all the studied subjects. Columns 2 presents the selected test ratio or AE time by the basic HyRTS technique. Columns 3 to 14 present the increased/decreased selected test ratio or AE time when transforming each type of method-level changes into file-level changes. Finally, Column 15 presents the increased/decreased selected test ratio or AE time by the FRTS technique as a reference. To further understand the results, we perform the Wilcoxon Signed-Rank Test [45] (α=0.05) to compare the result difference on all the revisions of each subject, because it is suitable even for the case that the sample differences may not be normally distributed. We highlight the cells with statistical differences in gray, and also use ❍, ✔, and ✖ to denote "no statistical difference", "significantly better", and "significantly worse", respectively. From the table, we have the following observations:

First, overall, FRTS performs the worst comparing with transforming any type of method-level changes into file-level changes. For example, the selected test ratio increase for FRTS is 8.83%, while it is only 3.68% when transforming CIM changes, the type of method-level changes with the highest impact. The findings on AE time are also similar. Furthermore, FRTS sometimes selects more tests than transforming all method-level changes into file-level changes. For example, when Mapdb evolves from revision `45e7679` to revision `1de4c60`, the ordering of two methods is changed in file `org/mapdb/HTreeMap$values$1.class`. Although such modification does not impact any method bytecode nor dynamic program behavior, the class file is actually changed. Therefore, based on this finding, it is possible to design more cost-effective HyRTS variants by further transforming a subset of method-level changes into file-level changes.

Second, transforming initializer changes into file-level changes cannot impact the RTS effectiveness much in terms of both selected test ratio and testing time. For example, the initializer changes at most incur 0.41% increase in selected test ratio for CSI, and 0.70s increase in AE time for CI. We looked into the code and found the reason to be that whenever a test accesses a class, it usually also has to invoke the class's static initializer or instance initializer. Therefore, transforming the initializer changes into file-level changes won't impact the RTS results much. This finding shows that it is not necessary to build HyRTS specifically considering initializer changes since they won't impact the RTS results much.

Third, method-body changes (e.g., CIM and CSM) usually have the most impacts on the RTS results. For example, among the changes on instance non-initializer methods, CIM has the highest average impacts on both selected test ratio and AE time (e.g., significantly worse than HyRTS for 23 and 8 subjects, respectively). Similarly, CSM also has high impacts among the changes on static non-initializer methods (e.g., significantly worse than HyRTS for 8 and 3 subjects in terms of selected test ratio and AE time, respectively). This finding shows that method-body changes have high impact on RTS effectiveness, and may deserve finer-grained analysis (e.g., at the basic-block level) to further improve RTS.

Fourth, instance method additions and deletions (i.e., AIM and DIM) have low to moderate impacts on RTS effectiveness. For example, transforming AIM and DIM changes into file-level changes only incurs 1.64s and 0.23s increases in AE time, respectively. We find the main reason to be that AIM and DIM changes are not as prevalent as CIM changes, e.g., CIM changes are 1.6X as many as the sum of AIM and DIM changes on average. We also find that AIM changes tend to have higher impacts than DIM changes. The reason is that there is usually a latency to add tests to execute the newly added methods, thus transforming AIM to file-level changes can cause to select many tests that do not directly execute the newly added methods. Therefore, transforming AIM changes into file-level changes may incur to select more tests than transforming DIM changes. The low to moderate impacts of AIM and DIM changes indicate that it may be possible to transform both AIM and DIM changes into file-level changes to have more powerful RTS. The reason is that without AIM and DIM changes, it is not necessary to consider the class-inheritance changes (i.e., computing the LC changes) and trace runtime type information for each instance method invocation (which can be expensive) for safe RTS [23] (Section 3.2.3).

5.3 RQ3: More Hybrid RTS Variants

Based on the four findings learnt from the above study, we come up with two new HyRTS variants, HyRTS$_B$ (Section 3.2.2) and HyRTS$_F$ (Section 3.2.3), further optimizing HyRTS in the following two directions: (1) even finer-grained analysis in case of method-body

Subs	HyRTS	Transformed method-level Changes												FRTS
		DSI	ASI	CSI	DI	AI	CI	DSM	ASM	CSM	DIM	AIM	CIM	
invokebinder	63.11%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	1.09% ❍	0.00% ❍	0.00% ❍	2.19% ❍	0.00% ❍	1.09% ❍	9.84% ✖	3.28% ❍	13.66% ✖
compile-testing	35.67%	0.00% ❍	0.00% ❍	0.00% ❍	0.27% ❍	0.27% ❍	0.00% ❍	3.87% ❍	3.87% ❍	6.91% ✖	0.00% ❍	0.00% ❍	5.07% ❍	15.25% ✖
logstash-logback-encoder	12.06%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.67% ❍	0.67% ❍	0.36% ❍	1.79% ✖	1.13% ✖	3.53% ✖
commons-cli	27.89%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	5.55% ❍	3.51% ❍	13.64% ✖
joda-time	12.65%	0.00% ❍	0.00% ❍	1.12% ❍	0.00% ❍	0.00% ❍	0.08% ❍	1.04% ❍	5.68% ❍	7.15% ❍	0.08% ❍	3.36% ✖	3.39% ✖	16.45% ✖
commons-dbutils	14.83%	0.00% ❍	0.00% ❍	3.85% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.49% ❍	0.00% ❍	1.25% ❍	5.10% ✖	11.61% ✖
commons-validator	4.77%	0.00% ❍	0.00% ❍	0.66% ✖	0.00% ❍	0.00% ❍	0.41% ❍	0.00% ❍	0.11% ❍	0.74% ❍	0.00% ❍	0.41% ✖	2.20% ✖	4.09% ✖
commons-fileupload	25.00%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.60% ❍	0.30% ❍	0.30% ❍	6.55% ✖	8.63% ✖
asterisk-java	1.50%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.06% ❍	0.69% ✖	0.87% ✖	1.00% ✖	3.56% ✖
commons-functor	9.72%	0.00% ❍	0.00% ❍	0.12% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.15% ❍	1.69% ✖	2.02% ✖
la4j	42.01%	0.00% ❍	0.00% ❍	0.58% ❍	0.05% ❍	0.00% ❍	1.35% ❍	0.00% ❍	0.73% ❍	6.47% ✖	4.51% ✖	13.27% ✖	18.31% ✖	33.50% ✖
commons-jxpath	32.36%	0.00% ❍	0.00% ❍	1.06% ❍	0.00% ❍	0.00% ❍	0.00% ❍	1.61% ❍	1.99% ❍	4.89% ✖	0.29% ❍	1.69% ❍	5.04% ✖	13.39% ✖
commons-email	22.02%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	5.95% ❍	1.19% ❍	7.14% ❍
commons-compress	8.48%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	1.30% ✖	0.36% ❍	0.17% ❍	1.18% ❍	0.55% ❍	0.27% ❍	0.67% ✖	1.87% ✖	4.25% ✖
commons-codec	2.17%	0.00% ❍	0.00% ❍	0.12% ❍	0.00% ❍	0.97% ❍	2.18% ✖	1.28% ❍	1.28% ✖	0.61% ❍	0.97% ✖	0.79% ❍	0.43% ❍	4.31% ✖
jfreechart	10.89%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.49% ❍	1.13% ❍	0.00% ❍	0.02% ❍	0.02% ❍	1.92% ✖	2.73% ✖
commons-collections	0.73%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	1.62% ❍	1.18% ❍	0.00% ❍	3.16% ❍
commons-lang	3.81%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.03% ❍	0.13% ❍	2.68% ✖	5.55% ✖	0.03% ❍	0.05% ❍	0.37% ✖	7.11% ✖
commons-imaging	13.07%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.05% ❍	0.00% ❍	0.03% ❍	0.09% ❍	1.75% ❍	1.78% ❍	9.22% ✖	11.72% ✖
commons-configuration	7.25%	0.00% ❍	0.00% ❍	0.40% ❍	0.00% ❍	0.00% ❍	0.02% ❍	0.00% ❍	0.86% ❍	1.37% ❍	0.33% ❍	4.05% ✖	3.45% ✖	9.33% ✖
commons-net	3.72%	0.00% ❍	0.00% ❍	0.08% ❍	0.00% ❍	0.65% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.15% ❍	2.30% ✖	2.99% ✖
closure-compiler	16.80%	0.00% ❍	0.00% ❍	1.92% ❍	0.02% ❍	0.02% ❍	0.03% ❍	0.04% ❍	1.43% ✖	4.04% ✖	1.39% ✖	2.06% ✖	10.09% ✖	16.38% ✖
java-apns	35.89%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	2.32% ❍	1.05% ❍	0.00% ❍	2.36% ✖	4.12% ✖	3.44% ✖	9.41% ✖
commons-io	8.49%	0.00% ❍	0.00% ❍	0.90% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.49% ❍	6.84% ✖	0.00% ❍	0.00% ❍	0.03% ❍	7.88% ✖
commons-math	3.86%	0.00% ❍	0.00% ❍	0.02% ❍	0.07% ❍	0.08% ❍	0.04% ❍	0.00% ❍	1.11% ❍	0.53% ✖	0.00% ❍	0.18% ❍	1.56% ✖	5.00% ✖
commons-dbcp	22.68%	0.00% ❍	0.00% ❍	1.48% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	1.46% ❍	8.99% ✖	10.47% ✖
log4j	9.23%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	1.23% ❍	0.00% ❍	6.46% ❍	0.00% ❍	0.00% ❍	1.23% ❍	2.77% ❍	14.46% ❍
stream-lib	11.89%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.57% ✖	0.25% ❍	0.49% ❍	0.00% ❍	0.08% ❍	0.33% ❍	0.08% ❍	1.15% ✖
HikariCP	48.46%	0.00% ❍	0.00% ❍	0.33% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.08% ❍	0.08% ❍	0.08% ❍	0.00% ❍	0.25% ❍	2.45% ✖	3.37% ✖
OpenTripPlanner	18.83%	0.01% ❍	0.01% ❍	0.31% ❍	0.00% ❍	0.01% ❍	0.06% ✖	0.22% ❍	0.42% ❍	0.51% ✖	1.12% ✖	1.67% ✖	3.59% ✖	5.44% ✖
commons-pool	29.26%	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.00% ❍	0.29% ❍	0.00% ❍	0.15% ❍	0.00% ❍	0.00% ❍	3.09% ✖	4.56% ✖	9.26% ✖
mapdb	28.06%	0.00% ❍	0.00% ❍	0.05% ❍	0.00% ❍	0.00% ❍	0.25% ❍	0.00% ❍	0.72% ❍	0.77% ❍	0.00% ❍	1.11% ❍	3.19% ✖	7.67% ✖
Avg.	18.35%	0.00%	0.00%	0.41%	0.01%	0.14%	0.22%	0.36%	1.09%	1.53%	0.54%	2.14%	3.68%	8.83%

Table 4: Selection ratio change when transforming different atomic changes into file changes

Subs	HyRTS	Transformed method-level Changes												FRTS
		DSI	ASI	CSI	DI	AI	CI	DSM	ASM	CSM	DIM	AIM	CIM	
invokebinder	1.58s	0.00s ❍	0.00s ❍	0.01s ❍	0.00s ❍	0.03s ✖	0.00s ❍	-0.01s ❍	0.00s ❍	-0.01s ❍	-0.02s ❍	0.03s ❍	-0.01s ❍	-0.03s ❍
compile-testing	3.14s	0.00s ❍	0.00s ❍	0.00s ❍	0.01s ❍	0.01s ❍	-0.01s ❍	0.01s ❍	-0.01s ❍	0.04s ❍	0.01s ❍	-0.01s ❍	-0.04s ❍	0.06s ❍
logstash-logback-encoder	3.18s	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	-0.01s ❍	-0.02s ❍	0.01s ❍	0.00s ❍	-0.01s ❍	-0.01s ❍	-0.02s ❍
commons-cli	3.62s	0.00s ❍	0.00s ❍	-0.03s ❍	0.00s ❍	0.00s ❍	-0.04s ❍	0.00s ❍	0.00s ❍	-0.01s ❍	0.00s ❍	0.01s ❍	-0.01s ❍	0.08s ❍
joda-time	3.75s	0.00s ❍	0.00s ❍	0.01s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.03s ❍	0.16s ❍	0.12s ❍	0.02s ❍	0.09s ❍	0.25s ✖	0.33s ❍
commons-dbutils	3.29s	0.00s ❍	0.00s ❍	-0.01s ❍	0.00s ❍	0.00s ❍	-0.04s ❍	0.00s ❍	0.00s ❍	0.00s ❍	-0.01s ❍	0.00s ❍	0.02s ❍	0.04s ❍
commons-validator	3.72s	0.00s ❍	0.00s ❍	0.04s ❍	0.00s ❍	0.00s ❍	0.02s ❍	0.00s ❍	0.01s ❍	0.05s ❍	0.00s ❍	0.02s ✖	0.04s ❍	0.23s ✖
commons-fileupload	3.91s	0.00s ❍	-0.01s ❍	0.00s ❍	0.00s ❍	-0.21s ❍	-0.21s ❍	0.00s ❍	0.00s ❍	0.22s ❍	0.03s ❍	-0.07s ❍	0.68s ✖	0.13s ❍
asterisk-java	3.26s	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.03s ❍	0.04s ✖	0.04s ❍	0.16s ❍
commons-functor	4.02s	0.00s ❍	0.00s ❍	0.01s ❍	-0.01s ❍	0.01s ❍	0.05s ❍	0.01s ❍	0.00s ❍	0.00s ❍	-0.02s ❍	-0.15s ❍	-0.19s ❍	-0.25s ❍
la4j	2.49s	0.00s ❍	0.00s ❍	0.01s ❍	0.00s ❍	0.00s ❍	0.08s ❍	0.00s ❍	0.00s ❍	0.26s ✖	0.04s ✖	0.33s ✖	0.34s ✖	0.93s ✖
commons-jxpath	3.54s	0.00s ❍	0.00s ❍	0.02s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.04s ❍	0.03s ❍	0.10s ❍	0.01s ❍	0.03s ❍	0.07s ❍	0.17s ❍
commons-email	6.29s	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	-0.02s ❍	0.13s ❍
commons-compress	6.09s	0.00s ❍	0.00s ❍	0.01s ❍	0.00s ❍	0.08s ✖	0.16s ✖	0.06s ❍	0.13s ✖	0.11s ❍	-0.04s ❍	0.10s ❍	0.08s ❍	0.32s ❍
commons-codec	4.93s	-0.01s ❍	0.00s ❍	0.05s ❍	-0.01s ❍	0.33s ❍	0.73s ✖	0.44s ❍	0.50s ✖	0.21s ❍	0.53s ✖	0.24s ❍	0.42s ❍	1.31s ✖
jfreechart	9.80s	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.04s ❍	0.12s ❍	0.00s ❍	-0.11s ❍	-0.15s ❍	0.21s ❍	0.18s ❍
commons-collections	8.72s	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	-0.15s ❍	0.00s ❍	0.20s ❍	0.52s ❍	0.06s ❍	0.13s ❍
commons-lang	8.58s	0.00s ❍	0.00s ❍	-0.01s ❍	0.01s ❍	-0.01s ❍	0.03s ❍	0.05s ❍	0.60s ✖	1.35s ✖	0.02s ❍	0.04s ❍	0.18s ❍	1.31s ❍
commons-imaging	14.35s	0.00s ❍	0.00s ❍	0.06s ❍	0.02s ❍	0.05s ❍	0.27s ❍	0.00s ❍	0.02s ❍	-0.04s ❍	0.46s ❍	0.50s ❍	2.10s ✖	2.48s ✖
commons-configuration	16.12s	0.00s ❍	0.00s ❍	0.11s ❍	0.00s ❍	0.00s ❍	-0.01s ❍	0.00s ❍	0.12s ❍	0.03s ❍	-0.03s ❍	2.98s ✖	2.28s ❍	5.18s ✖
commons-net	9.37s	0.00s ❍	0.00s ❍	0.04s ❍	0.00s ❍	-0.01s ❍	-0.02s ❍	0.00s ❍	0.00s ❍	0.04s ❍	0.00s ❍	-0.03s ❍	1.44s ❍	1.43s ❍
closure-compiler	38.41s	0.00s ❍	0.00s ❍	0.52s ❍	-0.05s ❍	0.05s ❍	-0.02s ❍	0.03s ❍	0.44s ❍	1.19s ❍	0.65s ❍	1.00s ❍	5.44s ✖	7.34s ✖
java-apns	49.16s	0.00s ❍	0.04s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.08s ❍	0.44s ❍	1.67s ❍	0.11s ❍	2.04s ✖	3.68s ✖	3.00s ✖	4.79s ✖
commons-io	16.59s	0.00s ❍	0.00s ❍	2.63s ❍	0.01s ❍	0.00s ❍	-0.02s ❍	0.00s ❍	1.04s ❍	14.03s ✖	0.03s ❍	0.02s ❍	-0.01s ❍	15.06s ✖
commons-math	20.71s	0.00s ❍	0.00s ❍	-0.02s ❍	0.00s ❍	-0.01s ❍	0.01s ❍	-0.01s ❍	0.95s ❍	0.56s ❍	-0.01s ❍	0.01s ❍	1.98s ❍	4.18s ❍
commons-dbcp	36.16s	0.01s ❍	0.00s ❍	1.73s ❍	0.00s ❍	-0.37s ❍	-0.37s ❍	0.00s ❍	0.04s ❍	0.00s ❍	0.03s ❍	1.31s ❍	3.88s ❍	5.35s ✖
log4j	11.59s	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.02s ❍	-0.01s ❍	8.37s ❍	0.02s ❍	0.00s ❍	0.02s ❍	0.04s ❍	19.45s ❍
stream-lib	26.18s	0.00s ❍	0.00s ❍	0.00s ❍	0.01s ❍	0.01s ❍	3.49s ✖	1.62s ❍	2.97s ✖	0.03s ❍	0.09s ❍	0.94s ❍	0.59s ❍	4.91s ✖
HikariCP	45.10s	0.00s ❍	0.00s ❍	0.17s ❍	0.00s ❍	0.00s ❍	0.05s ❍	0.00s ❍	0.00s ❍	-0.02s ❍	0.01s ❍	-0.01s ❍	0.72s ❍	1.29s ❍
OpenTripPlanner	121.88s	0.15s ❍	0.17s ❍	2.50s ❍	0.07s ❍	0.07s ❍	0.17s ❍	2.97s ❍	6.13s ❍	0.91s ❍	3.31s ❍	12.07s ✖	29.17s ❍	37.10s ✖
commons-pool	202.10s	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	0.00s ❍	5.44s ❍	0.00s ❍	6.06s ❍	0.00s ❍	0.00s ❍	15.66s ✖	37.67s ✖	61.40s ✖
mapdb	319.59s	0.00s ❍	-0.07s ❍	0.16s ❍	-0.05s ❍	0.19s ❍	12.65s ✖	0.12s ❍	5.39s ❍	5.26s ❍	0.14s ❍	13.39s ❍	41.51s ✖	95.60s ✖
Avg.	31.60s	0.00s	0.00s	0.25s	0.00s	0.01s	0.70s	0.18s	1.08s	0.77s	0.23s	1.64s	4.12s	8.46s

Table 5: AE time change when transforming different atomic changes into file changes

changes (i.e., CSM and CIM changes), and (2) faster HyRTS analysis via further transforming AIM and DIM changes into file-level changes. Note that HyRTS$_B$ focuses on the selection precision while HyRTS$_F$ focuses on the overall overhead. The main experimental results for comparing different HyRTS variants (i.e., basic HyRTS, HyRTS$_B$ and HyRTS$_F$) are shown in Table 6. In the table, Column "Subs" lists all the studied subjects; Column "Selected Tests" presents the selected test ratios for each HyRTS variant; Columns "AE Time" and "AEC Time" present the AE and AEC end-to-end testing time including the ratio to the original testing time in brackets. We also applied Wilcoxon Signed-Rank Test at the significance level of 0.05 to compare each HyRTS variant against state-of-the-art FRTS. All the cells with statistical differences are marked in gray, and ❍, ✔,

Subs	Selected Tests			AE Time						AEC Time					
	HyRTS	HyRTS$_B$	HyRTS$_F$	HyRTS		HyRTS$_B$		HyRTS$_F$		HyRTS		HyRTS$_B$		HyRTS$_F$	
invokebinder	63.11% ✔	61.47% ✔	72.95% ❍	1.58s ❍	(166.02%)	1.59s ❍	(167.76%)	1.58s ❍	(167.22%)	1.64s ❍	(173.18%)	1.68s ❍	(176.76%)	1.61s ❍	(169.75%)
compile-testing	35.67% ✔	33.57% ✔	35.67% ✔	3.14s ❍	(99.54%)	3.15s ❍	(99.60%)	3.13s ❍	(99.12%)	3.27s ❍	(103.52%)	3.21s ❍	(101.68%)	3.24s ❍	(102.89%)
logstash-logback-encoder	12.06% ✔	11.87% ✔	13.96% ✔	3.18s ❍	(88.60%)	3.13s ❍	(87.35%)	3.21s ❍	(89.28%)	3.17s ✔	(88.58%)	3.11s ✔	(87.07%)	3.19s ❍	(89.22%)
commons-cli	27.89% ✔	27.69% ✔	33.44% ✔	3.62s ❍	(101.26%)	3.64s ❍	(102.23%)	3.64s ❍	(101.86%)	4.01s ✖	(112.62%)	3.76s ✖	(105.30%)	3.73s ❍	(105.12%)
joda-time	12.65% ✔	12.05% ✔	16.11% ✔	3.75s ❍	(70.05%)	3.93s ❍	(73.41%)	3.89s ❍	(72.72%)	5.64s ✖	(105.80%)	5.86s ✖	(109.95%)	4.35s ❍	(81.27%)
commons-dbutils	14.83% ✔	15.16% ✔	16.73% ✔	3.29s ❍	(96.76%)	3.42s ❍	(100.82%)	3.24s ❍	(95.67%)	3.30s ❍	(97.78%)	3.40s ❍	(100.69%)	3.36s ❍	(99.20%)
commons-validator	4.77% ✔	4.66% ✔	5.55% ✔	3.72s ✔	(76.93%)	3.83s ❍	(79.02%)	3.72s ✔	(76.86%)	3.85s ❍	(79.65%)	4.06s ❍	(84.11%)	3.84s ❍	(79.58%)
commons-fileupload	25.00% ✔	21.43% ✔	25.59% ✔	3.91s ❍	(96.53%)	4.09s ❍	(99.34%)	4.26s ❍	(105.99%)	4.60s ❍	(114.69%)	4.49s ❍	(111.37%)	4.11s ❍	(101.20%)
asterisk-java	1.50% ✔	1.44% ✔	3.12% ✔	3.26s ❍	(47.70%)	3.25s ✔	(47.54%)	3.31s ❍	(48.44%)	3.42s ✔	(49.99%)	3.46s ✔	(50.61%)	3.47s ❍	(50.83%)
commons-functor	9.72% ✔	9.72% ✔	9.87% ✔	4.02s ❍	(57.34%)	3.94s ✖	(56.07%)	3.79s ❍	(54.07%)	3.85s ❍	(55.00%)	3.93s ❍	(55.92%)	3.93s ❍	(56.10%)
la4j	42.01% ✔	41.67% ✔	56.44% ✔	2.49s ✔	(85.80%)	2.54s ✔	(87.77%)	2.83s ❍	(93.87%)	4.19s ❍	(121.15%)	6.10s ✖	(178.94%)	3.46s ❍	(112.79%)
commons-jxpath	32.36% ✔	28.03% ✔	34.22% ✔	3.54s ❍	(71.92%)	3.52s ❍	(71.56%)	3.52s ❍	(71.93%)	4.13s ❍	(83.74%)	4.97s ❍	(98.46%)	3.78s ❍	(77.00%)
commons-email	22.02% ❍	19.64% ❍	27.97% ❍	6.29s ❍	(80.23%)	6.44s ❍	(81.63%)	6.31s ❍	(81.18%)	6.69s ❍	(85.45%)	6.70s ❍	(85.26%)	6.91s ❍	(88.09%)
commons-compress	8.48% ✔	8.10% ✔	9.15% ✔	6.09s ❍	(59.87%)	6.53s ❍	(64.17%)	6.19s ❍	(60.96%)	6.74s ❍	(66.28%)	7.34s ❍	(72.32%)	6.46s ❍	(63.55%)
commons-codec	2.17% ✔	100.00% ✖	3.51% ✔	4.93s ✔	(43.35%)	13.60s ✖	(119.53%)	5.75s ❍	(50.70%)	5.14s ✔	(45.15%)	13.60s ✖	(119.53%)	5.77s ✔	(50.84%)
jfreechart	10.89% ✔	10.89% ✔	10.91% ✔	9.80s ❍	(87.66%)	10.08s ❍	(90.36%)	9.82s ❍	(88.03%)	10.86s ❍	(97.09%)	12.02s ✖	(107.50%)	10.34s ❍	(92.26%)
commons-collections	0.73% ❍	0.73% ❍	3.52% ❍	8.72s ❍	(41.30%)	8.36s ❍	(39.53%)	9.09s ❍	(43.12%)	8.91s ❍	(42.24%)	9.22s ❍	(43.66%)	9.03s ❍	(42.76%)
commons-lang	3.81% ✔	3.71% ✔	3.93% ✔	8.58s ❍	(35.38%)	8.76s ❍	(36.06%)	8.73s ❍	(35.94%)	8.92s ✔	(36.78%)	9.31s ❍	(38.39%)	9.10s ✔	(37.48%)
commons-imaging	13.07% ✔	13.05% ✔	14.97% ✔	14.35s ✔	(49.50%)	14.54s ✔	(50.14%)	14.64s ✔	(50.54%)	16.64s ❍	(57.39%)	27.61s ✖	(94.76%)	15.80s ✔	(54.59%)
commons-configuration	7.25% ✔	7.19% ✔	11.30% ✔	16.12s ✔	(51.36%)	16.21s ✔	(51.63%)	19.02s ❍	(60.93%)	13.29s ✔	(42.39%)	13.66s ✔	(43.56%)	14.36s ✔	(45.93%)
commons-net	3.72% ✔	3.23% ✔	3.95% ✔	9.37s ❍	(15.30%)	9.35s ❍	(15.27%)	10.05s ❍	(16.42%)	9.41s ❍	(15.36%)	9.60s ❍	(15.67%)	10.14s ❍	(16.57%)
closure-compiler	16.80% ✔	15.09% ✔	19.64% ✔	38.41s ✔	(61.88%)	38.10s ✔	(61.25%)	39.86s ✔	(64.25%)	64.45s ❍	(103.88%)	71.72s ❍	(115.65%)	55.44s ✔	(89.41%)
java-apns	35.89% ✔	34.42% ✔	40.85% ✔	49.16s ✔	(68.18%)	47.10s ✔	(65.37%)	52.80s ❍	(73.26%)	49.27s ✔	(68.33%)	47.11s ✔	(65.38%)	52.85s ❍	(73.32%)
commons-io	8.49% ✔	7.62% ✔	8.49% ✔	16.59s ✔	(22.12%)	14.48s ✔	(19.31%)	16.68s ✔	(22.23%)	16.83s ✔	(22.44%)	14.74s ✔	(19.66%)	16.77s ✔	(22.36%)
commons-math	3.86% ✔	3.39% ✔	4.03% ✔	20.71s ❍	(27.45%)	19.78s ❍	(26.18%)	21.11s ❍	(28.00%)	42.11s ✖	(56.02%)	70.29s ✖	(93.23%)	35.94s ❍	(47.82%)
commons-dbcp	22.68% ✔	20.70% ✔	23.39% ✔	36.16s ✔	(45.40%)	33.49s ✔	(42.02%)	35.40s ❍	(44.38%)	36.57s ❍	(45.92%)	33.76s ❍	(42.37%)	35.46s ✔	(44.46%)
log4j	9.23% ❍	8.92% ❍	10.15% ❍	11.59s ❍	(13.36%)	11.62s ❍	(13.39%)	11.67s ❍	(13.45%)	11.80s ❍	(13.57%)	11.87s ❍	(13.66%)	11.67s ✔	(13.43%)
stream-lib	11.89% ✔	11.89% ✔	12.30% ✔	26.18s ✔	(26.03%)	26.41s ✔	(26.26%)	27.23s ❍	(27.11%)	30.87s ❍	(30.70%)	60.26s ✖	(59.94%)	28.99s ❍	(28.88%)
HikariCP	48.46% ✔	45.39% ✔	48.71% ✔	45.10s ❍	(55.21%)	43.81s ❍	(53.70%)	45.16s ❍	(55.34%)	45.23s ❍	(55.41%)	44.03s ❍	(54.03%)	45.26s ❍	(55.46%)
OpenTripPlanner	18.83% ✔	17.42% ✔	21.10% ✔	121.88s ✔	(51.48%)	109.49s ✔	(46.37%)	137.34s ✔	(57.83%)	194.21s ❍	(82.08%)	243.13s ✖	(103.08%)	151.80s ❍	(63.92%)
commons-pool	29.26% ✔	27.94% ✔	32.94% ✔	202.10s ✔	(51.16%)	202.53s ✔	(51.27%)	218.42s ✔	(55.30%)	203.81s ✔	(51.59%)	200.24s ✔	(50.69%)	216.54s ✔	(54.83%)
mapdb	28.04% ✔	25.84% ✔	29.05% ✔	319.59s ✔	(38.58%)	270.82s ✔	(32.89%)	332.87s ✔	(40.19%)	438.99s ❍	(52.97%)	530.96s ❍	(64.47%)	357.80s ✔	(43.19%)
Avg.	18.35%	20.43%	20.74%	31.60s	(42.87%)	29.73s	(40.33%)	33.38s	(45.28%)	39.56s	(53.66%)	46.41s	(62.96%)	35.58s	(48.26%)

Table 6: Experimental results for HyRTS variants

and ✖ represent "no statistical difference", "significantly better", and "significantly worse". According to the results shown in Table 6:

HyRTS$_B$ selects similar ratio of tests with basic HyRTS, which is counter-intuitive. We looked into the data and found two reasons. First, modern system design principles recommend writing simple method bodies for the ease of maintenance, making the majority of method body changes directly occur on the first basic block of the methods. In such cases, the detailed basic-block-level analysis selects similar number of tests with basic HyRTS. For example, when Invokebinder evolves from `c35f3ee` to `9c59df3`, method `SmartBinder.from()`, the only changed source method actually only has one line of code. Second, the basic-block-level dependency collection failed for one subject, Commons-Codec, which has several huge methods (e.g., `initSTRINGS` and `initBYTES`) inside class `Base64Codec13Test`. After the detailed code instrumentation for tracing the basic-block-level test dependencies, the code size became larger than the JVM specified maximum size (i.e., 64KB), crashing JVM with exception "`java.lang.RuntimeException: Method code too large!`". In such cases, our implementation simply reruns all the regression tests to ensure safety. That's actually why HyRTS$_B$ on average selects even slightly more tests than basic HyRTS. Furthermore, HyRTS$_B$ may perform even worse than basic HyRTS and FRTS in terms of AE or AEC time. Based on the Wilcoxon test, HyRTS$_B$ costs significantly more AE/AEC time than FRTS on 2/9 subjects due to the additional overhead for analyzing basic-block changes and tracing basic-block dependencies. On average HyRTS$_B$ costs 46.41s AEC time, which is even higher than that of FRTS (43.25s). Therefore, including finer-grained analysis may not be a good direction for more practical RTS.

HyRTS$_F$ extends basic HyRTS by further transforming AIM and DIM changes into file-level changes. Although HyRTS$_F$ may select more tests than basic HyRTS, without AIM and DIM changes, HyRTS$_F$ does not need to consider the class inheritance hierarchy changes (i.e., computing LC changes) or trace the runtime type information for each instance method invocation, and thus may be much faster than the basic HyRTS. According to Table 6, HyRTS$_F$ incurs small increase in selected test ratio (2.39%) and AE time (1.78s) compared with basic HyRTS due to the more imprecise analysis. However, in terms of the AEC time, HyRTS$_F$ is even much more efficient than both basic HyRTS and state-of-the-art FRTS. For example, HyRTS$_F$ only costs 48.26% of the original testing time (35.58s), while FRTS and HyRTS cost 58.67% (43.25s) and 53.66% (39.56s), respectively. Furthermore, HyRTS$_F$ is never statistically significantly worse than FRTS in terms of all the used metrics while the basic HyRTS is significantly worse than FRTS on three subjects in terms of AEC time. Overall, HyRTS$_F$ is 16.7%/17.7% faster than state-of-the-art FRTS in AE/AEC time, and can be a more cost-effective technique than the basic HyRTS in practice.

5.4 Threats to Validity

Threats to Internal Validity. The main threat to internal validity mainly lies in the implementation of the RTS techniques studied in the work. To reduce this threat, we built the proposed and studied techniques on mature frameworks/libraries (e.g., ASM and JavaAgent), and carefully reviewed our code and experiment scripts before and during the experimental study. Furthermore, since the binary version of the FRTS tool Ekstazi is publicly available, we also compared our FRTS implementation with Ekstazi in terms of selected test ratio and the end-to-end AEC time on sampled projects. We found that our FRTS selects the same number of tests with Ekstazi, and have competitive end-to-end time. To illustrate, Figure 4 presents the selected test number and AEC time for Ekstazi and our FRTS on Commons-Math. We can observe that Ekstazi and FRTS

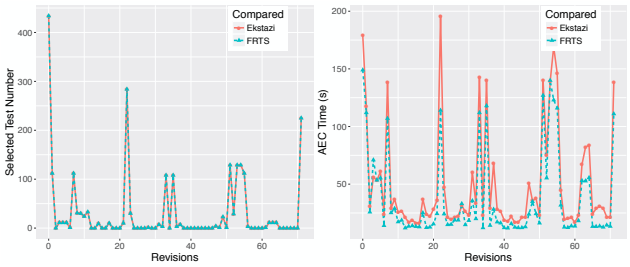

Figure 4: Ekstazi vs. FRTS on Commons-Math revisions

select exactly the same number of tests for all the studied revisions of Commons-Math with quite close end-to-end testing time.

Threats to External Validity. The main threat to external validity mainly lies in the subject systems used in this study. To reduce this threat, we included all the single-module Maven projects used in recent RTS work [23, 30], and strictly followed prior RTS work in selecting project revisions. Actually, our study includes all the projects used on Legunsen et al.'s work [30], and involves much more code revisions than recent RTS work. However, it is still not clear whether our findings can generalize to other projects.

Threats to Construct Validity. The main threat to construct validity lies in the metrics that we used to evaluate the studied RTS techniques. To reduce this threat, we use all the three widely used metrics in RTS, i.e., the selected test ratio, the *offline* testing time (i.e., AE time), and the *online* testing time (i.e., AEC time).

6 RELATED WORK

Dynamic RTS. Rothermel and Harrold [37] firstly investigated dynamic RTS for C programs at the basic-block granularity. Harrold et al. [26] then extended the basic-block-level RTS to Java programs with Object-Oriented features. Orso et al. [34] further proposed a two-phase RTS analysis, including a partitioning phase to exclude the non-affected classes from detailed CFG analysis and a selection phase that performs CFG analysis only on the remaining classes. Note that our work is different from their work: (1) their work simply excludes the non-affected classes and still performs uniform detailed analysis (i.e., basic-block-level analysis) on the affected classes, whereas our work performs RTS analysis at multiple levels in tandem (with only detailed analysis when necessary); (2) our work demonstrates that combining method-level and file-level analysis can outperform state-of-the-art file-level RTS, while further including basic-block-level analysis is not cost-effective.

Since finer-grained analysis may incur larger overheads, besides the basic-block-level RTS, researchers have also investigated RTS at coarser-granularities to improve the efficiency of RTS. Ren et al. [35] and Zhang et al. [50] performed RTS at the method level – they performed bytecode or source code analysis to detect the changed methods/fields and then selected any test that executed the changed methods/fields in the old program version. With the increasing scales of modern real-world projects, method-level RTS may still incur large overheads. Therefore, Gligoric et al. [23] proposed file-level RTS for Java projects, which traces the changed bytecode class files based on fast checksum computation, and selects any test accessing the changed files. Although imprecise, the file-level RTS can have negligible overhead and has been shown to outperform the method-level RTS in terms of end-to-end testing time. Vasic et al. also compared file-level RTS with even coarser grained RTS at the module level for .NET programs [44]. Recently, Celik et al. [16] designed dynamic file-level RTS across JVM boundaries. Despite its effectiveness, file-level RTS may still select more tests due to the coarse-grained analysis. Therefore, in this work, we propose the first hybrid RTS approach to combine the strengths of RTS at multiple granularities. Our work differs from all prior RTS techniques that only work at a fixed granularity and opens a new dimension for further advancing RTS.

Static RTS. Although dynamic RTS has been widely studied, it may not be suitable for all types of systems. For example, the dynamic test dependencies required by dynamic RTS may be challenging to collect for real-time systems since the code instrumentation may break the time constraints and interrupt normal test runs. Therefore, static RTS that uses static analysis to over-approximate test dependencies has also been proposed to further complement dynamic RTS. Kung et al. [29] proposed the first static RTS technique based on the *class firewall* analysis, which computes classes that may be affected by the changes using static class analysis. Since class firewall analysis may be imprecise, Ryder and Tip [39] further proposed static RTS at the method level, i.e., using static call graphs to over-approximate the dependencies for each test. Although the static RTS techniques have been proposed for decades, their effectiveness have been largely unknown due to the lack of studies on modern software systems. Legunsen et al. [30, 31] performed a timely and extensive study on static RTS recently, and showed that static file-level RTS can have close end-to-end testing time with state-of-the-art dynamic file-level RTS, but is sometimes unsafe due to reflections. In this paper, we propose to combine the strengths of both fine and coarse grained dynamic RTS analysis. Actually our idea is general and can also be applied to static RTS and other levels. We plan to further explore this direction in the future.

7 CONCLUSION

This paper proposes the first hybrid RTS approach that analyzes at multiple granularities to combine the strengths of traditional RTS techniques at different granularities. We evaluate the proposed approach in both the *online* and *offline* modes on 2707 revisions of 32 projects, totalling over 124 Million LoC. The study shows that HyRTS, our first hybrid technique that combines method and file granularity RTS analysis, can be significantly faster than state-of-the-art FRTS in the *offline* mode, but sometimes slower than FRTS in the *online* mode due to the collection of method-level dependencies. We then further studied the impact of each type of method-level changes on the RTS results, and designed two new HyRTS variants based on the study results. The additional study shows that further integrating finer-grained analysis at the basic-block level is not cost-effective, whereas transforming instance method additions and deletions into file-level changes can produce a cost-effective HyRTS variant that consistently outperforms existing FRTS for both *online* and *offline* modes.

8 ACKNOWLEDGMENTS

We thank the anonymous reviewers for the valuable comments. This work is supported in part by NSF Grant No. CCF-1566589, UT Dallas start-up fund, Google, Huawei, and Samsung.

REFERENCES

[1] SLOCCount. http://www.dwheeler.com/sloccount/.
[2] HyRTS Homepage. http://hyrts.org/.
[3] Java Agent. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html.
[4] JDT home page. http://www.eclipse.org/jdt/.
[5] Maven Failsafe Plugin. http://maven.apache.org/surefire/maven-failsafe-plugin/.
[6] Maven Surefire Plugin. http://maven.apache.org/surefire/maven-surefire-plugin/.
[7] Testing at the speed and scale of Google, Jun 2011. http://goo.gl/2B5cyl.
[8] Tools for continuous integration at Google scale, Jan 2011. https://goo.gl/Gqj7uL.
[9] Apache Camel. http://camel.apache.org/.
[10] Apache Commons Math. https://commons.apache.org/proper/commons-math/.
[11] Apache CXF. https://cxf.apache.org/.
[12] ASM. http://asm.ow2.org/.
[13] T. Ball. On the limit of control flow analysis for regression test selection. *ACM SIGSOFT Software Engineering Notes*, 23(2):134–142, 1998.
[14] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. DeFlaker: Automatically detecting flaky tests. In *International Conference on Software Engineering*, 2018. to appear.
[15] L. C. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on uml designs. In *International Conference on Software Maintenance and Evolution*, pages 252–261, 2002.
[16] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric. Regression test selection across jvm boundaries. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 809–820, 2017.
[17] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie. How do assertions impact coverage-based test-suite reduction? In *International Conference on Software Testing, Verification and Validation*, pages 418–423, 2017.
[18] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche. Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. *Software Testing, Verification and Reliability*, 25(4):371–396, 2015.
[19] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *Transactions on Software Engineering*, 32(9):733–752, 2006.
[20] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 235–245, 2014.
[21] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on software engineering*, 17(6):591–603, 1991.
[22] M. Gligoric, L. Eloussi, and D. Marinov. Ekstazi: Lightweight test selection. In *International Conference on Software Engineering, Tool Demonstration Track*, pages 713–716, 2015.
[23] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *International Symposium on Software Testing and Analysis*, pages 211–222, 2015.
[24] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-demand test suite reduction. In *International Conference on Software Engineering*, pages 738–748, 2012.
[25] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei. A unified test case prioritization approach. *Transactions on Software Engineering and Methodology*, 24(2):10:1–10:31, 2014.
[26] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 312–326, 2001.
[27] H. Hemmati and L. Briand. An industrial investigation of similarity measures for model-based test case selection. In *International Symposium on Software Reliability Engineering*, pages 141–150, 2010.
[28] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on software Engineering*, 29(3):195–209, 2003.
[29] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-Oriented Programming*, 8(2):51–65, 1995.
[30] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. An extensive study of static regression test selection in modern software evolution. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 583–594, 2016.
[31] O. Legunsen, A. Shi, and D. Marinov. Starts: Static regression test selection. In *International Conference on Automated Software Engineering, Tool Demonstration Track*, pages 949–954, 2017.
[32] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang. How does regression test prioritization perform in real-world software evolution? In *International Conference on Software Engineering*, pages 535–546, 2016.
[33] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming google-scale continuous testing. In *International Conference on Software Engineering, Software Engineering in Practice Track*, pages 233–242, 2017.
[34] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 241–251, 2004.
[35] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of Java programs. In *International Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 432–448, 2004.
[36] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, and J. Dolby. Chianti: A prototype change impact analysis tool for Java. Technical Report DCS-TR-533, Rutgers University CS Dept., 2003.
[37] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.
[38] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *International Conference on Software Maintenance and Evolution*, pages 179–189, 1999.
[39] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 46–53, 2001.
[40] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *International Conference on Software Engineering*, volume 1, pages 268–279, 2015.
[41] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 246–256, 2014.
[42] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, and J. Czerwonka. Optimizing test placement for module-level regression testing. In *Proceedings of the 39th International Conference on Software Engineering*, pages 689–699, 2017.
[43] A. Shi, T. Yung, A. Gyori, and D. Marinov. Comparing and combining test-suite reduction and regression test selection. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 237–247, 2015.
[44] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric. File-level vs. module-level regression test selection for. net. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Industry Track*, pages 848–853, 2017.
[45] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.
[46] G. Xu and A. Rountev. Regression test selection for aspectj software. In *International Conference on Software Engineering*, pages 65–74, 2007.
[47] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.
[48] K. Zhai, B. Jiang, and W. K. Chan. Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study. *Transactions on Services Computing*, 7(1):54–67, 2014.
[49] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *International Conference on Software Engineering*, pages 192–201, 2013.
[50] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *International Conference on Software Maintenance and Evolution*, pages 23–32, 2011.
[51] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. An empirical study of JUnit test-suite reduction. In *International Symposium on Software Reliability Engineering*, pages 170–179, 2011.
[52] H. Zhong, L. Zhang, and H. Mei. An experimental study of four typical test suite reduction techniques. *Information and Software Technology*, 50(6):534–546, 2008.