# Regression Test Selection Across JVM Boundaries

Ahmet Celik, Marko Vasic
University of Texas at Austin (USA)
{ahmetcelik,vasic}@utexas.edu

Aleksandar Milicevic
Microsoft (USA)
almili@microsoft.com

Milos Gligoric
University of Texas at Austin (USA)
gligoric@utexas.edu

## ABSTRACT

Modern software development processes recommend that changes be integrated into the main development line of a project multiple times a day. Before a new revision may be integrated, developers practice regression testing to ensure that the latest changes do not break any previously established functionality. The cost of regression testing is high, due to an increase in the number of revisions that are introduced per day, as well as the number of tests developers write per revision. Regression test selection (RTS) optimizes regression testing by skipping tests that are not affected by recent project changes. Existing dynamic RTS techniques support only projects written in a single programming language, which is unfortunate knowing that an open-source project is on average written in several programming languages.

We present the first dynamic RTS technique that does not stop at predefined language boundaries. Our technique dynamically detects, at the operating system level, all file artifacts a test depends on. Our technique is, hence, oblivious to the specific means the test uses to actually access the files: be it through spawning a new process, invoking a system call, invoking a library written in a different language, invoking a library that spawns a process which makes a system call, etc. We also provide a set of extension points which allow for a smooth integration with testing frameworks and build systems. We implemented our technique in a tool called RTSLinux as a loadable Linux kernel module and evaluated it on 21 Java projects that escape the JVM by spawning new processes or invoking native code, totaling 2,050,791 lines of code. Our results show that RTSLinux, on average, skips 74.17% of tests and saves 52.83% of test execution time compared to executing all tests.

## CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*; *Software evolution*;

## KEYWORDS

Regression test selection, language-agnostic

**ACM Reference format:**

Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression Test Selection Across JVM Boundaries. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 12 pages.
https://doi.org/10.1145/3106237.3106297

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany*
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106297

## 1 INTRODUCTION

To check that recent project changes do not break any established functionality developers practice *regression testing*—running available tests against the new changes. Although immensely important, regression testing often comes at a high cost [17, 22, 27, 52, 64]. Google recently reported that the company runs more than 70 million tests daily and the testing cost increases quadratically: both the number of revisions per day and the number of tests per revision increase linearly [3, 79].

**Background**: *Regression test selection* (RTS) optimizes regression testing by skipping tests that are not affected by recent project changes (i.e., changes between two given revisions) [19, 27–29, 37, 66, 67, 69, 70, 77, 79, 84–86]. To determine which tests are affected, *dependencies* on various source artifacts must be established and maintained for each test throughout the life cycle of a program. Before tests are executed, only those tests whose dependencies are invalidated by the recent changes are selected.

An RTS technique typically has to make two big choices regarding tracking dependencies: (1) what kind of dependencies to track (e.g., at what granularity level), and (2) how to track them (e.g., *statically* or *dynamically*). This leads to a whole spectrum of techniques, which all greatly vary in *safety* and *precision*: a safe technique always selects all affected tests, while a precise technique does not select any unaffected test [71].

In this paper, we consider Java projects that are either multilingual (i.e., make native calls via JNI) or span across multiple JVMs (by spawning multiple Java processes). We argue that the existing RTS techniques are not well suited for this class of projects. To address their shortcomings, we propose a novel RTS technique based on *dynamic (system-level) file and process monitoring*.

**Problem**: Existing dynamic RTS techniques [14, 21, 37, 67, 69, 71, 77, 86] are all *language-specific* (i.e., unable to observe dependencies that span across languages), making them *unsafe* for multilingual projects. Recent studies found that an open-source project is written in five programming languages on average [57, 68].

Existing static RTS techniques, on the other hand, are *imprecise*. Due to common limitations of static analyses, such techniques must typically overapproximate the set of dependencies [51, 61, 81, 82]. Examples include Google's continuous integration system TAP [3, 26, 79] and Microsoft's CloudBuild [31], which only track dependencies between projects. Assume, for example, a new method is added to a commonly used collections library; in such a setting, all tests of all dependent projects must be rerun, even though none of them are effectively affected by that change.

**Technique**: We present the first dynamic *language-agnostic RTS technique*, dubbed RTSLinux. RTSLinux collects dependencies at the *file level* granularity. To do so, RTSLinux integrates with the OS kernel (via a loadable kernel module [6]), where it intercepts all relevant process- and file-related system calls. The benefits of such tight

integration with the operating system are twofold: (1) it enables RTSLinux to precisely observe all dynamic dependencies, no matter which part of the test's process tree creates them, and (2) keeps the overhead to a minimum. Finally, RTSLinux provides primitives and extension points for easy integration with testing frameworks and build systems, as well as for further (domain-specific) improvements of the technique's precision.

**Evaluation**: We evaluated RTSLinux on 420 revisions of 21 popular open-source projects. Our evaluation has the following main objectives: (1) evaluate RTSLinux's effectiveness in its own right (in terms of the reduction in testing time), (2) evaluate the added benefits of the key novelty behind RTSLinux—dependency detection across JVM boundaries—comparing to an RTS tool for Java-only projects, and (3) provide initial assessment of the significance of kernel-level integration.

We measured the effectiveness in terms of the reduction in the *end-to-end testing time* and in terms of *the number of skipped tests*. Our results show that RTSLinux, on average, reduces testing time by 52.83% and skips 74.17% of tests (compared to *RetestAll*, i.e., executing all tests at each project revision).

To measure the added benefit, we compared RTSLinux to Ekstazi [35–37], a recent RTS technique that collects dynamic file dependencies for Java projects, but fails to collect dependencies created from child processes or from JNI calls to code that does not compile to Java bytecode. We compared the two techniques in terms of safety, i.e., the percent of dependencies that are missed by Ekstazi, and in terms of reduction in number of tests and testing time. Our findings show Ekstazi is *less safe*, as it collects only (on average) 16.99% of accessed files collected by RTSLinux. Despite being safer than Ekstazi, RTSLinux achieves comparable savings, both in the number of tests skipped and the overall testing time.

## 2 MOTIVATING EXAMPLES

We motivate our work with two sample unit tests taken from popular open-source projects: Hadoop [2] and Ant [1]. The first sample test illustrates the case when code written in multiple languages is executed in the same process. The second sample test illustrates the case when code written in a single language is executed in multiple processes. In neither of these two cases (or combinations thereof) the existing RTS techniques provide safety. We discuss the issues faced by the existing techniques and briefly describe our proposed technique to overcome the challenges.

### 2.1 Multiple Languages and a Single Process

Figure 1 shows a test that invokes C code from Java code; the test is from the Apache Hadoop project [2] (Git SHA: fe6c1bd7), a popular map-reduce framework. The `testSnappyNullPointerException` test method (which is declared in `TestSnappyCompressorDecompressor.java`) invokes (line 7) `compress` method (declared in `SnappyCompressor.java`), which in turn invokes (line 13) `compressBytesDirect` (a *native* method also declared in `SnappyCompressor.java`). The implementation of the native method is in `Java_compressBytesDirect` function (written in C and implemented in `SnappyCompressor.c`). Java uses the Java Native Interface (JNI) to invoke the native code [47, 53]. Note that JNI does not spawn a new process but executes the native code in the same process as the Java Virtual Machine (JVM).

```
1  // TestSnappyCompressorDecompressor.java
2  void testSnappyNullPointerException() {
3      SnappyCompressor compressor =
4          new SnappyCompressor();
5      byte[] bytes = BytesGenerator.get(1024 * 6);
6      compressor.setInput(bytes, 0, bytes.length);
7      compressor.compress(null, 0, 0);
8  ... }
9
10 // SnappyCompressor.java
11 int compress(byte[] b, int off, int len) {...
12     // Compress data
13     n = compressBytesDirect();
14     compressedDirectBuf.limit(n);
15     uncompressedDirectBuf.clear();
16 ... }
17 native int compressBytesDirect();
18
19 // SnappyCompressor.c
20 JNIEXPORT jint JNICALL Java_compressBytesDirect (JNIEnv *
       env, jobject thisj) {
21 ... }
```

**Figure 1: An example test from the Hadoop project that invokes C code from Java (without spawning a new process)**

The existing dynamic RTS techniques are unsafe in this scenario. For example, Ekstazi would collect dependencies only on classes that are loaded in the JVM (i.e., `TestSnappyCompressorDecompressor.class` and `SnappyCompressor.class`) but it would not collect the dependency on `libhadoop.so` that contains the compiled code for `SnappyCompressor.c`. Similarly, techniques that collect dependencies on executed methods, e.g., TestTube [21], would detect methods written in Java (i.e., `testSnappyNullPointerException`, `compress`, and `compressBytesDirect`) and would miss functions written in C (i.e., function `Java_compressBytesDirect`).

Our proposed technique, RTSLinux, is, by design, oblivious to the specificities of JNI. As such, it collects all dependencies introduced by the JVM loading whatever files it needs to execute JNI calls. RTSLinux only needs to be informed (by a build system or a testing framework) when each test starts and finishes; no changes to the project under test are needed.

### 2.2 A Single Language and Multiple Processes

Figure 2 shows a test that spawns a new Java process; the test is taken from the Ant project [1] (Git SHA: c50b683c), a popular build system for Java.

The `testForkedCapture` test method (declared in `JUnitTaskTest` class) checks the correctness of the JUnit task [4] with "fork" set to `true` (meaning that Ant should execute the test in a newly spawned JVM rather than the same one that Ant is already running in).

In the setup of the test, a build script to be executed by Ant is specified (line 3); the build script contains a target with a JUnit task in it. The test first sets the "fork" option to `true` (line 6) and then executes said target (line 7). The execution continues through the implementation of the build system (line 10) until it eventually spawns a new JVM where the requested JUnit task executes the tests found in the `Printer` class (line 16).

The existing RTS techniques would only collect dependencies within the JVM of `JUnitTaskTest` and would miss dependencies introduced from the spawned processes. For example, Ekstazi would

```
1  // JUnitTaskTest.java
2  public void setUp() {
3      buildRule.configureProject("src/etc/testcases/taskdefs/
           optional/junit.xml");
4  ...}
5  public void testForkedCapture() throws IOException {
6      buildRule.getProject().setProperty("fork", "true");
7      buildRule.executeTarget("capture");
8  ...}
9
10 // BuildFileRule.java → Project.java → Target.java →
11     Task.java → DispatchUtils.java
12
13 <!-- src/etc/testcases/taskdefs/optional/junit.xml -->
14 <target name="capture" depends="setUp">...
15     <junit fork="${fork}">
16         <test name="org.apache.tools.ant...optional.junit.Printer"
17         .../>...
18     </junit>
19 </target>
```

**Figure 2: An example test from the Ant project that spawns a new JVM to execute tests in the Printer class**

collect dependencies on `JUnitTaskTest`, `BuildFileRule`, `Project`, `Task`, `Target`, and `DispatchUtils .class` files, but not `Printer.class` (ultimately failing to select `testForkedCapture` if the `Printer` class is modified, impacting the regression testing safety). RTS techniques which collect fine-grained dependencies (e.g., TestTube [21] or FaultTracer [86]) face the same issue.

RTSLinux, automatically tracks the entire *process tree* of the test (rather than a single process at a time), and collects dependencies on all files accessed by either the root process or any of its children processes. Given the example in Figure 2, therefore, RTSLinux collects the same dependencies as Ekstazi as well as the dependencies introduced by the *spawned process*, i.e., the `Printer.class` file.

# 3 TECHNIQUE

This section describes the details of RTSLinux. We describe the common phases performed by traditional RTS techniques, present the way RTSLinux performs those phases, discuss a mechanism that a user can utilize to alter the behavior of the technique, and present the integration of RTSLinux with the existing testing frameworks and build systems.

RTS techniques typically include three phases: analysis, execution, and collection [37]. The *analysis phase* checks, for each test, if any of the previously collected dependencies have been affected by the recent changes. If a test is not invalidated by the changes, the test is not executed. (Tests for which dependencies have not previously been collected are always selected.) The *execution phase* runs selected tests. The *collection phase* collects dependencies for each test; these dependencies are used in the analysis phase of the subsequent project revision. The execution and collection phases are frequently interleaved, i.e., collection is done during test execution rather than in a separate run.

Figure 3 shows the three phases as performed by RTSLinux. Function `Run` (line 1) takes as input a command to execute and functions defined by the user that can alter RTSLinux's behavior (as discussed later in this section). The given command can be an arbitrary Unix command; in the context of this paper, that command

```
Require: cmd - user command to execute
Require: userFuns - functions defined by the user
1:  function Run(cmd, userFuns)
2:      if IsTestAffected(cmd, userFuns) then
3:          processMap ← EMPTY_MAP
4:          pid ← ExecuteTest(cmd, processMap)
5:          StoreDeps(cmd, pid, userFuns, processMap)
6:      end if
7:  end function
8:  function IsTestAffected(cmd, userFuns)
9:      if HasDependencies(cmd) then
10:         for all proc ∈ SetOfProcesses(cmd) do
11:             for all dep ∈ SetOfDeps(proc) do
12:                 nsum ← userFuns.cksum(proc.type, dep.path)
13:                 if nsum = NONE then
14:                     nsum ← SysCksum(dep.path)
15:                 end if
16:                 if nsum ≠ dep.cksum then
17:                     return TRUE
18:                 end if
19:             end for
20:         end for
21:         return FALSE
22:     end if
23:     return TRUE
24: end function
25: function SystemExecute(syscall, processMap)
26:     if SpawnsNewProcess(syscall) then
27:         processMap +← (syscall.currentProc, syscall.newPid)
28:     else if AccessesFiles(syscall) then
29:         for all dep ∈ syscall.getAccessedFiles() do
30:             SetOfDeps(syscall.currentProc) +← dep
31:         end for
32:     end if
33: end function
34: function StoreDeps(cmd, pid, userFuns, processMap)
35:     processes = ∅
36:     for all proc ∈ {p | ∃pid′ s.t. pid′ = p.pid ∧
37:           pid′ ∈ TransitiveClosure(pid, processMap)} do
38:         processes +← proc
39:         for all dep ∈ SetOfDeps(proc) do
40:             if userFuns.includeIn(dep) then
41:                 dep.cksum ←
42:                     userFuns.cksum(proc.type, dep.path)
43:                 if dep.cksum = NONE then
44:                     dep.cksum ← SysCksum(dep.path)
45:                 end if
46:             end if
47:         end for
48:     end for
49:     SetOfProcesses(cmd) ← processes
50: end function
```

**Figure 3: Analysis, execution, and collection phases as performed by RTSLinux**

will always execute a test. If any dependency of the given command is affected (line 2), then RTSLinux executes the command and stores (in the current working directory) new dependencies for the given process id (line 5). We next describe the details of the three phases.

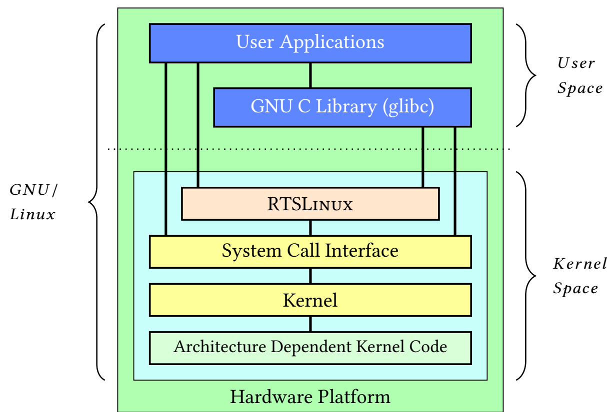


**Figure 4: Integration of RTSLinux in the Linux software stack. RTSLinux wraps system calls that manage processes and files**

## 3.1 Analysis Phase

Analysis phase (Figure 3, lines 8 - 24) checks if the given command/test is affected or not; the utility function `HasDependencies` checks if there are dependencies associated with the given command and the current working directory. As stated earlier, if there are no dependencies associated with the command (e.g., command has not been executed before or its meta-data was removed), the command is always affected (line 23). If dependencies are present, then RTSLinux checks if any dependency of the given command is affected since the latest run (lines 10 - 20). The set of dependencies for a given command includes all dependencies collected (in the previous run) by any process that was spawned by the command.

To check if a dependency file is modified or not (line 16), RTSLinux compares the *checksum* of the current version of the file and the checksum that was computed at the end of the prior execution of the same command. By default, RTSLinux computes the checksum of the entire file or (file names in a) directory.

RTSLinux provides an extension mechanism, which *allows* the user to specify a customized function for computing the checksums (line 12). The custom function accepts two arguments: the type of the process that accesses the dependency and the path to the dependency. Based on the type of the process and the type of the file, the user can adjust the computation of the checksum. For example, if a process parses an XML file, the user may exclude comments from all XML files. In such cases, the command would not be selected if the changes are only in the comments of XML files. Note that computing the checksum is application dependent; RTSLinux provides a function that ignores debug info in .class files if these files are accesses from a JVM. The extension mechanism for the checksum can also be used to implement hierarchical checksum, i.e., a user can first check the size of the file and then check the content only if the size is the same.

## 3.2 Execution Phase

In RTSLinux, collection happens for every executed process. Unlike most existing dynamic RTS techniques, RTSLinux *does not require changes to either sources or binaries* of the project under test to enable collection of dependencies.

Figure 4 shows the integration of RTSLinux in the Linux OS. Specifically, RTSLinux implements wrappers [33] for each system call available in Linux that manages the set of running processes (e.g., fork) or accesses the file system (e.g., open). After starting the execution of the command (line 4 in Figure 3), system calls go through RTSLinux wrappers. Specifically our `SystemExecute` function (line 25 in Figure 3) is invoked from the wrappers. The function checks if the system call is spawning a new process or if it accesses any file or directory on the file system. If the call spawns a new process, RTSLinux updates the mapping of processes to include that the current process is spawning a new process; the map is used later to reconstruct the process tree, which is used to find all transitively spawned processes from the given process id. On the other hand, if the system call accesses a file or a directory, RTSLinux updates the set of dependencies of the current process. Note that this phase only creates the mapping among processes and collects dependencies, but does not compute the checksum of dependencies or associate dependencies with the command being executed. We separated the two phases to enable the user to collect dependencies for any part of the process tree and to entirely ignore dependencies for some commands if necessary.

## 3.3 Collection Phase

Collection phase (Figure 3, lines 34 - 50) computes the set of processes transitively spawned from the given command and computes the checksum of each dependency accessed by at least one process. Initially, RTSLinux finds transitive closure of processes from the process that started the given command (line 37). To compute the transitive closure, RTSLinux uses process map, which was created during the execution. Note that it is necessary to find only processes that are spawned (transitively) from the given command, because there can be many other processes running in parallel, which are not related to the user command. Even more, there could be other processes running other tests in parallel. We discuss several aspects of parallel test execution in Section 6. Computing the checksum of each dependency of each process is similar to the checksum computation that was described in Section 3.1. As before, RTSLinux uses the user defined function (if provided) for computing the checksum. Additionally, in collection phase, RTSLinux provides an extension point where the user can provide a filter to exclude some dependencies, e.g., temp files and directories.

## 3.4 System Integration

We implemented RTSLinux as a loadable kernel module [6] and made only a few modifications to the Linux kernel [48]. We have tested our changes with several versions of the Linux kernel: from version 3.13.11-ckt39 (Apr 2016) to version 4.4.40 (Jan 2017). We had to change only five lines of code in our module to enable it to work across all Linux versions in the tested range. (Specifically, an argument to `do_execve` function was moved to a struct, so we had to adjust one of our wrapper functions.) We note, however, that other changes (e.g., addition of a new system call) could lead to a much higher maintenance cost.

RTSLinux implementation includes all the functions shown in Figure 3 and a number of wrapper functions for the existing system

calls. The user can invoke either the top level function `Run`, or invoke independently `IsTestAffected` and `StoreDeps` functions.

The `Run` function provides a mechanism to collect dependencies for any command (e.g., an arbitrary bash script) and skip the execution of the same command in the future if no dependency has changed [23]. (The `Run` is similar to Fabricate [32] and Memoize [59] build systems, except that RTSLinux runs as part of the OS; we discuss other differences in Section 7.) As an example, to enable RTSLinux when executing tests with Maven [56], instead of running `mvn test` the user should run `rtslinux mvn test`. The `rtslinux` command implements the `Run` function, thus it executes the command and collects dependencies only if the set of dependencies (collected in the previous run) has been affected. However, collecting dependencies for a test command that executes *all tests*, is likely to be imprecise because any change to *any dependency of any test* would trigger the execution of *all tests* in the future.

To improve precision, the user can run one test per process; most popular build systems (e.g., [1, 39, 56]) provide options to spawn a new process for each test.

`IsTestAffected` and `StoreDeps` can be used by developers of build systems to check if a test should be executed before starting a test and saving dependencies for the test after it was executed. We integrated RTSLinux with Maven.

**An alternative implementation**: In addition to RTSLinux, we implemented our technique in another tool, called RTSFab, which works in *user space* (i.e., runs outside the operating system's kernel). Our motivation was to explore and compare the overheads of various approaches for collecting test dependencies. RTSFab is implemented on top of Fabricate [32], a build system with dynamic dependencies that uses `strace` to collect accessed files [10].

# 4 EVALUATION

We assess the usability of RTSLinux by answering the following research questions:

**RQ1:** How effective is RTSLinux, i.e., what is the reduction in testing time and the number of executed tests?

**RQ2:** What are the benefits/drawbacks of dependency detection across JVM boundaries (as implemented in RTSLinux) compared to a single-JVM RTS (as implemented in Ekstazi, a recently developed RTS tool for Java). Specifically:

**RQ2.1** (Efficiency): Does RTSLinux achieve as much reduction in total testing time and number of executed tests?

**RQ2.2** (*Safety*): How many more dependencies are discovered by RTSLinux?

**RQ3:** What is the overhead of RTSFab (a naive implementation of our technique running in user space) compared to RTSLinux?

The main objective of our evaluation is to show that (1) RTSLinux is as effective as the state-of-the art, and (2) detecting file-based dependencies at the system level is beneficial. Additionally, by comparing RTSLinux to RTSFab, we provide an initial assessment of the importance of implementing our technique in kernel space instead of in user space. Although RTSFab may not be the most efficient possible implementation running in user space, we believe it provides a good starting point for seeking an implementation

```
Require: project - a project under study
Require: tool - an RTS tool under study
 1: function ExperimentProcedure(project, tool)
 2:     Clone(project.url)
 3:     Checkout(project.sha)
 4:     for all ρ ∈ Last20Revisions(project) do
 5:         project.sha ← ρ
 6:         Checkout(project.sha)
 7:         if Build(project) == FAILED then
 8:             continue
 9:         end if
10:         test_results ← Test(project)
11:         StoreAvailable(project, test_results)
12:         Integrate(project, tool)
13:         selected ← Select(project)
14:         test_results, deps ← Test(project, selected)
15:         StoreSelected(project, tool, test_results, deps)
16:     end for
17: end function
```

**Figure 5: Experiment procedure**

running in user space that is as efficient and as easy to integrate with the existing build systems as RTSLinux is.

We first describe the setup of our study (Section 4.1) and then answer the research questions (Section 4.2). Section 4.3 includes additional case studies, including an experiment in using RTSLinux with a Python project.

## 4.1 Study Setup

*4.1.1 Projects.* Table 1 lists the projects used in this evaluation, sorted by test execution time. This list includes projects from recent studies on regression testing [18, 20, 37, 41], as well as a number of new projects containing tests that escape the JVM by spawning processes and/or making native calls. Although not all of the projects used in previous studies require RTSLinux's cross-JVM capabilities, they still serve as fair benchmarks for evaluating RTSLinux on its own merit (RQ1) and comparing it to Ekstazi in terms of efficiency (RQ2.1). We required that each project is available on GitHub and builds with Maven, which simplified our experiment infrastructure.

Columns in Table 1 designate project name, GitHub repository URL, latest revision (SHA) used in the study, number of lines of code (LOC) (as measured with `sloccount`), number of Maven modules, total number of files, total number of test classes, test execution time for all test classes for a single project revision, and the way in which (at least some) tests escape the JVM they run in (Processes - spawns subprocesses, Native Calls - has native calls, Files - accesses external files, e.g., txt files); for projects that escape the JVM via both subprocesses and JNI, we mark them only as "Processes" in the table, because that is the more interesting case from RTSLinux's perspective. For each project, we evaluated our selection technique over 20 revisions (such that the latest of these revisions is the one reported in the SHA column); we report the averages in the second and third to last columns in Table 1.

The last two rows show the total ($\sum$) and average (Avg.) values for each column (if appropriate). In summary, our evaluation spans across 420 revisions of 21 projects, totaling 2,050,791 lines of code.

*4.1.2 Experiment Procedure.* Figure 5 provides the procedure that we used to collect data for the analysis for a single project.

**Table 1: Projects Used in the Evaluation**

| Project | URL [https://github.com/] | SHA | LOC | #Maven modules | #Files | #Test classes | Test time [s] | Escape method |
|---|---|---|---|---|---|---|---|---|
| la4j | vkostyukov/la4j.git | 358be77e | 13390 | 1 | 147 | 22.85 | 14.68 | N/A |
| ScribeJava | fernandezpablo85/scribe-java.git | 5175a416 | 7613 | 5 | 219 | 20.00 | 14.85 | N/A |
| Bukkit | Bukkit/Bukkit.git | f210234e | 32555 | 1 | 762 | 38.00 | 21.66 | N/A |
| ZT-Exec | zeroturnaround/zt-exec | 36654400 | 2938 | 1 | 104 | 18.45 | 25.04 | Processes |
| Crypto | apache/commons-crypto.git | dc1769ed | 5079 | 1 | 140 | 24.00 | 27.94 | Native Calls |
| Retrofit | square/retrofit | ec0635c6 | 12331 | 16 | 202 | 30.75 | 28.74 | Files |
| Codec | apache/commons-codec.git | 535bd812 | 17625 | 1 | 299 | 48.00 | 31.72 | Files |
| Vectorz | mikera/vectorz.git | 425109e2 | 52096 | 1 | 414 | 70.50 | 38.98 | N/A |
| Lang | apache/commons-lang.git | 17a6d163 | 69014 | 1 | 381 | 133.50 | 41.21 | Files |
| Net | apache/commons-net.git | 4450add7 | 26928 | 1 | 315 | 42.00 | 65.13 | Files |
| Config | apache/commons-configuration.git | 8dddebf1 | 64341 | 1 | 642 | 162.30 | 66.05 | Files |
| IO | apache/commons-io.git | e8c1f057 | 27186 | 1 | 302 | 91.00 | 89.10 | Files |
| OkHttp | square/okhttp | d854e6d5 | 48783 | 18 | 344 | 59.40 | 101.72 | Files |
| ClosureC | google/closure-compiler.git | 283d8161 | 284131 | 7 | 1548 | 309.30 | 190.41 | Native Calls |
| Dropwizard | dropwizard/dropwizard.git | 1e40fef4 | 37914 | 34 | 969 | 232.00 | 328.84 | Processes |
| CloudStack | apache/cloudstack.git | 56a35265 | 572503 | 104 | 7585 | 292.00 | 335.42 | Processes |
| Tika | apache/tika | 9cf82589 | 96220 | 15 | 1936 | 227.65 | 370.08 | Processes |
| Math | apache/commons-math.git | 471e6b07 | 174832 | 1 | 1501 | 431.00 | 376.46 | Files |
| Guava | google/guava.git | 061da3b3 | 244083 | 5 | 1737 | 401.00 | 424.66 | Files |
| Metron | apache/incubator-metron | 29646550 | 57720 | 28 | 1507 | 145.00 | 462.28 | Processes |
| Activiti | activiti/activiti.git | b2eba94b | 203509 | 7 | 5523 | 312.35 | 879.99 | Processes |
| ∑ | N/A | N/A | 2050791 | 250 | 26577 | 3111.05 | 3934.96 | N/A |
| Avg. | N/A | N/A | 97656.71 | 11.90 | 1265.57 | 148.14 | 187.37 | N/A |

In the first step (line 2), we clone the project and then (line 3) checkout the latest revision used in the study (SHA column in Table 1). Next, we iterate over the last 20 project revisions (lines 4 - 16), starting from the oldest revision and moving towards newer revisions. We consider only project revisions that are on the master branch, because many projects run regression tests only against those revisions. In each iteration of the loop, we build the project and skip the revision if the build fails. If the build is successful, we execute the available tests and save the results. To force each test to run in a separate JVM (which is a common practice [18]), we run `mvn test -DforkCount=1 -DreuseForks=false`. Note that `mvn test` runs several build phases (including the compilation phase) prior to running the tests. We will use $\mathcal{E}^{available}$ to denote this *end-to-end* test execution time and $\mathcal{N}^{available}$ to denote the number of available tests.

In the next step, the procedure (line 12) integrates RTSLinux by enabling our module in the Linux kernel and including the RTSLinux Maven plugin into the project under study. The tests are then selected based on the dependencies collected in the previous run and those tests are executed (line 14). Finally, the procedure stores the results and newly collected dependencies for the executed tests (line 15). We will use $\mathcal{E}^{sel}$ to denote *end-to-end* time to select tests, execute those tests, and collect dependencies; we will use $\mathcal{N}^{sel}$ to denote the number of selected tests.

Using the collected data, we compute *test selection ratio* as $\mathcal{S} = \mathcal{N}^{sel} / \mathcal{N}^{available} * 100$, and *savings in terms of end-to-end time* as $\mathcal{T} = \mathcal{E}^{sel} / \mathcal{E}^{available} * 100$. Savings in terms of end-to-end time (compared to RetestAll) is the key metric for measuring the benefits of an RTS technique [37]. To be consistent with prior studies, we also report test selection ratio; the saving in terms of the number of tests is a metric independent of the machine used for running the experiments.

*4.1.3 Execution Platform.* We run all experiments on a 4-core Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz with 16GB of RAM, running (our version of) Ubuntu 14.04 LTS. Similar to several recent studies [37, 41, 72], we used multiple versions of Java (Oracle JDK 7u72 and 8u60), which was necessary because several projects (e.g., `Lang`) do not work with newer Java versions.

## 4.2 Answers to Research Questions

*4.2.1 RQ1: How Effective is RTSLinux.* Table 2 shows the main results for RTSLinux. Columns 2-5 are explained in Section 4.1.2; Column 6 denotes the total number of dependencies used by tests (we counted dependencies that are in the project under study, the local Maven cache, or executable files used by tests). The last two rows show the summary values (where appropriate) and average values computed over all projects. We discuss the right-hand part (Ekstazi) of the table in the next section.

*Our results show that RTSLinux reduces test execution time for all projects.* In the best case (for the `Net` project) time to execute tests is decreased to only 13.52% (of RetestAll time). In the worst case (for the `ClosureC` project) time to execute tests is decreased to 85.12%. On average, across all projects, test execution time is decreased to 47.17%. Similarly, RTSLinux leads to substantial reduction in terms of the number of executed tests. In the best case (for the `Config` project), RTSLinux selects only 7.94% of available tests. In the worst case (for the `Vectorz` project), RTSLinux selects 64.32% of available tests. We can observe that there are significant

**Table 2: Test Execution Time and Test Selection Ratio for RTSLinux and Ekstazi**

| Project | RTSLinux | | | | | Ekstazi | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time | | Tests | | Deps | Time | | Tests | | Deps | |
| | $\mathcal{E}^{sel}$[s] | $\mathcal{T}$[%] | $\mathcal{N}^{sel}$ | $\mathcal{S}$[%] | # | $\mathcal{E}^{sel}$[s] | $\mathcal{T}$[%] | $\mathcal{N}^{sel}$ | $\mathcal{S}$[%] | # | [%] |
| la4j | 10.82 | 73.76 | 12.70 | 55.57 | 861 | 10.56 | 71.96 | 12.70 | 55.57 | 298 | 34.61 |
| ScribeJava | 6.79 | 45.72 | 1.65 | 8.25 | 764 | 6.51 | 43.82 | 1.65 | 8.25 | 83 | 10.86 |
| Bukkit | 6.25 | 28.88 | 3.20 | 8.42 | 1723 | 6.01 | 27.75 | 3.20 | 8.42 | 457 | 26.52 |
| ZT-Exec | 14.61 | 58.35 | 7.95 | 43.08 | 1046 | 14.18 | 56.63 | 7.95 | 43.08 | 81 | 7.74 |
| Crypto | 11.93 | 42.69 | 6.05 | 25.20 | 752 | 10.43 | 37.32 | 4.65 | 19.37 | 79 | 10.50 |
| Retrofit | 14.63 | 50.92 | 8.85 | 28.78 | 15082 | 14.56 | 50.66 | 8.85 | 28.78 | 559 | 3.70 |
| Codec | 7.27 | 22.92 | 4.25 | 8.85 | 1363 | 7.00 | 22.07 | 4.25 | 8.85 | 144 | 10.56 |
| Vectorz | 31.66 | 81.22 | 45.35 | 64.32 | 1257 | 31.05 | 79.64 | 45.35 | 64.32 | 379 | 30.15 |
| Lang | 15.14 | 36.74 | 17.35 | 12.99 | 1757 | 14.84 | 36.01 | 17.35 | 12.99 | 604 | 34.37 |
| Net | 8.80 | 13.52 | 3.45 | 8.21 | 773 | 8.44 | 12.96 | 3.45 | 8.21 | 196 | 25.35 |
| Config | 17.22 | 26.07 | 12.90 | 7.94 | 5281 | 15.98 | 24.20 | 12.90 | 7.94 | 689 | 13.04 |
| IO | 33.06 | 37.11 | 18.10 | 19.89 | 1335 | 25.48 | 28.60 | 12.25 | 13.46 | 286 | 21.42 |
| OkHttp | 79.08 | 77.74 | 25.70 | 43.26 | 10745 | 77.63 | 76.31 | 25.25 | 42.50 | 686 | 6.38 |
| ClosureC | 162.09 | 85.12 | 164.80 | 53.28 | 7075 | 153.75 | 80.74 | 164.80 | 53.28 | 2416 | 34.14 |
| Dropwizard | 155.04 | 47.14 | 44.60 | 19.22 | 318493 | 109.43 | 33.27 | 41.35 | 17.82 | 1102 | 0.34 |
| CloudStack | 270.55 | 80.65 | 66.40 | 22.73 | 189878 | 221.83 | 66.13 | 60.70 | 20.78 | 3758 | 1.97 |
| Tika | 153.10 | 41.36 | 52.35 | 22.99 | 71404 | 104.24 | 28.16 | 40.25 | 17.68 | 1300 | 1.82 |
| Math | 114.02 | 30.28 | 101.75 | 23.60 | 4732 | 113.06 | 30.03 | 101.75 | 23.60 | 1938 | 40.95 |
| Guava | 86.17 | 20.29 | 57.55 | 14.35 | 13771 | 83.39 | 19.63 | 56.70 | 14.13 | 5145 | 37.36 |
| Metron | 159.86 | 34.58 | 20.70 | 14.27 | 206994 | 79.53 | 17.20 | 13.80 | 9.51 | 1138 | 0.54 |
| Activiti | 489.54 | 55.63 | 116.50 | 37.29 | 41418 | 440.86 | 50.09 | 116.50 | 37.29 | 1873 | 4.52 |
| $\sum$ | 1847.63 | N/A | 792.15 | N/A | 896504 | 1548.76 | N/A | 755.65 | N/A | 23211 | N/A |
| Avg. | 87.98 | 47.17 | 37.72 | 25.83 | 42690.66 | 73.75 | 42.53 | 35.98 | 24.56 | 1105.28 | 16.99 |

differences in reduction in terms of the number of tests and time. Recall (Section 4.1) that we measure end-to-end time for the entire build (as developers would do when running their tests) rather than measuring only test execution time; therefore, we tend to observe smaller savings for projects where build phases (e.g., compilation) take longer than test execution.

*4.2.2 RQ2.1: How Does RTSLinux Compare to Ekstazi in Terms of Efficiency.* Table 2 (right) shows the results for Ekstazi, including time taken to execute selected tests (Column 7), time to run selected tests normalized by RetestAll (Column 8), the average number of selected tests (Column 9), test selection ratio (Column 10), number of dependencies (Column 11), and number of dependencies normalized by RTSLinux dependencies (Column 12). The last two rows show the total and average values.

In summary, test selection ratio is 25.83% and 24.56%, and execution time is decreased to 47.17% and 42.53% for RTSLinux and Ekstazi, respectively. It is expected that RTSLinux takes longer to execute tests, because it captures more dependencies than Ekstazi (as discussed in the next section), which may lead to more tests being affected per run and higher cost for computing checksums.

Although test selection ratio is the same for most of the projects, we can observe differences in several cases, e.g., `IO` (19.89% vs. 13.46%). Such a difference can happen for two reasons: (1) tests create directories and/or files that are not removed when test execution finishes, and (2) Ekstazi misses to collect some files that are modified between revisions. We illustrate these two cases using the `IO` project. Many tests in the `IO` project create the `test` directory in the root of the project, but they do not remove the directory upon completion. Consider a test `t` that creates the `test` directory. When the test finishes, RTSLinux computes the checksum of the `test` directory (which is non-empty at this point). At the next project revision (under the assumption that we are running tests on a continuous integration service (CIS) [11, 46], e.g., Travis CI, which always does a clean build), RTSLinux will compute the checksum for the `test` directory and find that the checksum is different from before, thus RTSLinux will select `t` every time for the execution. (Note that the same problem can happen even if developers do not use CIS, but multiple tests use the same test directory that is not cleaned [18].) This problem happens for several other projects including `Config` and `ClosureC`; observe that the test selection ratio in the table is the same for these two projects because we automatically insert code to cleanup project repository prior to computing the checksum. As mentioned before, another reason for the difference in test selection ratio is the difference in the set of dependencies. We observed that several tests in `IO` (e.g., `DirectoryFileComparatorTest`) traverse all files from the root of the project; whenever any of these files change RTSLinux selects tests for the execution, while Ekstazi misses to select these tests.

*4.2.3 RQ2.2: How Does RTSLinux Compare to Ekstazi in Terms of Safety.* We find that Ekstazi collects only 16.99% of dependencies collected by RTSLinux (see the last row in the last column in Table 2). Therefore, Ekstazi is less safe than RTSLinux. Despite these differences in the set of dependencies, the reduction in test execution time by RTSLinux and Ekstazi is similar, which demonstrates that RTSLinux is efficient. We observed, for RTSLinux, that projects with many Maven modules have high number of dependencies; the

**Table 3: Overhead Introduced by RTSLinux, Ekstazi, and RTSFab for the First Revision Compared to RetestAll**

| Project | RetestAll [s] | RTSLinux [%] | Ekstazi [%] | RTSFab [%] |
|---|---|---|---|---|
| la4j | 17.15 | 14.16 | 14.95 | 44.29 |
| Bukkit | 21.41 | 15.53 | 18.17 | 44.74 |
| Codec | 25.18 | 19.62 | 22.26 | 59.73 |
| Vectorz | 39.57 | 17.84 | 15.69 | 50.64 |
| Lang | 40.44 | 26.69 | 36.20 | 66.26 |
| Net | 65.42 | 5.43 | 4.83 | 88.62 |
| Config | 66.49 | 25.61 | 25.07 | 82.78 |
| IO | 90.19 | 8.02 | 7.66 | 23.87 |
| Math | 374.82 | 10.20 | 12.60 | 39.96 |
| ZT-Exec | 23.49 | 11.00 | 10.26 | 137.84 |
| Crypto | 27.90 | 9.84 | 11.21 | 34.13 |
| Avg. | 72.00 | 14.90 | 16.26 | 61.16 |

number of modules is reported in Table 1. This high number of dependencies happens because Java searches for a class file in all modules on the classpath prior to searching in third party libraries. Note that RTSLinux collects even files that are non-existent (and assigns a special flag to them), because adding such files can affect test execution.

*4.2.4 RQ3: What is the Overhead of RTSFab (our Technique Implemented in User Space) Compared to RTSLinux.* Table 3 shows, for each project, time to run all tests (repeated values from Table 1), and overhead of RTSLinux, Ekstazi, and RTSFab when *all tests* are run (which happens for the first revision). Table 3 includes only single module projects, because RTSFab currently supports only those projects. On average, RTSLinux, Ekstazi, and RTSFab introduce 14.90%, 16.26%, and 61.16% overhead, respectively. We can conclude that RTSFab is significantly less efficient than RTSLinux. A more efficient way to implement our technique in user space could still exist (e.g., using FUSE); this question is left for future work.

Additionally, we can conclude that the collection phase of RTSLinux introduces lower overhead than that of Ekstazi, but it takes more time to compute checksums (due to larger number of collected dependencies). The latter conclusion is supported by Tables 2 and 3. We can see that for the first project revision (when computation of checksums is done only once after tests were executed) RTSLinux introduces lower overhead. However on average across 20 revisions, when checksums are computed both during the analysis phase and collection phase, RTSLinux introduces slightly higher overhead.

## 4.3 Case Studies

We performed several case studies to further evaluate RTSLinux and test its correctness. Specifically, we checked if RTSLinux gives the expected results for our motivating examples (Section 2), tried several interesting tests from various projects, and evaluated the benefits of RTSLinux when applied to a Python project.

*4.3.1 Multiple Languages and a Single Process.* Section 2.1 introduced a test, from the Hadoop project, which is written in multiple languages. Recall that the test is executed in a single process. Our goal is to confirm that RTSLinux collects necessary dependencies. (We did not use Hadoop in our experiments in Section 4.2 due to long RetestAll time and limited resources.) In our study, we executed only the `testSnappyNullPointerException` test described in Section 2.1 and we collected dependencies with RTSLinux and Ekstazi. RTSLinux collected 870 dependencies (including `libhadoop.so`, which contains the compiled C code) while Ekstazi collected only 99 dependencies (not including `libhadoop.so`).

*4.3.2 A Single Language and Multiple Processes.* Section 2.2 introduced a test from the Ant project that spawns subprocesses. (We did not include Ant in our benchmark projects in Section 4.2 because it does not build with Maven.) As for Hadoop, we executed only the test method of interest (`testForkedCapture`); as expected, RTSLinux collected `Printer.class` (among 617 dependencies) while Ekstazi didn't (among 218 dependencies).

*4.3.3 Additional Cases.* We have also checked correctness of RTSLinux on a few hand-picked interesting tests that either use JNI or spawn subprocesses. Table 4 shows these tests and compares number of dependencies collected by RTSLinux (#Deps) and percent of those dependencies collected by Ekstazi (Deps [%]). We manually confirmed that expected dependencies are collected by RTSLinux. For example, `JavaCPP` spawns a subprocess that runs a g++ compiler (and depends on .h files) and `PySonar2` spawns a process that runs Python code (and depends on .py files).

*4.3.4 A Python Project.* Being implemented at the system level and not tied to Java at all, RTSLinux can be applied to a project written entirely in a non-JVM language. Although this paper primarily focuses on predominantly Java projects, here we discuss our experience of applying RTSLinux to `pendulum` [7], the most popular Python project on GitHub (based on the number of stars).

The `pendulum` project comes with 514 test methods and includes a custom `py.test` command used to run them. Because tests for this project take negligible time, we measured only test selection ratio. We modified `py.test` to integrate RTSLinux; next, we followed the same procedure as for other projects (Figure 5). On average, RTSLinux reduced the number of executed tests to 61.45%. In terms of the number of selected tests, however, RTSLinux selected most of them in 12 out of 20 revisions. The main reason for such high selection rate is the small size of the project, so many tests end up importing most of the source `.py` files (unlike in Java, the import statement in Python evaluates the target `.py` file, and, thus, creates a dependency even if nothing from it gets used by the test).

Our motivation for including a Python project was to show that our technique readily generalizes to other languages; we do not mean, however, to imply that the empirical evaluation results from Section 4.2 do too. To make such a claim, a much larger collection of non-Java projects would have to be included. Further research is also necessary to explore ways of improving precision of RTSLinux for Python (and other languages).

# 5 THREATS TO VALIDITY

**External**: The projects used in our study may not be representative. To mitigate this threat, we used popular open-source projects that vary in size, number of authors, number of revisions, number of tests, application domain, and ways their tests escape from JVM. Furthermore, several projects used in our study were used in recent work on regression testing [37].

**Table 4: Case Studies with Native Calls and Sub-Processes**

| Project | URL [https://github.com/] | SHA | Test name | Escape method | RTSLinux #Deps | Ekstazi Deps [%] |
|---|---|---|---|---|---|---|
| Zeppelin | apache/zeppelin | 63294785 | org.apache.zeppelin.integration.AuthenticationIT | Processes | 775 | 1.03 |
| Ratis | apache/incubator-ratis | 7e71a2e0 | org.apache.ratis.server.storage.TestRaftStorage | Native Calls | 1294 | 2.16 |
| JavaCPP | bytedeco/javacpp | b41028b9 | org.bytedeco.javacpp.AdapterTest | Processes | 449 | 14.47 |
| PySonar2 | yinwang0/pysonar2.git | dc6d8f10 | org.yinwang.pysonar.TestRefs | Processes | 784 | 20.28 |

We performed experiments with projects mostly written in Java (although RTSLinux is, in theory, more broadly applicable). Our main motivation for this work were projects that escape from JVM. In the future, we plan to evaluate RTSLinux and develop necessary extensions for projects written in other languages.

We performed our experiments on 20 revisions per project. The results could differ if we chose different number of revisions or another time frame. For projects used in a previous evaluation of Ekstazi [37], we used the same revisions as reported in that study. For other projects, added in this study, we used the latest revisions available at the time of our experiments.

The overhead introduced by the collection phase may differ for other Linux kernel versions. Although we tested RTSLinux with several Linux versions, we have not measured the time for each version. Considering that the version 4 was introduced only to avoid high version numbers [5], we do not expect that our results would differ on a few latest releases of the kernel.

**Internal**: Implementation of RTSLinux may contain bugs that may impact our conclusions. To mitigate this threat, we wrote a number of tests, manually inspected the output of several examples, and compared the outputs of Ekstazi and RTSLinux. We also compared RTSLinux and RTSFab, which should always have the same test selection ratio.

**Construct**: We compared RTSLinux only with Ekstazi, although many other RTS techniques have been proposed in the past (Section 7). We justify our decision with two reasons. First, other existing RTS techniques collect fine-grained dependencies (e.g., methods, statements, basic blocks, elements of the extended control-flow graphs, etc.) and they are less safe than Ekstazi. Ekstazi also compares favorably with the existing RTS techniques in terms of end-to-end regression testing time [37]. Second, to the best of our knowledge, Ekstazi is the only publicly available RTS tool (for Java) at the moment.

## 6 DISCUSSION

**Dependent tests**: Some test suites may have order dependencies among tests, e.g., if test `t1` executes before test `t2`, then `t2` passes, otherwise it fails. Therefore, if RTSLinux selects only test `t2` for the execution, the test would fail if the test `t1` is not selected. RTSLinux, as other RTS tools, does not reason about order dependencies among tests. However, because RTSLinux requires that each test (or group of tests specified by a developer) executes in a separate process, there cannot be a problem due to order dependencies on values in the main memory. Still, there could be a problem if tests share state on the disk. We believe that RTSLinux can be a base for a framework for detecting such dependencies. Using RTSLinux we were able to detect bugs (when a test does not clean the state on disk) in `IO`, `Config`, and `ClosureC`; our patches were recently accepted by `IO` and `Config` developers [24, 25].

**Flaky tests**: Flaky tests are tests that non-deterministically pass and fail for the same project revision [30, 54, 60, 62, 80, 87]. There are a number of sources of non-determinism, including GUI events, networking, concurrency, etc. Similar to the existing RTS techniques, RTSLinux collects dependencies only for one possible execution; if a test is not selected, then the previous execution trace is still feasible. Note that this is consistent with existing static RTS techniques that are used in practice [31, 79].

**Parallel execution**: RTSLinux supports parallel test execution by tracking dependencies for each user's command separate and monitoring the accessed files/directories by (transitively) spawned processes. Note however that RTSLinux does not support distributed tests (i.e., tests that communicate over network), but only tests that are executed on a single machine.

**One test per JVM**: Although running one test (or groups of dependent tests) per JVM can introduce overhead, it is still a common approach practiced in industry [18]. As a result, RTSLinux fits well in the common development practices and reduces the testing cost by skipping many tests after code changes.

**Detecting files loaded by JVM in Ekstazi**: Ekstazi has a hidden option to collect files loaded by JVM, which is untested and disabled by default. In our experiments, hence, we used the default (stable) Ekstazi configuration. In principle, even when collecting all files loaded by JVM, Ekstazi still cannot detect file accesses made from spawned processes and from native code, making it strictly less safe than RTSLinux.

## 7 RELATED WORK

**Regression test selection**: There has been a lot of work, in the last three decades, on regression testing [27, 44, 85] and on regression test selection [19, 28]. Prior work on dynamic RTS techniques mostly explored fine-grained dependencies, including dependencies on functions/methods (e.g., [21]), statements (e.g., [71]), basic blocks (e.g., [77]), and elements of (extended) control flow graphs [69, 86]. Recent work introduced Ekstazi [37] that collects dynamic file dependencies, which we discussed throughout this paper. Unlike the existing techniques, RTSLinux supports tests that are written in multiple languages and spawn multiple processes.

Kung et al. [49] introduced the class firewall, a technique that (statically) identifies modified set of classes for two project revisions. Skoglund and Runeson proposed an RTS technique based on the class firewall [73] and later improved the precision of their technique [74]. Orso et al. [67] combined the class firewall (static

technique) and dangerous edge (dynamic technique) to increase safety and improve precision. Recent work [51] compared static and dynamic RTS and showed that static techniques tend to be unsafe and imprecise.

Several researchers proposed RTS techniques that collect dependencies on external resources [42, 43, 64, 84]. Haraty et al. [42] and Daou [43] introduced regression testing techniques for database applications. Willmor and Embury [84] presented a new definition of safety that takes into account the interaction of the program with a database state. Nanda et al. [64] recently proposed an RTS technique that collects dependencies on non-code elements, such as configuration files and databases. RTSLinux, collects not only dependencies on configuration files and databases but on any file accessed by any process (transitively) spawned by the test.

**Tracing tools**: There are several tools for tracing system calls in the Linux kernel. The most popular tool for tracing system calls is the `strace` tool [10]. Internally, `strace` uses `ptrace` system call that stops the traced process(es) for each call and introduces observable overhead when many system calls are made by the traced process(es) [88]. For example, the execution of `du -sh ~/` is hundreds of times faster than the execution of `strace du -sh ~/`. Another popular tool available in Linux is `perf trace`. Although this tool has much lower overhead than the `strace` tool, `perf trace` does not provide a way to extract the file names in human readable format [8]. Other third-party tools are available, including `systemtap`, `LTTng`, and `ktap`. These tools are more generic than RTSLinux and require the use of domain specific languages to specify calls to be traced. Additionally, our attempt to use `systemtap` was unsuccessful due to incompatibility with recent versions of the Linux kernel.

**Build systems and memoization**: Many build systems (including Ant [1], Make [75], Maven [56], and Gradle [39]) support incremental execution of a build target, but the incremental computation is commonly based on statically computed dependencies, which makes them unsafe. Modern cloud-based build systems [16, 23] improve safety of incremental builds by keeping the explicit list of fine-grained dependencies, but require substantial effort by the users [38] and are rather imprecise [20, 61, 81]. Memoize [59] and Fabricate [32] collect dynamic dependencies on executed files by using the `strace` command; these systems do not collect dependencies on accessed files that do not exist. SCons [9] and Vesta [45] capture accesses to files even if the files do not exist; however SCons does not support an arbitrary language by default and Vesta requires that used files are under revision control. Tup [12] collects dynamic file dependencies via FUSE; therefore, Tup currently cannot collect separate set of dependencies for tests that are running in parallel processes. Pluto is a build system for Java that dynamically collects file dependencies and uses semantic hashing. Regarding memoization, Guo and Engler proposed IncPy [40] that memoizes calls to functions even if functions access files. Similar to existing RTS techniques, IncPy is language-specific.

RTSLinux was inspired by both the existing build systems and memoization. However, RTSLinux is complementary to build systems and can be integrated with any build system to enable RTS, as long as the build system notifies RTSLinux when each test starts and finishes; these notification can be easily added via plugins that are supported by many existing build systems. RTSLinux also may reduce the need for build system migration that can be nontrivial [38, 58, 65, 78].

**Provenance-aware systems**: Provenance-aware systems provide meta-data that describe the history of various objects in a system. Initial work on provenance-aware systems was language-specific (e.g., [13]). Recent work explored support for multilingual projects via libraries [55] (that require manual annotations) and OS logging [63, 76]. Muniswamy-Reddy et al. [63] proposed a technique that collects meta-data on various abstraction levels. Bates et al. [15] presented a whole-system provenance-aware technique that collects meta-data for the entire system with negligible overhead, and Gehani and Tariq introduced support for provenance auditing in distributed environments [34]. Lee et al. [50] presented an approach that avoids dependency explosion. RTSLinux (and any other RTS technique) and the provenance-aware systems have different goals, which is reflected in the type of collected meta-data and the way the meta-data is used. For example, an RTS technique needs to know when each test starts and finishes.

**Continuous integration services**: Recent work by Vasilescu et al. [83] and Hilton et al. [46] showed that continuous integration services (CISs), such as Travis CI [11], are widely used and improve the productivity of project teams. Currently, more than 300K projects use Travis CI [46], which is only one out of more than 25 publicly available CISs. CISs are used by projects written in different (combination of) programming languages and build languages. RTSLinux could easily be integrated with a number of CISs that run Linux based VMs, which would lead to reduction in regression testing cost for a number of projects.

# 8 CONCLUSION

We presented a novel regression test selection technique, dubbed RTSLinux. The key novelty is that RTSLinux supports tests that escape JVM (e.g., spawn multiple subprocesses). For each test, RTSLinux collects all accessed files by the process running the test and all spawned subprocesses; the overhead of collection is reduced by implementing RTSLinux at the system level (as a loadable kernel module). RTSLinux provides extension mechanism for smooth integration with build systems and testing frameworks. Our results show that RTSLinux substantially reduces testing time compared to RetestAll. Additionally, our experiments showed that RTSLinux achieves similar savings for Java projects as Ekstazi (a regression test selection tool for JVM languages) while improving the safety. Although more research is needed to evaluate RTSLinux for various languages and improve its precision by combining it with language-specific techniques, we believe that current savings achieved by RTSLinux can make a significant difference for a large number of multilingual projects and any continuous integration service.

# ACKNOWLEDGMENTS

We thank the anonymous reviewers for help in improving this paper; Oguz Demir, Nima Dini, Darko Marinov, and John Micco for their feedback on this work; and Casen Hunger, Nick Kelly, and Nghia (Tommy) Huynh for providing us access to their hardware. This research was partially supported by the US National Science Foundation under Grants Nos. CCF-1566363 and CCF-1652517, and by a Google Faculty Research Award.

## REFERENCES


[1] Ant Home Page. http://ant.apache.org.
[2] Apache Hadoop Home Page. http://hadoop.apache.org.
[3] Build in the Cloud. http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html.
[4] JUnit Task. https://ant.apache.org/manual/Tasks/junit.html.
[5] Linus Torvalds - Moving to Linux 4.0. https://plus.google.com/+LinusTorvalds/posts/jmtzzLiiejc.
[6] Linux Loadable Kernel Module HOWTO. http://tldp.org/HOWTO/Module-HOWTO.
[7] pendulum - Python datetimes made easy. https://github.com/sdispater/pendulum.
[8] perf trace & vfs_getname. http://www.spinics.net/lists/linux-perf-users/msg02975.html.
[9] SCons. http://www.scons.org.
[10] strace - trace system calls and signals. http://linux.die.net/man/1/strace.
[11] Travis CI - Test and Deploy Your Code with Confidence. https://travis-ci.org.
[12] Tup. http://gittup.org/tup.
[13] Elaine Angelino, Daniel Yamins, and Margo I. Seltzer. 2010. StarFlow: A Script-Centric Data Analysis Environment. In *International Provenance and Annotation Workshop*. 236–250.
[14] Thomas Ball. 1998. On the Limit of Control Flow Analysis for Regression Test Selection. In *International Symposium on Software Testing and Analysis*. 134–142.
[15] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. 2015. Trustworthy Whole-system Provenance for the Linux Kernel. In *USENIX Conference on Security Symposium*. 319–334.
[16] Bazel. http://bazel.io/.
[17] Boris Beizer. 1990. *Software Testing Techniques (2nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA.
[18] Jonathan Bell and Gail E. Kaiser. 2014. Unit test virtualization with VMVM. In *International Conference on Software Engineering*. 550–561.
[19] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. 2011. Regression Test Selection Techniques: A Survey. *Informatica (Slovenia)* 35, 3 (2011), 289–321.
[20] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build System with Lazy Retrieval for Java Projects. In *International Symposium on Foundations of Software Engineering*. 643–654.
[21] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. 1994. TestTube: A System for Selective Regression Testing. In *International Conference on Software Engineering*. 211–220.
[22] Pavan Kumar Chittimalli and Mary Jean Harrold. 2007. Re-computing Coverage Information to Assist Regression Testing. In *International Conference on Software Maintenance*. 164–173.
[23] Maria Christakis, K. Rustan M. Leino, and Wolfram Schulte. 2014. Formalizing and Verifying a Modern Build Language. In *International Symposium on Formal Methods*. 643–657.
[24] Cleaning directories after each test run, to prevent repository pollution. https://github.com/apache/commons-io/pull/13.
[25] Cleaning directories after each test run, to prevent repository pollution. https://issues.apache.org/jira/browse/CONFIGURATION-638.
[26] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *International Symposium on Foundations of Software Engineering*. 235–245.
[27] Emelie Engström and Per Runeson. 2010. A Qualitative Survey of Regression Testing Practices. In *Product-Focused Software Process Improvement*. 3–16.
[28] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. *Journal of Information and Software Technology* 52, 1 (2010), 14–30.
[29] 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*. 22–31.
[30] Eradicating Non-Determinism in Tests. http://martinfowler.com/articles/nonDeterminism.html.
[31] 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. In *International Conference on Software Engineering, Software Engineering in Practice*. 11–20.
[32] Fabricate. https://github.com/SimonAlfie/fabricate.
[33] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
[34] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *International Middleware Conference*. 101–120.
[35] Milos Gligoric. 2015. *Regression Test Selection: Theory and Practice*. Ph.D. Dissertation. The University of Illinois at Urbana-Champaign.
[36] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *International Conference on Software Engineering, Demo*. 713–716.
[37] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *International Symposium on Software Testing and Analysis*. 211–222.
[38] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. 2014. Automated Migration of Build Scripts using Dynamic Analysis and Search-Based Refactoring. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 599–616.
[39] Gradle Build Tool - Modern Open Source Build Automation. http://gradle.org.
[40] Philip J. Guo and Dawson Engler. 2011. Using Automatic Persistent Memoization to Facilitate Data Analysis Scripting. In *International Symposium on Software Testing and Analysis*. 287–297.
[41] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: detecting state-polluting tests to prevent test dependency. In *International Symposium on Software Testing and Analysis*. 223–233.
[42] Ramzi A. Haraty, Nash'at Mansour, and Bassel Daou. 2001. Regression testing of database applications. In *Symposium on Applied Computing*. 285–289.
[43] Ramzi A. Haraty, Nashat Mansour, and Bassel A. Daou. 2004. Regression test selection for database applications. *Advanced Topics in Database Research* 3 (2004), 141–165.
[44] Jean Hartmann. 2012. 30 Years of Regression Testing: Past, Present and Future. In *Pacific Northwest Software Quality Conference*. 119–126.
[45] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. 2002. *The Vesta Software Configuration Management System*. Research Report. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-177.pdf.
[46] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Automated Software Engineering*. 426–437.
[47] JNI APIs and Developer Guides. https://docs.oracle.com/javase/8/docs/technotes/guides/jni.
[48] KernelGitGuide. https://wiki.ubuntu.com/Kernel/Dev/KernelGitGuide.
[49] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. 1995. Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs. *Journal of Object-Oriented Programming* 8, 2 (1995), 51–65.
[50] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Network and Distributed System Security Symposium*.
[51] 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. In *International Symposium on Foundations of Software Engineering*. 883–894.
[52] Hareton K. N. Leung and Lee White. 1989. Insights into regression testing. In *International Conference on Software Maintenance*. 60–69.
[53] Sheng Liang. 1999. *Java Native Interface : Programmer's Guide and Specification*. Addison-Wesley Longman.
[54] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering*. 643–653.
[55] Peter Macko and Margo Seltzer. 2012. A General-purpose Provenance Library. In *USENIX Conference on Theory and Practice of Provenance*. 6–6.
[56] Apache Maven. https://maven.apache.org.
[57] Philip Mayer and Alexander Bauer. 2015. An Empirical Analysis of the Utilization of Multiple Programming Languages in Open Source Projects. In *International Conference on Evaluation and Assessment in Software Engineering*. 1–10.
[58] Shane Mcintosh, Bram Adams, and Ahmed E. Hassan. 2012. The Evolution of Java Build Systems. *Empirical Software Engineering* 17, 4–5 (2012), 578–608.
[59] Memoize. https://github.com/kgaughan/memoize.py.
[60] Atif M. Memon and Myra B. Cohen. 2013. Automated Testing of GUI Applications: Models, Tools, and Controlling Flakiness. In *International Conference on Software Engineering*. 1479–1480.
[61] Brian S. Mitchell and Spiros Mancoridis. 2006. On the automatic modularization of software systems using the Bunch tool. *Transactions on Software Engineering* 32, 3 (2006), 193–208.
[62] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding Bugs by Isolating Unit Tests. In *International Symposium on Foundations of Software Engineering*. 496–499.
[63] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland, Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, and Robin Smogor. 2009. Layering in Provenance Systems. In *Conference on USENIX Annual Technical Conference*. 10–10.
[64] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. 2011. Regression Testing in the Presence of Non-code Changes. In *International Conference on Software Testing, Verification, and Validation*. 21–30.
[65] Alexander Neundorf. Why the KDE project switched to CMake - and how. http://lwn.net/Articles/188693.

[66] Alessandro Orso and Gregg Rothermel. 2014. Software Testing: A Research Travelogue (2000–2014). In *Future of Software Engineering*. 117–132.

[67] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling Regression Testing to Large Software Systems. In *International Symposium on Foundations of Software Engineering*. 241–251.

[68] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in GitHub. In *International Symposium on Foundations of Software Engineering*. 155–165.

[69] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 432–448.

[70] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *Transactions on Software Engineering* 22, 8 (1996), 529–551.

[71] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210.

[72] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing Trade-offs in Test-suite Reduction. In *International Symposium on Foundations of Software Engineering*. 246–256.

[73] Mats Skoglund and Per Runeson. 2005. A case study of the class firewall regression test selection technique on a large scale distributed software system. In *International Symposium on Empirical Software Engineering*. 74–83.

[74] Mats Skoglund and Per Runeson. 2007. Improving Class Firewall Regression Test Selection by Removing the Class Firewall. *International Journal of Software Engineering and Knowledge Engineering* 17, 3 (2007), 359–378.

[75] Peter Smith. 2011. *Software Build Systems: Principles and Experience*. Addison-Wesley Professional.

[76] R. Spillane, R. Sears, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok. 2009. Story Book: An Efficient Extensible Provenance Framework. In *Workshop on on Theory and Practice of Provenance*. 11:1–11:10.

[77] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively Prioritizing Tests in Development Environment. In *International Symposium on Software Testing and Analysis*. 97–106.

[78] Roman Suvorov, Meiyappan Nagappan, Ahmed E. Hassan, Ying Zou, and Bram Adams. 2012. An empirical study of build system migrations in practice: Case studies on KDE and the Linux kernel. In *International Conference on Software Maintenance*. 160–169.

[79] Testing at the speed and scale of Google. http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

[80] TotT: Avoiding Flakey Tests. http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html.

[81] Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab Mirrokni. 2015. Automated Decomposition of Build Targets. In *International Conference on Software Engineering*. 123–133.

[82] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. 2017. File-Level vs. Module-Level Regression Test Selection for .NET. In *Symposium on the Foundations of Software Engineering, Industry Track*. TO APPEAR.

[83] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *International Symposium on Foundations of Software Engineering*. 805–816.

[84] David Willmor and Suzanne M. Embury. 2005. A Safe Regression Test Selection Technique for Database Driven Applications. In *International Conference on Software Maintenance*. 421–430.

[85] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[86] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *International Conference on Software Maintenance*. 23–32.

[87] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muslu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *International Symposium on Software Testing and Analysis*. 385–396.

[88] Jörg Zinke. 2009. System call tracing overhead. http://www.linux-kongress.org/2009/slides/system_call_tracing_overhead_joerg_zinke.pdf.