Regression Test Selection Across JVM Boundaries

Ahmet Celik[1], Marko Vasic[1], Aleksandar Milicevic[2] and Milos Gligoric[1]

1

2

November 21st , 2017

University of Texas at Austin, TX, USA

Supported by

Regression Testing

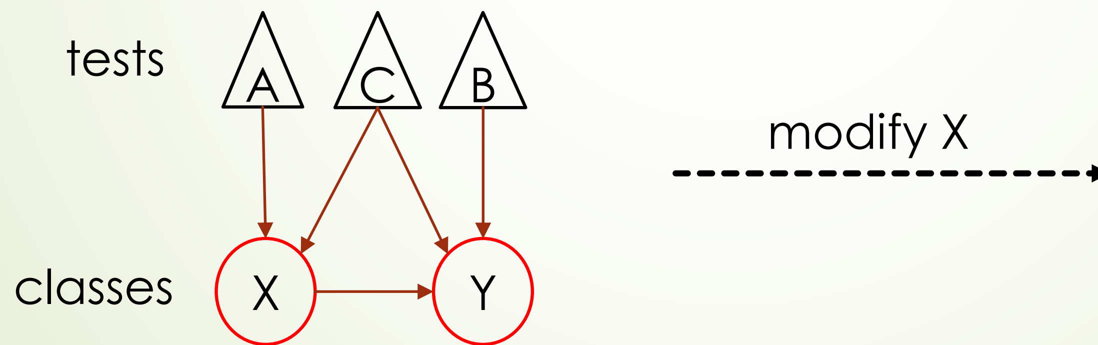
- Checks that recent changes do not break previously working functionality
- Is costly $$$ (time consuming and resource intensive)
- Google's Test Automation Platform (TAP) system on an average day * :
  - 800K builds and 150 Million test runs
- Microsoft's CloudBuild on an average day**:
  - Used by more than 4K developers in Bing, Exchange, SQL, OneDrive, Azure and Office, 20K builds

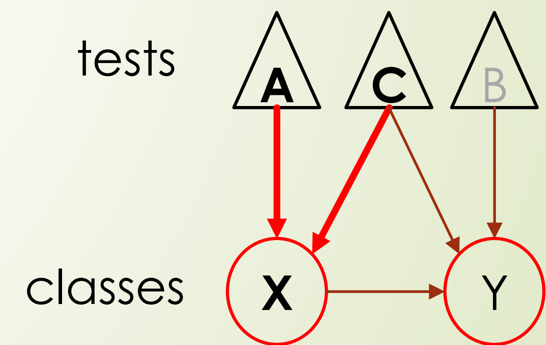
* Memon, Atif, et al. "Taming google-scale continuous testing." Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track. IEEE Press, 2017.

** Esfahani, Hamed, et al. "CloudBuild: Microsoft's distributed and caching build service." Proceedings of the 38th International Conference on Software Engineering Companion. ACM, 2016.

Regression Test Selection (RTS)

- An optimization for regression testing
- Selects tests those are affected by the recent changes
- If it selects all tests affected by the change, then it is **safe**, otherwise **unsafe**
- Two important design choices for RTS
  - What kind of dependencies to track?
    - Below class level file dependencies are tracked
  - How to track them (statically or dynamically)?





Motivation

- Existing dynamic RTS techniques are language specific
  - *Ekstazi* is one of the many dynamic RTS tools, and it only supports Java
  - Recent studies show that several open-source projects are written more than one programming language, i.e. those projects are multilingual
  - **Existing RTS techniques are unsafe for multilingual projects, e.g. Java code that invokes C/C++**
- Existing static RTS techniques are imprecise
  - Google's TAP and Microsoft's CloudBuild track dependencies between projects
  - Over approximate the set of dependencies

An Example from Open Source

- **JavaCPP** (github.com/bytedeco/javacpp) is a popular open source project:
  - Provides efficient access to native C++ inside Java
- Existing dynamic RTS techniques will miss the dependency on `AdapterTest.h`:
  - Since new spawned process access that file
  - Hence it affects **safety** of the technique, i.e. a dependency is not reported by the technique

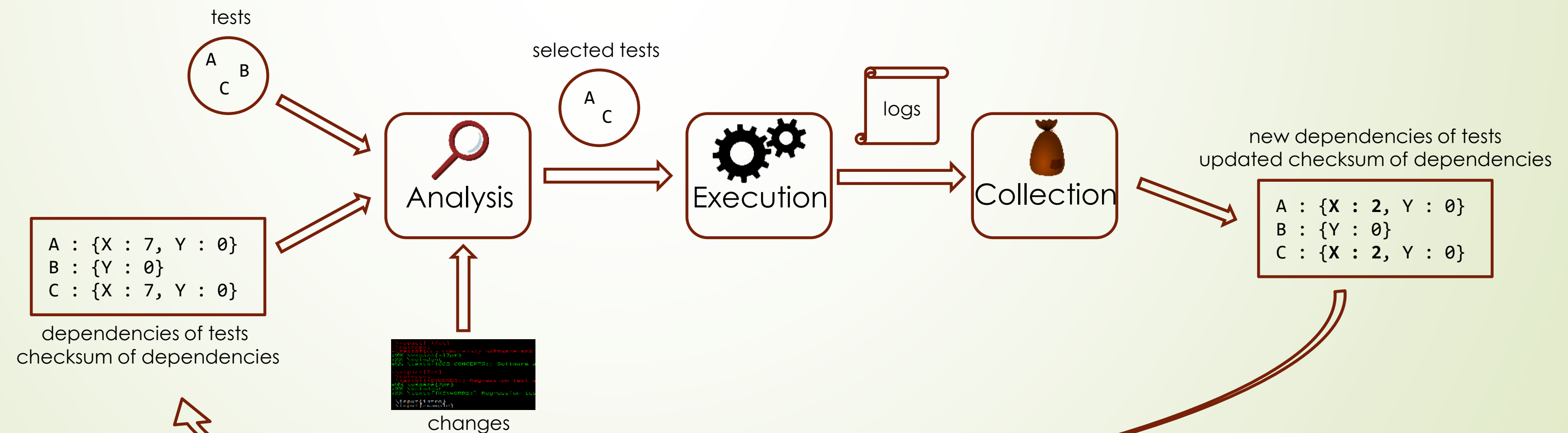
```
...
40: @Platform(compiler = "cpp11",
41:      define = {"SHARED_PTR_NAMESPACE std",
42:                "UNIQUE_PTR_NAMESPACE std"},
43:       include = "AdapterTest.h")
44: public class AdapterTest {
       ...
54:    static native IntPointer testIntString(IntPointer str);
       ...
88:    @BeforeClass public static void setUpClass() throws Exception {
89:        Class c = AdapterTest.class;
90:        Builder builder = new Builder().classesOrPackages(c.getName());
91:        File[] outputFiles = builder.build();
92:        Loader.load(c);
93:    }
       ...
148:   @Test public void testIntString() {
149:       String textStr = "This is a normal ASCII string.";
150:       IntPointer textPtr1 = new IntPointer(textStr);
151:       IntPointer textPtr2 = testIntString(textPtr1);
152:       assertEquals(textStr, textPtr1.getString());
153:       assertEquals(textStr, textPtr2.getString());
154:   }
      ...
209: }
```

Our Solution: RTSLinux

- Collects dependencies at the file level granularity dynamically
- Traces file accesses with Linux support
- Not restricted to one programming language, i.e. language-agnostic
- Focus on capturing dependencies which escape Java Virtual Machine (JVM):
  - Files apart from .class files (such .xml, .txt etc)
  - Files accessed by a newly spawned process
  - Files accessed during the use of native API

Phases of RTSLinux

- RTSLinux includes three phases as a traditional RTS technique:
  - *Analysis:* Checks if any dependency of a test is affected by the recent changes
  - *Execution:* Runs the selected tests by analysis phase
  - *Collection*: Collects dependencies for each test
- Our motivating example is used to demonstrate how RTSLinux performs these phases



Analysis Phase

- In analysis phase
  - if for a test, there is no dependency has been found, it is selected
  - If for a test, there are dependencies and if any of them is modified, then it is selected. We use checksums to decide if a dependency is changed
- For running example, dependencies of `AdapterTest` is shown below. Checksum is -1 for non-existing files

```
# AdapterTest.h
# file or directory                         checksum
/usr/bin/java                               1694755281
...
classes/AdapterTest.h                       3210793863
classes/AdapterTest.class                   3556203470
classes/org/junit/Test.class                -1
...
libs/                                       4098231283
libs/junit-4.12.jar                         2756529828
...
/usr/bin/g++                                3833323531
...
```

Execution

- During test execution, each system call available in Linux that manages the set of running processes (e.g. fork) or accesses the file system (e.g. open) is intercepted
- We save:
  - Map of parent processes
  - File accesses of each processes
- Below you can see logs for the execution of `AdapterTest`

```
# pid program
1186 /usr/bin/mvn
1222 /usr/bin/java
1284 /usr/bin/g++
```

Executions

```
# parentpid pid
1186 1222
1222 1223
1223 1224
1223 1225
1223 1284
```

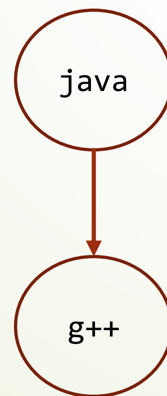
Map of Parents

```
# pid file
1186 /usr/bin/mvn
1186 /usr/bin/java
...
1223 classes/AdapterTest.class
1223 classes/org/junit/Test.class
...
1223 libs/
1223 libs/junit-4.12.jar
...
1284 /usr/bin/g++
1284 classes/AdapterTest.h
...
```

File accesses of each process

Collection

- Create process tree from these logs. Root of the tree is the `java` command which is used to run tests
- Compute and save checksum of each file which is accessed by a process in the tree
- As we shown before in analysis, output of this phase for `AdapterTest` is below:

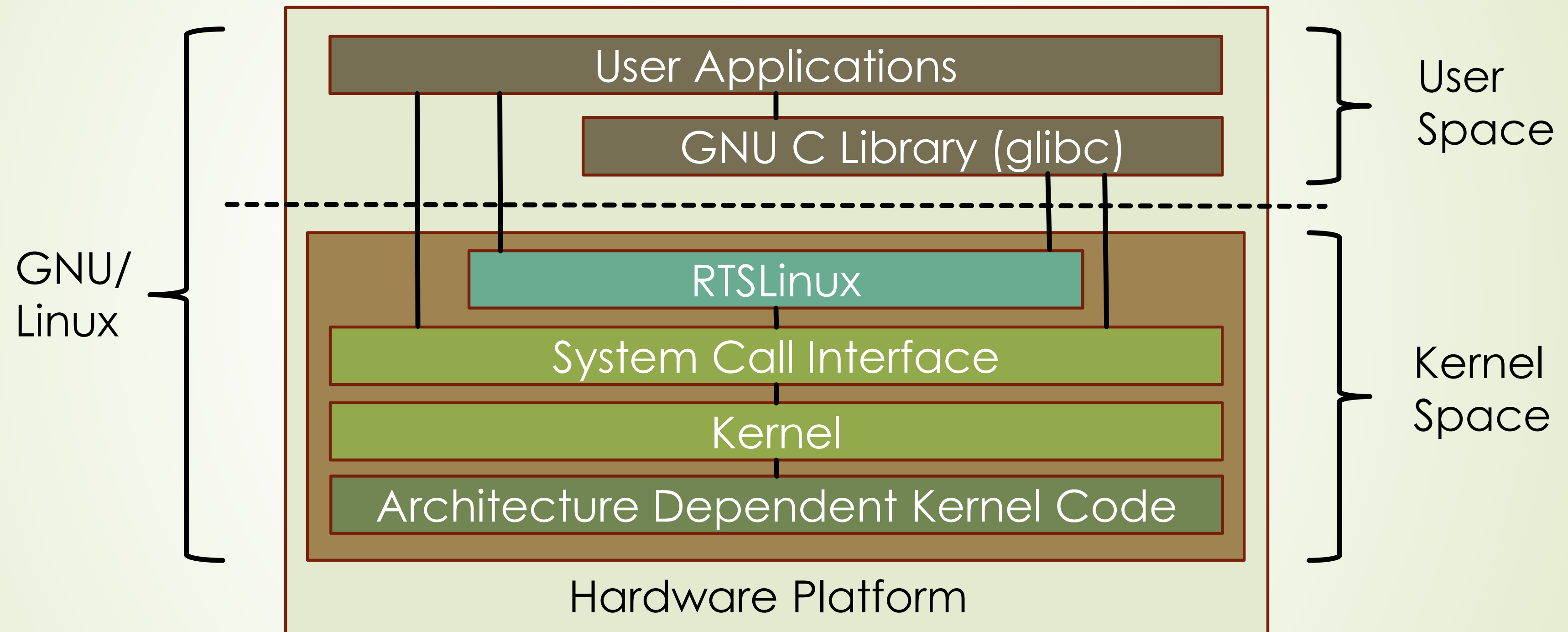



```
# AdapterTest.h
# file or directory                          checksum
/usr/bin/java                                1694755281
...
classes/AdapterTest.h                        3210793863
classes/AdapterTest.class                    3556203470
classes/org/junit/Test.class                 -1
...
libs/                                        4098231283
libs/junit-4.12.jar                          2756529828
...
/usr/bin/g++                                 3833323531
...
```

System Integration

- We implemented the proposed technique in two different ways:
  - Using an existing build system Fabricate in user space, dubbed RTSFab
  - Loadable Linux kernel module in kernel space, dubbed RTSLinux
- There are many other ways to implement
- `IsTestAffected` (Analysis), `SystemExecute` (Execution), and `StoreDeps` (Collection) are implementations of the three phases
- A developer of a build system can use these primitives to select tests
  - We integrated with Maven
  - Instead of running `mvn test` the user should run `rtslinux mvn test`

Loadable Kernel Module



Evaluation

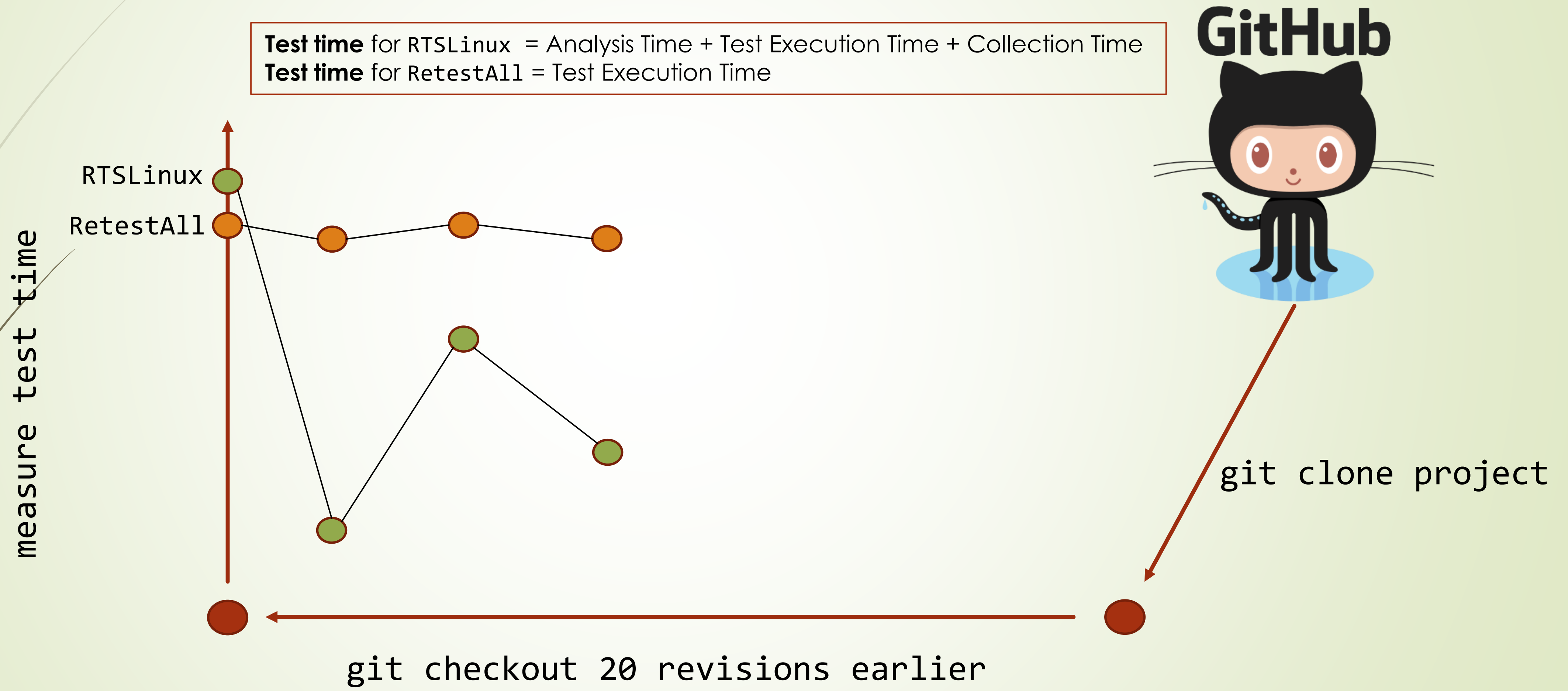
- RQ1: How effective is RTSLinux , i.e. what is the reduction in testing time and the number of executed tests?
- RQ2: What are the benefits/drawbacks of dependency detection across JVM boundaries (as implemented in RTSLinux) compared to a single-JVM RTS (as implemented in Ekstazi):
  - RQ2.1 (Effectiveness): 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 (recall: an implementation of our technique running in user space) compared to RTSLinux?

Study Setup

- Used 21 projects, and 20 revisions for each project
- Escape Method is that how a project escapes from JVM

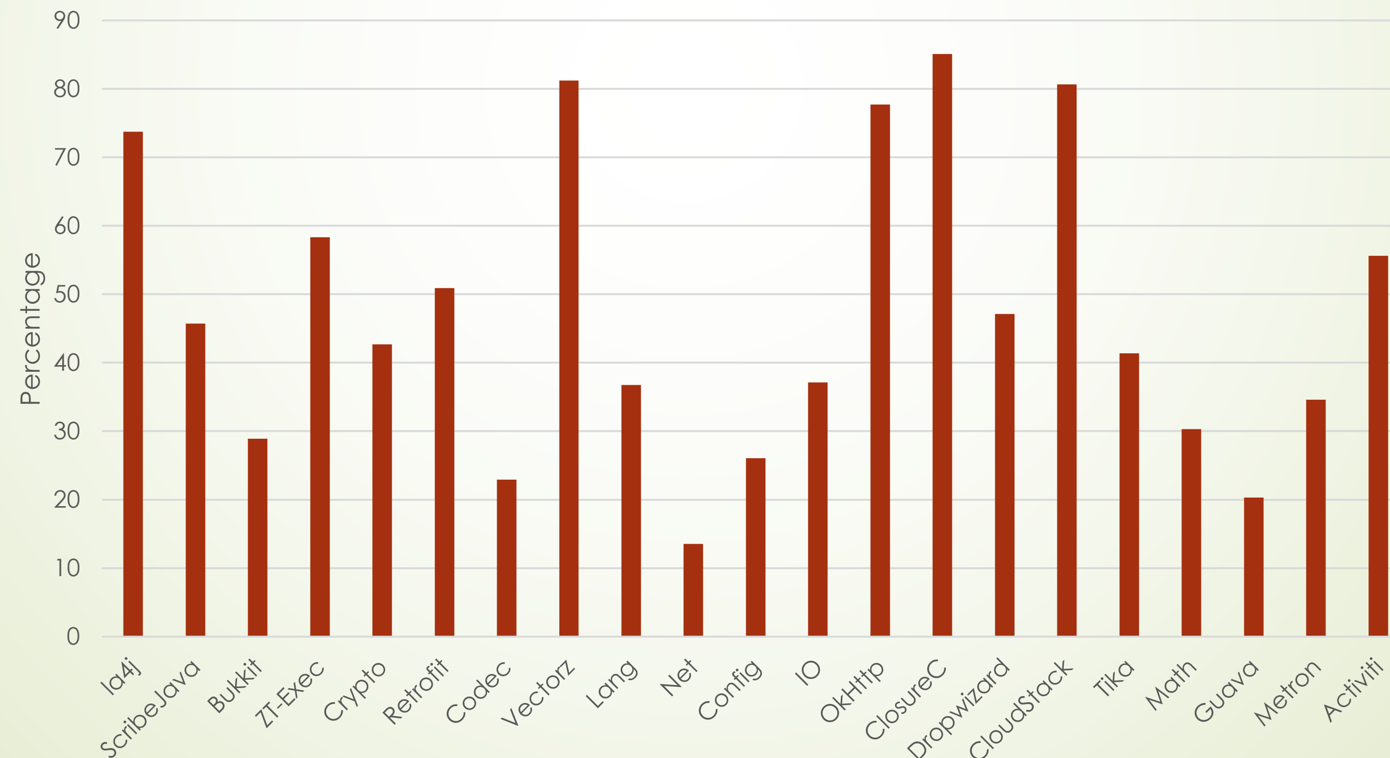
Project	LOC	# Files	# Test Classes	Test Time [s]	Escape Method
la4j	13390	147	22.85	14.68	N/A
ScribeJava	7613	219	20	14.85	N/A
Bukkit	32555	762	38	21.66	N/A
ZT-Exec	2938	104	18.45	25.04	Processes
Crypto	5079	140	24	27.94	Native Calls
Retrofit	12331	202	30.75	28.74	Files
Codec	17625	299	48	31.72	Files
Vectorz	52096	414	70.5	38.98	N/A
Lang	69014	381	133.5	41.21	Files
Net	26928	315	42	65.13	Files
Config	64341	642	162.3	66.05	Files
IO	27186	302	91	89.1	Files
OkHttp	48783	344	59.4	101.72	Files
ClosureC	284131	1548	309.3	190.41	Native Calls
Dropwizard	37914	969	232	328.84	Processes
CloudStack	572503	7585	292	335.42	Processes
Tika	96220	1936	227.65	370.08	Processes
Math	174832	1501	431	376.46	Files
Guava	244083	1737	401	424.66	Files
Metron	57720	1507	145	462.28	Processes
Activiti	203509	5523	312.35	879.99	Processes

Experiment Procedure



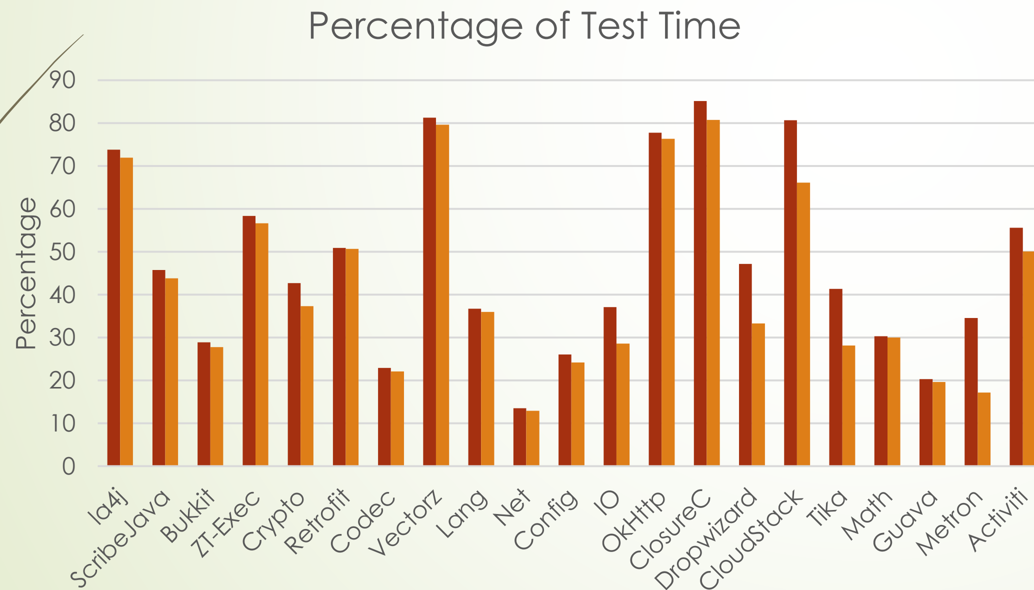
RQ1: How effective is RTSLinux?

- Our results show that RTSLinux reduces **test time** for all projects
- For each revision `100 * RTSLinux / RetestAll,` averages of 20 revisions are shown on chart
- On average, across all projects, test time is decreased to 47.17%

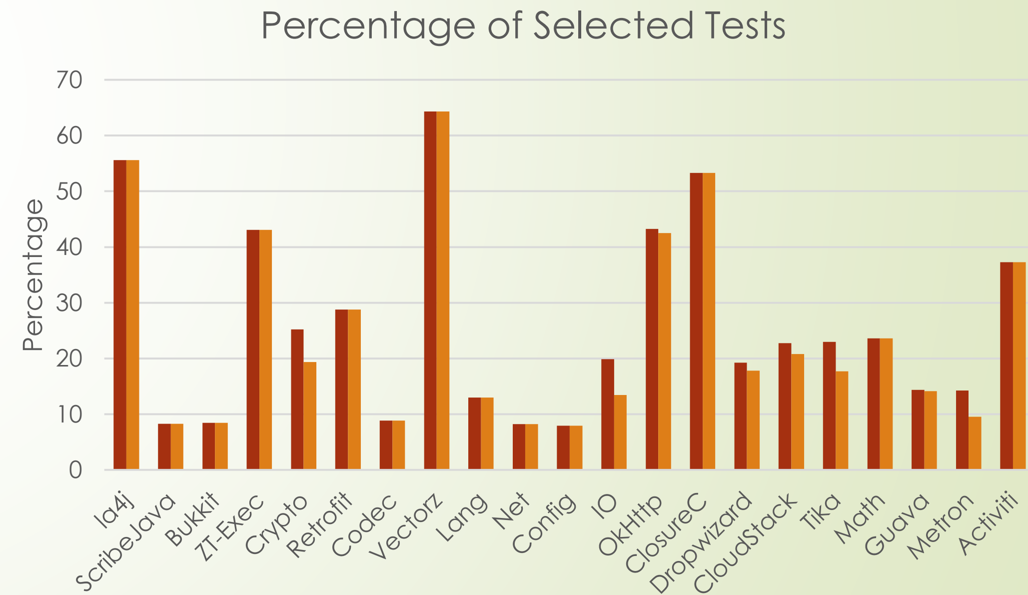


RQ2.1:Does RTSLinux achieve as much reduction in total testing time and number of executed tests (as Ekstazi)?

- Averages of 20 revisions are shown
- Test time, `100 * X / RetestAll where X = {RTSLinux, Ekstazi}`
- # selected tests, `100 * X / RetestAll where X = {RTSLinux, Ekstazi}`



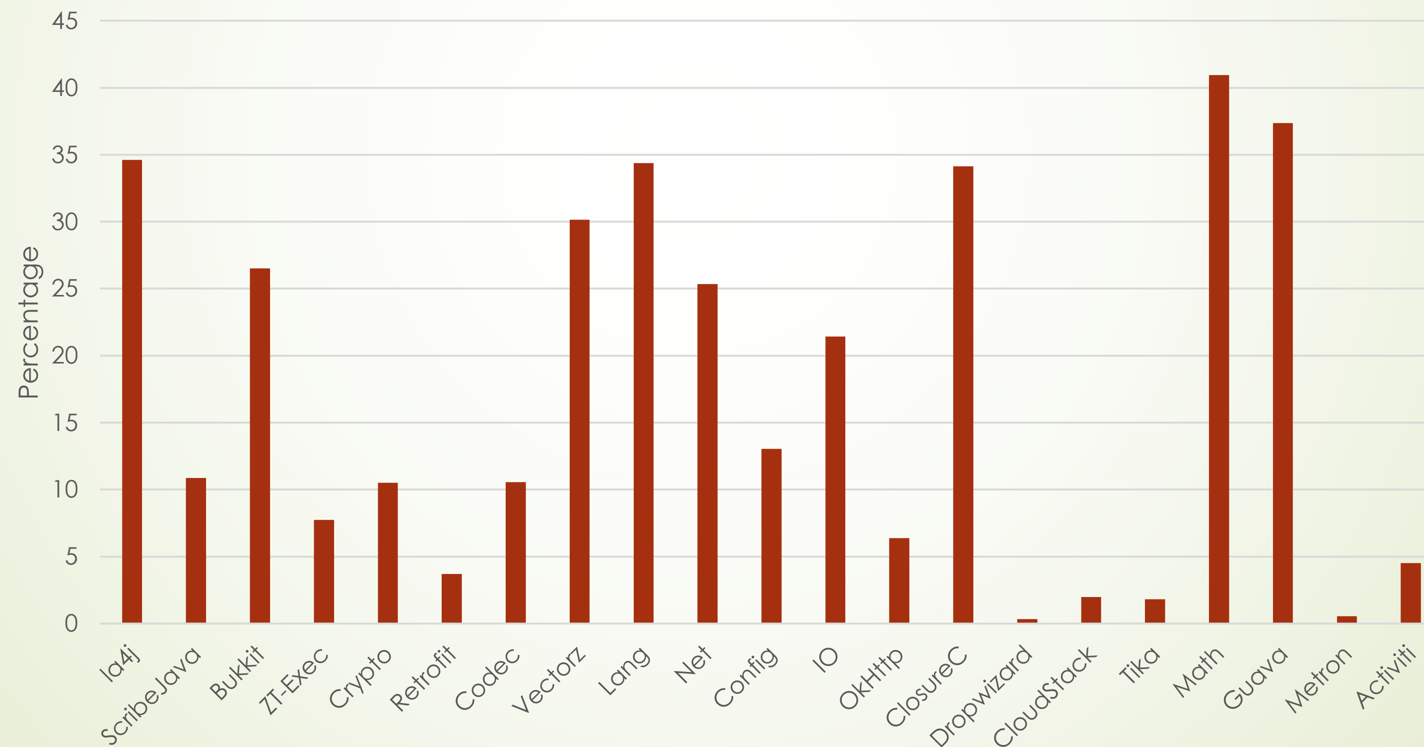
47.17% vs 42.53%



25.83% vs 24.56%

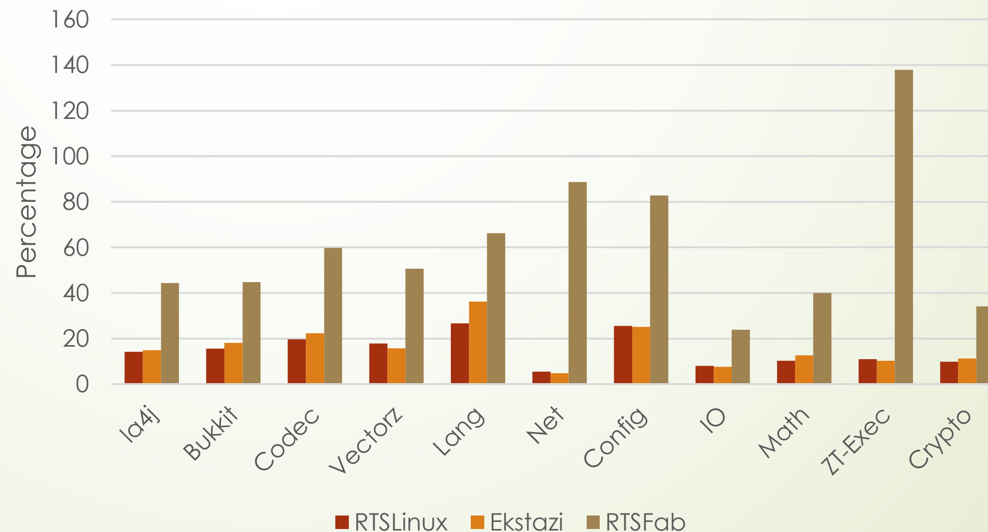
RQ2.2: How many more dependencies are discovered by RTSLinux?

- Averages of 20 revisions are shown
- # dependencies discovered, `100 * Ekstazi / RTSLinux`



RQ3: Overhead of RTSFab, RTSLinux and Ekstazi compared to RetestAll?

- RTSFab is an implementation of our technique running in user space
- Here we show overhead during execution phase for projects with single Maven module for the first revision
- Test time, `100 * X / RetestAll - 100` where X is one of `{RTSLinux, Ekstazi, RTSFab}`



Conclusion

- A novel regression test selection technique, dubbed RTSLinux
- RTSLinux supports tests that escape JVM
- RTSLinux saves 52.83% of test time compared to RetestAll
- In future, we are considering
  - To evaluate and extend to other languages
  - Try different implementations using different tracing options

Ahmet Celik <ahmetcelik@utexas.edu>
Marko Vasic <vasic@utexas.edu>
Aleksandar Milicevic <almili@microsoft.com>
Milos Gligoric <gligoric@utexas.edu>