Probe-based Syscall Tracing for Efficient and Practical File-level Test Traces

Daniel Elsner
daniel.elsner@tum.de
Technical University of Munich
Munich, Germany

Roland Wuersching
roland.wuersching@tum.de
Technical University of Munich
Munich, Germany

Markus Schnappinger
markus.schnappinger@tum.de
Technical University of Munich
Munich, Germany

Alexander Pretschner
alexander.pretschner@tum.de
Technical University of Munich
Munich, Germany

ABSTRACT

Efficiently collecting per-test execution traces is a common prerequisite of dynamic regression test optimization techniques. However, as these test traces are typically recorded through language-specific code instrumentation, non-code artifacts and multi-language source code are usually not included. In contrast, more complete test traces can be obtained by instrumenting operating system calls and thereby tracing *all* accessed files during a test's execution. Yet, existing test optimization techniques that use syscall tracing are impractical as they either modify the Linux kernel or operate in user space, thus raising transferability, performance, and security concerns. Recent advances in operating system development provide versatile, lightweight, and safe kernel instrumentation frameworks: They allow to trace syscalls by instrumenting probes in the operating system kernel. Probe-based Syscall Tracing (ProST), our novel technique, harnesses this potential to collect file-level test traces that go beyond language boundaries and consider non-code artifacts. To evaluate ProST's efficiency and the completeness of obtained test traces, we perform an empirical study on 25 multi-language open-source software projects and compare our approach to existing language-specific instrumentation techniques. Our results show that most studied projects use source files from multiple languages (22/25) or non-code artifacts during testing (22/25) that are missed by language-specific techniques. With the low execution time overhead of 4.6% compared to non-instrumented test execution, ProST is more efficient than language-specific instrumentation. Furthermore, it collects on average 89% more files on top of those collected by language-specific techniques. Consequently, ProST paves the way for efficiently extracting valuable information through dynamic analysis to better understand and optimize testing in multi-language software systems.

CCS CONCEPTS

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

KEYWORDS

software testing, dynamic program analysis, multi-language software, non-code artifacts

ACM Reference Format:
Daniel Elsner, Roland Wuersching, Markus Schnappinger, and Alexander Pretschner. 2022. Probe-based Syscall Tracing for Efficient and Practical File-level Test Traces. In *IEEE/ACM 3rd International Conference on Automation of Software Test (AST '22), May 17–18, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524481.3527239

1 INTRODUCTION

Regression testing is regularly performed on software systems to ensure that changes have not inadvertently affected existing system behavior [43]. Since regression testing is prone to be costly for large software systems [24, 75], optimization techniques such as regression test selection [15, 21, 22, 28, 42, 59, 65, 66, 69, 70, 78], test case prioritization [13, 16, 19–22, 49, 61, 68, 70], test suite minimization [17, 18, 35, 48, 67, 72, 73], and flaky test analysis [10, 41, 45, 64, 82] have been extensively studied since the 1970s [25]. These techniques exhibit methodological overlap [75]: Several of the proposed approaches collect *per-test dependencies*, i.e., those parts of the code that are covered by one specific test. Techniques which collect test dependencies during test execution are called *dynamic* and essentially record per-test execution traces. These *test traces* can be collected on different granularity levels such as statement [10], basic-block [36, 59], function [79, 80], class/file [15, 28, 29], or combinations thereof [78][1]. Due to the lower instrumentation overhead, file-level test traces have been shown to be particularly effective [15, 28, 29].

Nonetheless, collecting test traces usually requires language-specific instrumentation of the code and often bears prohibitively expensive costs [21, 46, 62]. Despite this effort, collected test traces are incomplete as language-specific techniques are usually not capable of collecting test traces across language boundaries in multi-language software and ignore non-code artifacts [15, 46, 56]. This is problematic since software projects are on average written in five to seven general purpose programming languages (GPLs) and domain specific languages (DSLs) [53, 54], and make significant use of non-code artifacts [11]. Incomplete test traces thus distort results

[1] Here, a *test trace* refers to the set of (code) artifacts covered by a test during execution.

AST '22, May 17–18, 2022, Pittsburgh, PA, USA

of test optimization techniques and pose non-negligible threats to supposedly safe regression test selection [81].

Celik et al. [15] propose a different approach to obtain test traces: They collect the set of per-test dependencies by intercepting process- and file-related system calls during test execution. While they are able to extract more complete file-level test traces and achieve language inter-operability, their technique is arguably impractical, as it requires tainting the Linux kernel to run in kernel space. This raises concerns regarding transferability (e.g., what about closed-source operating systems), maintainability (e.g., every kernel release may break it), and security (e.g., risk of corrupting the kernel). Alternative, less intrusive approaches that operate in user space are too inefficient to be considered as cost-effective [15].

In this paper, we propose a different approach, Probe-based Syscall Tracing (ProST), for obtaining file-level test traces which include multi-language source files and non-code artifacts. In contrast to prior approaches, ProST harnesses modern kernel instrumentation frameworks that allow tracing relevant syscalls by instrumenting probes in the operating system kernel. These frameworks are versatile, lightweight, and operate in safe execution environments inside the kernel. Therefore, they are already used for performance analyses in production systems at Netflix and Facebook, among others [30, 33].

To provide an empirical view on the state-of-the-art in testing multi-language software, we first performed a motivating study and empirically investigated how non-code artifacts and source files are accessed during testing. We investigated 25 open-source projects with a combined size of $>18M$ physical lines of code (LOC). We find that 24 of the analyzed multi-language projects do indeed access source files from multiple programming languages (22/25) or non-code artifacts during testing (22/25), which are missed by language-specific instrumentation. This emphasizes the need for more complete test traces.

We then applied our technique ProST in these projects to empirically evaluate efficiency, on the one hand, and the increased completeness of file-level test traces when compared to language-specific code instrumentation, on the other hand. Our results show that while ProST has a low execution time overhead of 4.6% added to non-instrumented test execution, it collects on average 89% more files on top of those covered by language-specific instrumentation techniques. These encouraging results show that ProST is an efficient and more effective approach to recording test traces—the overhead introduced by the language-specific instrumentation is slightly higher at 7.4%. Finally, to demonstrate the practicality of ProST, we implemented it for Windows, Linux, and macOS and report differences in the experimental results.

ProST can be used to analyze multi-language software systems and enables advances in regression test optimization. In fact, we are already using test traces collected with ProST for regression test selection in a large-scale ($>20M$ LOC) multi-language software inside Windows environments at our industry partner IVU Traffic Technologies[2].

In summary, our contributions are as follows:

- **Empirical pre-study:** Evidence that multi-language software projects do indeed access source files of multiple programming languages and non-code artifacts during testing.
- **Novel approach:** A practical approach for obtaining file-level test traces from system calls through probe-based kernel instrumentation. Contrary to prior Linux-only syscall tracing approaches, we implement ProST as an analysis tool that currently supports the operating systems Windows, Linux, and macOS.
- **Comparative evaluation:** Demonstration of ProST's efficiency and effectiveness by comparing its overhead and test trace completeness to language-specific instrumentation techniques.

2 RELEVANCE OF THE PROBLEM: MOTIVATING STUDY

Studies by Mayer et al. in 2015 [53] and 2017 [54] found that software projects are on average written in five to seven GPLs and DSLs, making multi-language programming "a factor which must be dealt with in tooling and when assessing development and maintenance" [53]. The authors statically analyzed open-source software repositories and interviewed software developers for issues during multi-language development. Similarly, Bigliardi et al. [11] statically analyzed open-source repositories for non-code software artifacts. They found that, on average, almost 50% of files in a software project are non-code artifacts. Moreover, in the set of 21 open-source projects analyzed by Shi et al. [69] on average 7.5% of commits contain non-code changes that are relevant for testing.

While testing appears to be especially challenging with the existence of cross-language links [54], there is no empirical evidence of whether multi-language software projects actually make use of such cross-language links during testing. However, if (1) software is often tested across language boundaries, or (2) tests frequently operate on non-code artifacts, the need to collect test traces across boundaries arises. To motivate our main study and validate the relevance of the problem at hand, we first investigate the motivating research questions (MRQs):

- **MRQ$_1$**: How commonly do projects test across language boundaries and which combinations of GPLs and DSLs are most frequently observed?
- **MRQ$_2$**: How commonly do projects access non-code artifacts during testing and which file extensions are most frequent?

To answer these questions, we analyze popular open-source multi-language software projects as described in the following.

2.1 Project Selection

In this study, we collect test traces by monitoring file accesses. However, traces must only be collected during testing activities. Hence, it is crucial to clearly separate the build process from the testing process. To keep the manual effort at a reasonable level, we limit ourselves to projects using Maven as their build management tool. Its unified build lifecycle allows to separate building from testing [6]. In addition, Maven is considered frequently in related literature and provides a large ecosystem of readily available analysis and instrumentation tools [15, 50]. Considering only Maven projects will most likely result in projects that use at least

[2]IVU Traffic Technologies is a leading provider of public transport software solutions.

one GPL from the Java Virtual Machine (JVM) ecosystem (e.g., Java, Groovy, Scala, or Kotlin). Yet, analyzing such projects is particularly interesting, given that most existing regression testing research also targets Java [27]. We query GitHub's application programming interface (API)[3] to find relevant open-source projects for our study. Eventually, all projects in our sample satisfy the following requirements:

- **Real-world project:** The project is an actual software library, framework, or application or a combination thereof. We exclude pure demo or example projects.
- **Popularity and active development:** The project repository on GitHub has at least 100 stars and commit activity within the last year was observed.
- **Multi-language project:** At least two GPLs are found in the repository with all JVM-based languages counting as one instance. We distinguish between DSL and GPL in this case, to study actual multi-language projects, which excludes projects that only use additional languages for build, documentation, or, in general, auxiliary purposes (e.g., project website, or release and smoke test scripts).
- **Buildable and testable:** The project uses the Maven build system and is separately buildable and testable on Ubuntu 20.04.
- **Testing frameworks:** To limit manual interference and to ensure consistent control of test execution and analysis of test reports, we only include projects that use Maven's testing plugins Surefire [7] and Failsafe [51], and the JUnit framework [38].
- **English project documentation:** The project documentation is available in English, which facilitates understandability and reproducibility.

Not all of these requirements can be automatically checked for and GitHub's API does not provide capabilities to query for Maven projects only. Hence, to obtain 25 valid projects that meet the outlined requirements, we manually inspected the 114 most popular projects with a `pom.xml` file as retrieved via the GitHub API. The reproduction package contains analysis scripts, the list of GitHub projects and, if required, an explanation why a project was not included in the final set of projects[4]. Table 1 summarizes those reasons for project exclusion. Notably, it is not uncommon that only roughly one out of five open-source projects is usable in studies that perform dynamic program analysis [69].

Table 1: Reasons for project exclusions with frequencies

Reason for exclusion	Frequency
Not English	27
Not multi-language	24
Not buildable and testable	22
No tests	8
Not JUnit/Surefire/Failsafe	5
Demo/Example project	2
No activity	1

[3]GitHub REST API: https://docs.github.com/en/rest
[4]Reproduction package [23]

Table 2 lists the final set of 25 projects together with the revision used in our study and their size (in physical LOC as measured with `scc` [12]). In total, the study corpus accounts for >18M LOC and represents a variety of domains, developer groups, and project sizes.

2.2 Experiment Execution

As some projects require globally installed software packages (e.g., the GNU Compiler Collection), a specific Java Development Kit (JDK) version, a Docker daemon for running tests against a containerized database, or certain command line flags, we manually created two shell scripts for each project, one for building (`build.sh`), and one for testing the project (`test.sh`). The execution of the tests was embedded into the tool `strace` [5]. Although `strace` is a Linux-only system call tracing tool that operates in user-space and thus induces high overhead [15, 83], it suffices for the sake of this motivating study. Using `strace`, we log all `open` and `openat` system calls for the shell process executing `test.sh` into a log file and transitively follow all spawned child processes.

During the analysis of all accessed files, we excluded (1) log and temporary files generated by the build or testing framework, and (2) files that are not located below the project directory, the `/usr/local` directory (contains e.g., self-compiled C libraries), or the local Maven repository. We exclude such files as they are related to the execution environment (i.e., JDK or operating system) and thus not under immediate control of the developer. For instance, just because the executing JVM uses a system library implemented in a different language does not make the test cross-language. We then identified the used DSLs and GPLs via their corresponding file extensions by using the definitions for GPL and DSL from Mayer et al.'s taxonomy for cross-language links [52]: Languages which can be used to write "arbitrary application code" [52] are considered as GPLs (e.g., Java, C/C++, or JavaScript), whereas DSLs have a distinct application area. Examples for DSLs are HTML for UI specification or SQL for querying databases, but also configuration formats such YAML, JSON, XML, or Java-Properties (`.properties`). All other extensions such as `.csv` or `.txt` are considered as non-code artifacts.

2.3 Results

Table 2 shows the number of test classes and test methods that were executed during testing. The rightmost three columns of the table show the results of the file access analysis. Each accessed file is only counted once to not distort results due to commonly loaded configuration files or setup classes.

During testing, 22 (88%) projects access files of more than one programming language: 20 (80%) projects use at least one DSL and 8 (32%) use more than one GPL. On average, the studied projects access more than 4 distinct programming language file extensions (DSL or GPL). The three most common GPL/DSL and GPL/GPL combinations are Java/XML (18), Java/Properties (14), Java/JSON (9), and Java/JavaScript (5), Java/C/C++ (4), Java/Python (2), respectively. This confirms static analysis results from prior multi-language research [11, 53, 54]. Moreover, it empirically manifests the necessity of test traces that also capture file accesses across language boundaries, since 88% of the studied multi-language projects would otherwise have incomplete test traces.

Table 2: Projects used in the empirical pre-study. The rightmost three columns of the table show the measured distinct accessed file extensions during the execution of the projects' tests.

Project	SHA	LOC	#Files	#Test classes	#Test methods	Distinct file extensions during testing*		
						#Non-code	#DSLs	#GPLs
anserini	ed57c3fb	6,086,489	981	137	462	40	3	1
cometd	00db4aca	123,530	681	209	1,120	1	2	2
consulo	0d1ecce2	1,990,244	15,705	152	770	1	1	1
enunciate	866b50b2	181,003	1,311	17	56	3	6	1
fess	23e17bef	357,220	1,744	58	202	4	5	1
graphwalker-project	60d07c01	334,273	650	96	435	4	3	1
guacamole-client	002cfded	202,865	1,580	13	38	0	0	1
hopsworks	f6539935	355,920	2,163	19	173	1	1	1
incubator-streampipes	4de5956b	185,193	2,745	57	140	1	0	1
IPED	41e9632f	296,302	1,436	22	58	1	0	2
joyqueue	a88914c5	333,963	2,881	47	105	0	1	1
jzmq	00e699ef	203,742	1,563	12	80	2	0	2
languagetool	5a553989	2,361,342	2,814	576	1,283	12	4	1
lavagna	dc3a01a5	117,429	1,518	61	486	3	4	1
openmeetings	feb25454	171,419	1,025	77	286	16	4	2
primefaces	5b548c59	1,951,938	5,350	42	439	2	2	1
psi-probe	19f74553	80,701	655	123	254	0	3	1
redpen	87502989	72,900	358	105	1,126	3	2	2
searchcode-server	b6188ced	95,872	428	46	390	17	0	1
servicecomb-java-chassis	9f90bcdf	307,463	3,600	601	2,928	8	7	1
servicecomb-pack	441a6adc	74,267	849	61	285	2	4	1
sonar-php	16f4c41c	1,029,242	7,843	545	1,322	3	2	2
steady	b78e2b27	373,942	2,274	96	336	20	5	3
tinkerpop	dca51406	525,289	8,462	377	29,000	11	5	2
webprotege	3b2f18d9	344,394	4,682	621	4,397	2	3	1

* Projects having non-code artifacts, projects having DSLs, projects having more than one GPL in their test trace.

> **MRQ$_1$** *We find that in 88% of the multi-language projects we studied, tests access source files of multiple programming languages. The most common GPL/GPL and GPL/DSL combinations are Java/JavaScript and Java/XML.*

In addition, 22 (88%) out of the 25 projects use non-code artifacts during their test execution. On average, a project accesses more than 6 distinct non-code artifact file extensions, where the most common extensions are `.txt` (12), `.csv` (5), and `.zip` (5). Again, these results show that in order to provide reliable insights, test traces must reflect this state-of-practice and take non-code artifacts into account.

> **MRQ$_2$** *We find that in 88% of the multi-language projects we studied, tests access non-code artifacts. The most commonly accessed non-code artifacts are* `.txt` *files.*

3 PROBE-BASED SYSCALL TRACING FOR FILE-LEVEL TEST TRACES

Multi-language software requires complete test traces that include cross-language links and non-code artifacts. This can be achieved using system call tracing. However, existing system call tracing approaches are impractical, which motivates our proposed technique, ProST. In this section, we first give a brief introduction to operating system call tracing and elaborate on the idea of tracing system calls using probe-based kernel instrumentation. We then explain implementation details of ProST and its integration with testing frameworks.

3.1 System Call Tracing

Operating systems have two primary functions, managing resources, and providing services and abstractions to user programs [71]. For the latter, *system calls* (or *syscalls*) represent the interface to the operating system kernel that is visible to application programmers. While the available system calls vary between operating systems, they often share concepts for managing processes, files, memory, threads, or sockets.

By tracing the invoked system calls during a program's execution, its low-level behavior can be analyzed. This has been shown to be useful for performance engineering [39], malware detection [74], intrusion detection [47], energy consumption estimation [60], or fault localization [77]. Yet, most relevant to our work, system call analysis has further been successfully applied for regression test selection by Celik et al. [15]: Their technique `RTSLinux` traces process- and file-related system calls to obtain file-level test traces and then selects affected tests based on which files have changed since the last test execution. However, `RTSLinux` requires modifying the Linux

kernel through a custom Linux kernel module to run efficiently in kernel mode. We have alluded to why this is impractical for transferability, maintainability, and security reasons. To provide an alternative, less intrusive approach to system call tracing, Celik et al. implement their technique in user space using `strace`, which, however, introduces the significant overhead of 61.16% compared to non-instrumented test execution (approximately 4 times the overhead of `RTSLinux`, 14.9%) [15].

Given that none of the existing approaches for system call tracing is both efficient and non-intrusive, we suggest the use of *probe-based kernel instrumentation* in the next subsection.

3.2 Probe-based Kernel Instrumentation

Dynamic instrumentation (also referred to as *dynamic tracing*) has been around at least since the early 1990s and builds on the idea of dynamically inserting instrumentation during the execution of a system [14, 32, 34, 37]. This reduces instrumentation overhead as it allows engineers to instrument only those parts of the system that they are interested in [37]. Technology that enables such dynamic instrumentation is commonly referred to as *dynamic instrumentation framework* or *observability technology* [32, 34].

`DTrace` [14][5] is the first such technology that we investigate: After its development was initiated at Oracle for Solaris in 2005, it has been integrated into macOS/Mac OS X (2007), FreeBSD (2008), and more recently into Windows (2019) [34, 63]. `DTrace` provides capabilities to dynamically instrument so-called *probes*, which are static or dynamic instrumentation points. Users can then write programs in a C-like scripting language that are executed if those probes fire. While there have been efforts to port `DTrace` to Linux[6], they were never merged into the kernel. However, in 2014[7], an extension of the Berkley Packet Filter (BPF)—an in-kernel virtual machine originally created to analyze network traffic in 1993 [55]—has been introduced to the Linux kernel [26]. This extension, called eBPF, allows to run programs in response to instrumentation points in the kernel *beyond* networking subsystems. This new flexibility has led to the development of several front-ends for eBPF, including `bpftrace`[8], which provides a similar interface and capabilities as `DTrace`, albeit using eBPF [31].

In particular, both `DTrace` and `bpftrace` share the concepts of *probes*, i.e., instrumentation points, *predicates*, i.e., conditions that allow filtering when a probe fires, and *actions*, i.e., the programs or sets of instructions that are executed if a probe fires [32, 34]. For all the above mentioned operating systems' kernels, probes are (static) instrumentation points which are added during kernel compile-time as `NOP`-instructions. `DTrace` and `bpftrace` can dynamically register actions as event listeners to these probes during runtime. Similar to `DTrace`, `bpftrace` offers a C-like scripting language to define actions for probes. Listing 1 shows two sample scripts that instrument the `open` system call on macOS using `DTrace` and on Linux using `bpftrace`. In the given examples, an action is defined, that prints the current process identifier (PID) and the file path to be opened. This action is associated to the probe identifier (`syscall::open:entry`) and a predicate (`/pid == 123/`).

```
// Instrumentation of `open` on macOS using DTrace
syscall::open:entry
/pid == 123/
{
    printf("%d %s\n", pid, cleanpath(copyinstr(arg0)));
}

// Instrumentation of `open` on Linux using bpftrace
tracepoint:syscalls:sys_enter_open
/pid == 123/
{
    printf("%d %s\n", pid, str(args->filename));
}
```

Listing 1: Sample scripts that instrument the `open` system call using `DTrace` on macOS and `bpftrace` on Linux.

Both technologies, `DTrace` and `bpftrace`, contribute to the practicality of tracing regarding performance overhead and security issues [33]: Registered actions run in their own safe execution environment inside the kernel and are designed for minimal overhead during operation [34]. In fact, both `DTrace` and `bpftrace` are used in production environments at Netflix and Facebook, among others [30, 33].

3.3 Implementation and Integration

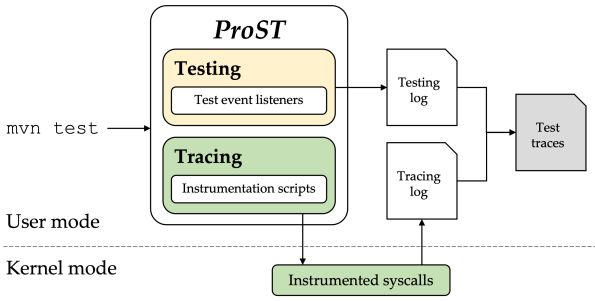

Figure 1: High level architecture of ProST

Fig. 1 illustrates the high level architecture of ProST. ProST has two distinct components, one concerned with (syscall) *tracing* and one with integration into *testing* frameworks. In the following, we outline how these two components work together to collect file-level test traces by instrumenting system calls.

3.3.1 Tracing. We have already motivated why we rely on the dynamic instrumentation technologies `DTrace` and `bpftrace` to trace system calls. To provide support for different operating systems, ProST acts as a facade to these technologies and selects the suitable `DTrace` or `bpftrace` instrumentation script for the corresponding host operating system. Each script takes a root process' PID as input. It then keeps track of all transitive child processes: When a process is spawned, the new PID is added to the set of tracked PIDs and the information triple (*timestamp*, *parentPID*, *PID*) is written to the standard output stream, which is redirected to a *tracing log* file. Furthermore, system calls related to file accesses are instrumented

[5] `DTrace` is short for Oracle Solaris Dynamic Tracing Facility [34]
[6] Linux port of `DTrace`: https://github.com/dtrace4linux/linux
[7] Initial proposal of eBPF on the Linux kernel mailing list by Alexei Starovoitov in 2013: https://lkml.org/lkml/2013/12/2/1066
[8] `bpftrace`: https://bpftrace.org/

under the condition (predicate) that the caller's PID is in the set of tracked PIDs. Whenever a file is opened, the instrumentation adds the information triple (*timestamp*, *PID*, *filepath*) to the tracing log. While this information already allows collecting all transitively accessed files for each PID, it is still insufficient to link tests and files to obtain test traces. The reasons are that (1) PIDs may be reused by the operating system and (2) we miss a mapping of test identifier to PID. Therefore, we require a second artifact, the *testing log*, which is described next.

3.3.2 Testing. To associate accessed files from the tracing log with the corresponding test that opened them, we need to know when a test started and when it ended. Testing frameworks typically offer a test event listener API, that allows defining custom event listeners (e.g., JUnit [8], GoogleTest [2], Jest [3]). ProST currently implements a JUnit test event listener that logs the information (*timestamp*, *PID*, *testidentifier*, *eventtype*) to the testing log whenever a test suite is started and ended[9]. The JUnit test event listener can either be added as an external dependency or to the JVM classpath of the system under test. ProST can be easily used with other testing frameworks that support test event listeners as well, since our logging format is not specific to any testing framework or programming language.

Mapping file accesses to tests through PIDs and timestamps, ProST combines tracing and testing log to construct file-level test traces as shown in Fig. 1. More formally, the test traces are a set of tests T, where each test $t \in T$ is associated to its set of accessed files F_t. Notably, a test trace does *not* imply the order in which files are accessed, but only constitutes the set of accessed files during execution.

3.3.3 Integration. We implemented ProST as a command line program which takes an arbitrary user command as input, e.g., `mvn test`. If the test event listener is added to the system under test, this will output both, tracing and testing log. ProST then converts these log files into a structured JSON output file, containing the file-level test traces.

One important aspect of test execution is parallelization. Similar to `RTSLinux`, ProST supports tracing parallel execution of tests through multi-processing (i.e., running multiple processes with one test per process), as it transitively follows all child processes of the user command and thereby collects all file accesses for each test [15].

Since `DTrace` and `bpftrace` provide far more probes (beyond system call instrumentation points), ProST is easily extensible to trace further events than file accesses during test execution. Examples are events related to socket communication or multi-threading which can be useful to detect flaky tests or prioritize regression tests. We discuss extension points and use cases in Sec. 4.5.

4 EVALUATION

To evaluate the effectiveness and efficiency of ProST, we perform an empirical study on the multi-language software projects introduced in Sec. 2. We strive to answer the following research questions (RQs):

- **RQ$_1$:** How many more files are included in ProST's test traces compared to language-specific instrumentation?
- **RQ$_2$:** How efficient is ProST in terms of introduced execution time overhead compared to language-specific instrumentation?

Since ProST aims to be practical and portable, we discuss the transferability of our evaluation results to other operating systems than Linux and other language environments than the JVM in Sec. 4.5. All experiment results and scripts are part of our reproduction package.

4.1 Study Subjects

During our motivating pre-study, we identified 25 multi-language software projects from open-source development listed in Table 2. We have outlined the project selection criteria in Sec. 2. Notably, as previously stated, all projects use one GPL that runs on the JVM, which is one of the threats to validity of our study (see Sec. 4.6).

Throughout the course of our empirical study, we encountered problems with the projects `joyqueue` and `tinkerpop`: `joyqueue` has fluctuating test cleanup times due to complex locking and multi-threading mechanisms, which would lead to distorted test execution runtimes in RQ$_2$. For `tinkerpop`, ProST's JUnit test listener was not applicable due to incompatibility issues with a custom JUnit runner that `tinkerpop` uses for several tests. Thus, only 23 out of the initial 25 projects could be included for the sake of this evaluation.

4.2 Comparing Language-specific Techniques

Since all of our projects use a JVM programming language as their main GPL (see Sec. 2), we need to compare ProST to language-specific instrumentation techniques that target the JVM. We employ one publicly available coverage tool, `JaCoCo`[10], that supports collecting test traces on the JVM (also called *per-test coverage*) and has been used in prior studies on regression test optimization [10, 13, 58]. Additionally, we implemented a straightforward file-level instrumentation technique for the JVM, `ClassInst`. The reason why we implement a language-specific technique ourselves and do not only rely on `JaCoCo`, is that `JaCoCo` is not specifically designed to collect file-level test traces. It collects more fine-grained execution information and offers additional runtime features, such as a Web server to interact with `JaCoCo` during test execution. Since these features lead to higher overhead during test execution, comparing ProST only to `JaCoCo` would be unfair. Thus, we compare ProST to both, an existing tool and a *raw* instrumentation technique.

Notably, similar to ProST, `JaCoCo` and `ClassInst` also require a simple JUnit test execution listener to link covered Java `.class` files to tests.

4.2.1 JaCoCo. `JaCoCo` is a popular code coverage library for the JVM that instruments bytecode during runtime using the ASM bytecode manipulation framework[11] and the Java Agent API [1]. With single-threaded execution of tests, `JaCoCo` allows to collect per-test coverage by dumping collected coverage after a test has been run (e.g., triggered by a test execution listener) [10, 13]. Similar

[9] Similar to prior regression testing research, we consider one *test* as one test suite (i.e., test class) [22, 29, 69, 76].

[10] `JaCoCo`: https://www.eclemma.org/jacoco/

[11] ASM bytecode manipulation framework: https://asm.ow2.io

to [58], we use a `JaCoCo` wrapper[12] that facilitates collecting per-test coverage. This wrapper collects statement coverage by default. However, per-test statement coverage information can simply be aggregated to file-level test traces.

4.2.2 Language-specific File-level Instrumentation. Regression test selection tools such as `Ekstazi` [28, 29] or `HyRTS` [78] internally rely on file-level instrumentation techniques. Yet, they are specifically designed for regression test selection and either do not offer an interface to only collect test traces or do not support JDK or JUnit versions newer than 8 or 4, respectively. Hence, we cannot directly apply them in our study, as the majority of projects (16/23) uses a newer JDK version. Thus, we implemented a simple language-specific instrumentation approach (`ClassInst`) that is based on the same underlying concepts of dynamic code instrumentation using the Java Agent API. Similar to `Ekstazi`, `HyRTS`, and `JaCoCo`, `ClassInst` intercepts class loading into the JVM during *runtime*. Thus, when a class or a Java Archive (JAR) containing a class is loaded into the JVM, its corresponding `.class` or `.jar` file path is printed into a log file (i.e., *tracing log*). Due to the JVM's lazy class loading, this instrumentation will only cover all `.class` files, that are actually used by the test that is executed.

4.3 Experiment Execution

For the experiments, we reuse the two shell scripts, `build.sh` and `test.sh` (see Sec. 2.2) for building and testing the project. Recall that this allows us to separate the build process from the testing process. The experiments are conducted as follows: We iterate over all 23 projects and first build the project. Second, similar to Celik et al. [15], we add the Maven Surefire test options `-DforkCount=1` and `-DreuseForks=false` to the test script (`test.sh`). This instructs Maven Surefire to run tests in isolation in a separate JVM which removes effects of JVM caching or parallel execution (partly not supported by `JaCoCo`, see Sec. 4.2.1) for the sake of our experiments. While this may introduce additional overhead for JVM forking, it is a common approach used in industry and the default in several of the studied projects [9, 15, 57]. In fact, when collecting test traces, it is beneficial to use JVM forking in general, as it increases the reliability of test results by preventing shared test state pollution or test-order dependencies [9, 15, 57].

We then execute the adjusted `test.sh` script with ProST, `JaCoCo`, and `ClassInst` instrumentation and measure the wall-clock end-to-end test execution time for each of them. Additionally, we execute the tests once without instrumentation, only including the JUnit test execution listener that is needed for all instrumentation techniques (`NoInst`).

We perform our experiments on a machine with an Intel® Core™ i7-6700 processor containing 8 logical cores which run at 3.4 GHz, 16 GB main memory, and running the GNU/Linux Ubuntu 20.04 (64 bit) operating system.

4.4 Results

4.4.1 RQ_1: Test Trace Comparison. Table 3 contains the number of files that are part of the test traces for each studied project as collected by ProST, `JaCoCo`, and `ClassInst`. Notably, `ClassInst` and `JaCoCo` are based on the same dynamic instrumentation approach using the Java Agent API and therefore cover the same `.class` files that were loaded into the JVM. On average, 89.2% (335) more files are collected by ProST, indicating that ProST's test traces are more complete and language-specific techniques alone would miss per-test file dependencies in the studied projects. The measured average number of file dependencies per test (class) increases by a factor of 3.5. It is important to note that not only the numbers are higher, but also that all files tracked by `JaCoCo` and `ClassInst` are also covered by ProST. If ProST's test traces are used in the context of regression test selection, this enables higher safety than using `JaCoCo` or `ClassInst` [15].

In general, this supports prior regression testing research, where language-specific techniques only collected 16.99% of the files collected with system call tracing [15]. However, while these findings imply a factor of 5.89 more file dependencies, we only collect roughly a factor of 0.9 (89%) more files with ProST. We expect these differences to stem from the fact that Celik et al. [15] do not filter out accessed directories, which, in Linux, are considered files, and do not exclude files generated by the build or testing frameworks as well as system and JDK-related files. Yet, we believe that applying these filters is reasonable, since these files are not tracked by the version control system and, depending on the system, can be out of the developers control (e.g., continuous integration build machines).

> **RQ_1** *We find that in the considered projects, test traces collected with ProST contain all files found with language-specific techniques and on average 89.2% more files.*

4.4.2 RQ_2: Efficiency. In order to calculate the introduced execution overhead by `ClassInst`, ProST, and `JaCoCo`, we measure the wall-clock end-to-end test execution time for each project in our experimental Linux setup. The rightmost four columns of Table 3 depict these times for each technique including `NoInst`. The average introduced test execution time overhead are 7.4% for `ClassInst`, 4.6% for ProST, and 286.5% for `JaCoCo`. While the comparatively high overhead introduced by `JaCoCo` is expected, as it instruments Java bytecode at statement level, `ClassInst` has significant smaller overhead. This emphasizes that it is fairer to compare to `ClassInst` than to `JaCoCo`. ProST has an even slightly smaller overhead than `ClassInst` (2.8 pp). Despite this efficiency, ProST collects all files covered by `ClassInst` and `JaCoCo` and additional 89.2% files.

> **RQ_2** *We find that ProST adds an execution time overhead of 4.6% on top of non-instrumented test execution. This overhead is smaller than with lightweight language-specific instrumentation (7.4%).*

4.5 Discussion

In the following, we discuss the results, transferability and implementation aspects, as well as use cases and extensions for ProST.

4.5.1 Transferability to other languages and frameworks. We have alluded to why system call tracing is a language-agnostic approach

[12] `JaCoCo` wrapper: https://github.com/cqse/teamscale-jacoco-agent

Table 3: Empirical evaluation results for RQ_1 and RQ_2 for the studied projects (excludes 2 discarded projects)

Project	Number of distinct files in test traces (RQ_1)				Test execution time [sec] (RQ_2)			
	`ClassInst` / `JaCoCo`		ProST		`NoInst`	`ClassInst`	ProST	`JaCoCo`
	Total	Avg./Test	Total	Avg./Test				
anserini	481	31.96	1,949	213.04	160.31	177.27	166.08	2,570.32
cometd	1,213	53.07	1,343	70.47	2,264.20	2,304.74	2,268.18	2,632.76
consulo	555	12.96	641	67.00	154.87	160.59	158.87	327.25
enunciate	213	16.94	405	92.94	23.76	25.27	25.10	39.71
fess	1,303	104.78	1,747	359.48	164.92	172.67	168.96	267.28
graphwalker-project	486	37.56	1,112	62.41	103.78	115.18	106.81	293.86
guacamole-client	104	11.77	149	22.92	6.59	7.53	8.03	24.31
hopsworks	174	20.80	434	227.87	16.13	16.88	17.22	44.00
incubator-streampipes	262	15.20	490	114.64	50.93	55.07	52.91	111.23
IPED	56	6.86	300	249.95	10.53	11.84	12.14	42.72
jzmq	68	12.27	72	13.55	12.63	11.91	13.20	28.72
languagetool	1,715	44.77	2,493	127.17	1,440.13	1,494.65	1,473.34	2,367.55
lavagna	537	132.89	662	261.20	151.66	161.76	153.63	280.56
openmeetings	941	156.13	1,578	806.43	2,611.11	2,862.62	2,792.20	2,928.79
primefaces	368	22.60	402	64.83	48.13	51.51	50.95	102.84
psi-probe	497	20.47	584	101.78	82.86	91.98	86.33	267.47
redpen	279	67.89	325	128.70	127.88	142.68	131.25	258.10
searchcode-server	185	25.51	966	86.31	36.08	38.24	37.17	69.71
servicecomb-java-chassis	3,571	51.43	3,882	130.04	3,502.01	3,580.96	3,484.01	4,718.90
servicecomb-pack	646	64.05	872	159.16	292.26	313.84	295.89	374.53
sonar-php	1,391	82.29	1,726	105.60	240.32	279.51	249.65	1,040.06
steady	708	28.36	1,188	111.26	493.82	502.48	502.84	624.78
webprotege	1,699	16.96	1,834	98.78	426.23	494.24	437.96	1,444.52
Avg.	758.78	45.11	1,093.65	159.81	540.05	568.41	551.86	906.96
$\sum$	17,452.00	1,037.52	25,154.00	3,675.53	12,421.14	13,073.42	12,692.72	20,859.97

to collect dynamic per-test dependencies at file-level granularity. Nonetheless, there are language-specific peculiarities that affect the effectiveness of ProST.

Considering the JVM, syscall tracing can be imprecise, if accessed `.class` files reside inside JARs, as only the accessed JAR will be part of the test trace [29]. This limitation applies to ProST as well. Notably, in contrast to ProST, both `JaCoCo` and `ClassInst` can track which `.class` files *inside* a JAR were accessed.

If ProST is used on C/C++ software, where source files are compiled to (often large) binary files, ProST's test traces might be too coarse grained for precise per-test analysis. Still, as opposed to language-specific instrumentation, ProST will cover all non-code artifacts and file accesses made by spawned subprocesses. ProST can thus be applied on top of existing language-specific instrumentation techniques, e.g., augment established code coverage-based techniques.

Furthermore, interpreted languages such as Python or JavaScript use different implementations for lazy loading and evaluation of source files than the JVM. In the worst case, this may lead to all source files to be opened that are imported by a test—even if no code from these files is actually used by the test [15].

Overall, while ProST is applicable to arbitrary language environments, it definitely has limitations with respect to precision for languages that compile to large binary files or with interpreted languages. We believe that ProST is specifically useful in scenarios where cross-language links are present and tests intensively make use of non-code artifacts.

4.5.2 Portability to other operating systems. To evaluate whether our results for RQ_2 hold for different operating systems, we repeated our efficiency analysis again for a subset of five randomly sampled projects, that were also buildable and testable on Windows and macOS, namely `anserini`, `enunciate`, `lavagna`, `primefaces`, `searchcode-server`. On those projects, ProST introduced on average 3.9% (Linux), 4.1% (macOS), and 7.6% (Windows) execution time overhead. Hence, while the measured execution overhead is indeed affected by the operating system and ProST has a slightly smaller overhead on Linux on the project sample (3.9% versus 4.6% on all projects), we can observe that ProST introduces relatively low overhead of <10% on all three operating systems.

4.5.3 Running one test per process. In our experiments, we rely on Maven Surefire's JVM forking mechanism to run each test in its own process. While we have discussed that this is a common practice as it leads to more stable test results, it might not be supported by an arbitrary testing framework for a given programming language. Furthermore, such process forking might be undesired as it implies additional overhead and removes caches between tests, such as the JVM class loader cache. Yet, even in those cases, ProST can provide significant benefits: As long as the testing framework allows registering a test event listener, which is a requirement for any

per-test dependency analysis, one can also use ProST on top of an existing language-specific instrumentation. More precisely, if a project already uses a language-specific technique to obtain test traces, adding ProST can enrich these test traces by non-code or multi-language artifacts. In fact, we are combining ProST with lightweight Java bytecode instrumentation at our industry partner, in order to gain more fine-grained test traces.

4.5.4 Writing to log files. As described, ProST and `ClassInst` currently log every file access directly into a tracing log file. While this is not the most efficient way to store these data, it allows us to measure the actual instrumentation overhead in isolation, regardless of the used storing mechanism (e.g., in-memory storage or storing accessed files directly into a database). Furthermore, this way, we circumvent multi-threading problems with synchronized code blocks, but simply rely on the operating system's file locking mechanism. Hence, for the sake of comparing actual instrumentation overhead, we deem the log file approach to be reasonable.

4.5.5 Use Cases and Extensions for ProST. As ProST uses generic kernel instrumentation frameworks, it can be applied and extended for several use cases which we discuss here.

Regression Test Selection. Test traces obtained with system call tracing have been used for regression test selection by Celik et al. [15]. To select tests, they compute checksums for each file in a test's trace and check if any of those checksums is affected by changes to the code base. This approach is also applicable to test traces recorded with ProST. In addition, ProST can be easily extended to instrument further available probes. These include, for instance, instrumentation points specifically for the JVM[13]: `DTrace` provides probes to instrument any class loading activity in the JVM, which can be used to also trace accesses to `.class` files packaged into `.jar` files. Thus, through small context-specific additional instrumentation, we can make test traces obtained with ProST more precise (i.e., fine-grained `.class` instead of coarse-grained `.jar` accesses). This can in turn lead to better results for regression test selection techniques [29].

We are successfully using ProST to create test traces used for regression test selection at our industry partner IVU in a large-scale (>20M LOC) multi-language software project that targets the Windows operating system.

Test Case Prioritization. There are plenty of coverage-based test case prioritization techniques which use more fine-grained test traces (e.g., method or statement level) [44, 68] to rank tests: Tests with more (additional) coverage should thereby be executed first. There are, however, no test case prioritization techniques available that incorporate non-code artifact coverage or coverage of source files in other languages. Yet, as we have shown in Sec. 2.3, tests commonly make use of such artifacts in multi-language software. Test traces obtained with ProST provide all information necessary to rank tests also based on those criteria. Furthermore, ProST contains an experimental implementation of tracing socket interaction that tracks whenever a socket is opened from any currently traced process (i.e., test). The number of traced per-test socket interactions can also be used to prioritize tests, as tests that make network calls are commonly prone to fail [62].

Flaky Test Detection and Classification. Flaky tests are tests that may non-deterministically fail and pass with the same program version. They are a well-known problem in regression testing [10, 41, 45, 64] and a potential threat for evaluating regression test optimization techniques [61]. To decrease debugging costs it is thus crucial to detect flaky tests, understand the root causes for flakiness, and, if possible, repair the tests [45]. File-related input/output is one common root cause for flakiness, that can be analyzed by using test traces from ProST. Flakiness related to concurrency or networking problems [45] could also be diagnosed with ProST by instrumenting more probes related to networking or multi-threading (see previous paragraph). We are currently experimenting with using ProST for detecting and classifying flaky tests.

4.6 Threats to Validity

4.6.1 External Validity. The main threats to external validity stem from the representativeness of our empirical results. We address them by analyzing a set of popular, open-source multi-language, yet only JVM-based, projects from different domains, but cannot, by nature of an empirical evaluation, easily generalize the findings beyond these projects.

Another threat is that six of the studied projects have relatively short test runtimes of <1 minute which arguably questions whether test optimization would be necessary in these projects. However, while these projects may not necessarily require test selection, we have discussed further use cases for test traces in Sec. 4.5.5, such as failure diagnosis or flakiness root cause classification. In fact, the approach proposed in this paper explicitly does *not* target a specific use case, but introduces a universal, yet efficient and practical analysis technique based on probe-based system call tracing. Moreover, even in prior studies on regression test selection, 9 of 21 studied projects had test runtimes of <1 minute [15].

Finally, as described in Sec. 4.1, we exclude two projects from our initial motivating pre-study for the main study, as they provide unreliable evaluation results. While this does not preclude applicability of ProST to these projects, other projects could pose similar challenges to engineers. Yet, such compatibility issues are a general threat for any instrumentation technique that needs to be hooked into the build and testing frameworks and are a known problem for existing regression test selection tools as well [81].

4.6.2 Internal Validity. The main internal threats to validity stem from the implementation of ProST, `ClassInst`, and our evaluation scripts used within our empirical study. To check `ClassInst`'s correctness, we compared the `.class` files covered by `ClassInst` with those traces by `JaCoCo`, a widespread Java instrumentation tool, and did not find any errors. ProST relies on the kernel instrumentation frameworks `DTrace` and `bpftrace`. During our experiments and the operation at our industry partner, we found that `DTrace` on Windows occasionally produces probe errors, which in our case did not lead to any missed accessed files. `bpftrace` currently only

[13] `DTrace` for Java 8: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/dtrace.html

supports strings with up to 200 characters, which caused problems in a few of the studied projects[14]. ProST therefore contains its own compiled binary of `bpftrace` to support strings with up to 220 characters and an updated timestamp API, which has recently been added to `bpftrace`, but not yet released. To address further implementation-related threats, we wrote unit tests and runtime assertions while developing ProST and manually checked test traces for their validity.

5 RELATED WORK

We have referenced work from different research areas throughout this paper. Among these areas, we consider multi-language software and file-level test traces in regression testing to be most relevant for our approach.

5.1 Multi-language Software

Mayer et al. [52–54] investigate multi-language software development in open-source and industrial settings. By statically analyzing source code repositories and interviewing developers, they find that software projects use on average five to seven languages. Their results indicate that XML, JSON, and YAML are among the most used DSLs, which we confirm in our pre-study, albeit with dynamic analysis of tests [53]. Furthermore, the interviewed developers report Java/XML and Java/JavaScript to be the most common language combinations which also matches our findings [54].

Bigliardi et al. [11] quantitatively analyze the prevalence and development of non-code software artifacts in open-source repositories. Their results show that on average 50% of files in each project are non-code artifacts and that one third of all commits includes changes to these artifacts. In our pre-study even 88% of the studied multi-language projects use non-code artifacts during testing.

Kochhar et al. [40] study the impact of multiple languages on the quality of software. In their large scale study of open-source projects, they find that in projects with more languages, the defect proneness is significantly increased. We, on the other hand, do not investigate bug proneness, but study what parts of multi-language software are actually executed during regression testing.

5.2 File-level Test Traces in Regression Testing

Gligoric et al. [28, 29] propose the first dynamic file-level regression test selection technique, `Ekstazi`. `Ekstazi` relies on Java bytecode instrumentation and reduces the end-to-end testing time on average by 32% across 32 open-source projects. Furthermore, the authors show that test traces at file granularity are more effective than more fine-grained traces.

Shi et al. [69] combine `Ekstazi` with the static incremental Maven build tool `GIB` [4]. The resulting tool, `GIBstazi`, selects an entire Maven module for testing, in case any non-code artifacts are changed. To measure the impact of such non-code artifacts on the test selection, the authors apply an `strace`-based tool to see whether any tests access non-code artifacts and how often these non-code artifacts have been changed in the version control history. They find that on average 7.5% of commits contain non-code changes that are relevant for testing.

[14]There is a pending pull request that solves the string length limitation issue: https://github.com/iovisor/bpftrace/pull/1360

Celik et al. [15] propose `RTSLinux`, the first and only regression test selection technique that uses system call analysis to trace arbitrary file accesses during testing. `RTSLinux` thereby collects roughly 6 times more accessed files than `Ekstazi` and saves 53% of test execution time compared to a retest-all strategy. However, `RTSLinux` is only applicable in Linux environments and taints the Linux kernel through a custom kernel module. This is arguably impractical in an industrial context, as this implies maintaining that extension for future kernel versions with utmost care to avoid kernel panics. In contrast, ProST uses safe kernel instrumentation techniques available for all major operating systems and, similar to `RTSLinux`, is highly efficient as it also operates in kernel rather than user mode.

In summary, none of existing studies on multi-language software projects analyze how intensively such software is tested across language boundaries by applying dynamic program analysis. Moreover, to the best of our knowledge, there are no studies on regression testing that specifically investigate how to efficiently collect file-level test traces through system call tracing beyond Linux environments.

6 CONCLUSION

Dynamic regression test optimization techniques require per-test execution traces to analyze test runtime behavior and reduce testing effort or feedback time. However, these test traces are typically collected through language-specific code instrumentation, which ignores cross-language boundaries and non-code artifacts. Tracing operating system calls yields more complete test traces by collecting all accessed files during test execution. However, existing approaches are impractical in most contexts due to transferability, maintainability, security, or performance concerns.

In this paper, we propose ProST, a novel system call analysis approach to collect file-level test traces. ProST harnesses probe-based kernel instrumentation frameworks to instrument all relevant file- and process-related system calls. The kernel instrumentation runs in a safe execution environment and currently supports Linux, Windows, and macOS. We empirically study the efficiency and effectiveness of ProST on 25 open-source multi-language software projects. The results show that in 96% (24) of the studied projects tests access non-code artifacts or source files from multiple languages that are missed by language-specific instrumentation techniques. We further find that ProST introduces only 4.6% of execution time overhead during testing (compared to 7.4% overhead with language-specific techniques) and collects on average 89% more accessed files on top of those collected by language-specific instrumentation.

ProST can thus be used as an efficient, practical, yet more complete approach for obtaining file-level test traces or complement existing language-specific instrumentation to analyze multi-language software systems.

ACKNOWLEDGMENTS

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant SOFIE 01IS18012B. The responsibility for this article lies with the authors.

REFERENCES

[1] 2017. Java Agent API. https://docs.oracle.com/javase/9/docs/api/java/lang/instrument/package-summary.html

[2] 2021. Advanced googletest Topics. https://google.github.io/googletest/advanced.html{#}defining-event-listeners

[3] 2021. Configuring Jest. https://jestjs.io/docs/configuration{#}reporters-arraymodulename--modulename-options

[4] 2021. gitflow-incremental-builder (GIB). https://github.com/gitflow-incremental-builder/gitflow-incremental-builder

[5] 2021. strace - trace system calls and signals. http://linux.die.net/man/1/strace

[6] Apache Maven. 2021. Maven Build Lifecycle. https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

[7] Apache Maven. 2021. Maven Surefire Plugin – surefire:test. https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html

[8] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt, and Christian Stein. 2021. JUnit 5 User Guide: Advanced Topics. https://junit.org/junit5/docs/current/user-guide/{#}launcher-api-listeners-custom

[9] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *Proceedings of the International Conference on Software Engineering*. 550–561. https://doi.org/10.1145/2568225.2568248

[10] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. *Proceedings of the International Conference on Software Engineering* (2018), 433–444. https://doi.org/10.1145/3180155.3180164

[11] Luca Bigliardi, Michele Lanza, Alberto Bacchelli, Marco Dambros, and Andrea Mocci. 2014. Quantitatively exploring non-code software artifacts. In *Proceedings of the International Conference on Quality Software*. 286–295. https://doi.org/10.1109/QSIC.2014.31

[12] Ben Boyter. 2021. Sloc Cloc and Code (scc). https://github.com/boyter/scc

[13] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: An industrial case study. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 975–980. https://doi.org/10.1145/2950290.2983954

[14] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. https://doi.org/10.5555/1247415.1247417

[15] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression test selection across JVM boundaries. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 809–820. https://doi.org/10.1145/3106237.3106297

[16] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing test prioritization via test distribution analysis. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 656–667. https://doi.org/10.1145/3236024.3236053

[17] Carmen Coviello, Simone Romano, Giuseppe Scanniello, Alessandro Marchetto, Giuliano Antoniol, and Anna Corazza. 2018. Clustering support for inadequate test suite reduction. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. 95–105.

[18] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. 2019. Scalable Approaches for Test Suite Reduction. In *Proceedings of the International Conference on Software Engineering*. 419–429.

[19] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2002. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering* 28, 2 (2002), 159–182. https://doi.org/10.1109/32.988497

[20] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. 2004. Selecting a cost-effective test case prioritization technique. *Software Quality Journal* 12, 3 (2004), 185–210. https://doi.org/10.1023/b:sqjo.0000034708.84524.22

[21] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 235–245. https://doi.org/10.1145/2635868.2635910

[22] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. In *Proceedings of the International Symposium on Software Testing and Analysis*. 491–504. https://doi.org/10.1145/3460319.3464834

[23] Daniel Elsner, Roland Wuersching, Markus Schnappinger, and Alexander Pretschner. 2022. Supplemental Material for: Probe-based Syscall Tracing for Efficient and Practical File-level Test Traces. (2022). https://doi.org/10.6084/m9.figshare.16811518.v2

[24] Kurt Fischer, Farzad Raji, and Andrew Chruscicki. 1981. A Methodology for Retesting Modified Software. In *Proceedings of the National Telecommunications Conference*. 1–6.

[25] Kurt F. Fischer. 1977. A test case selection method for the validation of software maintenance modifications. In *Proceedings of International Computer Software and Applications Conference*. 421–426.

[26] Matt Fleming. 2017. A thorough introduction to eBPF. https://lwn.net/Articles/740157/

[27] Ben Fu, Sasa Misailovic, and Milos Gligoric. 2019. Resurgence of Regression Test Selection for C++. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. 323–334. https://doi.org/10.1109/ICST.2019.00039

[28] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *Proceedings of the International Conference on Software Engineering*. 713–716. https://doi.org/10.1109/icse.2015.230

[29] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis*. 211–222. https://doi.org/10.1145/2771783.2771784

[30] Brendan Gregg. 2016. DTrace for Linux. http://www.brendangregg.com/blog/2016-10-27/dtrace-for-linux-2016.html

[31] Brendan Gregg. 2018. bpftrace (DTrace 2.0) for Linux. http://www.brendangregg.com/blog/2018-10-08/dtrace-for-linux-2018.html

[32] Brendan Gregg. 2019. *BPF Performance Tools*. Addison-Wesley Professional.

[33] Brendan Gregg. 2019. Learn eBPF Tracing: Tutorial and Examples. http://www.brendangregg.com/blog/2019-01-01/learn-ebpf-tracing.html

[34] Brendan Gregg and Jim Mauro. 2011. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional.

[35] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2, 3 (1993), 270–285. https://doi.org/10.1145/152388.152391

[36] Mary Jean Harrold, Alessandro Orso, James A. Jones, Tongyu Li, Maikel Pennings, Saurabh Sinha, Ashish Gujarathi, Donglin Liang, and S. Alexander Spoon. 2001. Regression test selection for Java software. *ACM SIGPLAN Notices* 36, 11 (2001), 312–326. https://doi.org/10.1145/504311.504305

[37] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. 1994. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the Scalable High-Performance Computing Conference*. 841–850. https://doi.org/10.1109/shpcc.1994.296728

[38] JUnit. 2021. JUnit 5. https://junit.org/junit5

[39] Nikhil Khadke, Michael Kasick, Soila Kavulya, Jiaqi Tan, and Priya Narasimhan. 2012. Transparent System Call Based Performance Debugging for Cloud Computing. In *2012 Workshop on Managing Systems Automatically and Dynamically (MAD 12)*.

[40] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. 2016. A large scale study of multiple programming languages and code quality. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. 563–573. https://doi.org/10.1109/SANER.2016.112

[41] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the International Symposium on Software Testing and Analysis*. 101–111. https://doi.org/10.1145/3293882.3330570

[42] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. In *Proceedings of the International Conference on Automated Software Engineering*. 949–954. https://doi.org/10.1109/ase.2017.8115710

[43] Hareton K.N. Leung and Lee White. 1989. Insights into regression testing. In *Proceedings of the International Conference on Software Maintenance*. 60–69.

[44] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How does regression test prioritization perform in real-world software evolution?. In *Proceedings of the International Conference on Software Engineering*. 535–546. https://doi.org/10.1145/2884781.2884874

[45] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the Symposium on the Foundations of Software Engineering*. 643–653. https://doi.org/10.1145/2635868.2635920

[46] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018

[47] Federico Maggi, Matteo Matteucci, and Stefano Zanero. 2010. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing* 7, 4 (2010), 381–395.

[48] Alexey G. Malishevsky, Gregg Rothermel, and Sebastian Elbaum. 2002. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Press, 230–240. https://doi.org/10.1109/icsm.2002.1167767

[49] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test case prioritization for continuous regression testing: An industrial case study. In *Proceedings of the International Conference on Software Maintenance*. 540–543. https://doi.org/10.1109/icsm.2013.91

[50] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. In *Proceedings of the Conference on Mining Software Repositories*. 385–396. https://doi.org/10.1145/3379597.3387458

[51] Apache Maven. 2021. Maven Failsafe Plugin – integration-test. https://maven.apache.org/surefire/maven-failsafe-plugin/integration-test-mojo.html

[52] Philip Mayer. 2017. A taxonomy of cross-language linking mechanisms in open source frameworks. *Computing* 99 (2017), 701–724. https://doi.org/10.1007/s00607-016-0528-3

[53] Philip Mayer and Alexander Bauer. 2015. An empirical analysis of the utilization of multiple programming languages in open source projects. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*. Nanjing, China, 1–10. https://doi.org/10.1145/2745802.2745805

[54] Philip Mayer, Michael Kirsch, and Minh Anh Le. 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development* 5, 1 (2017). https://doi.org/10.1186/s40411-017-0035-z

[55] Steven McCanne and Van Jacobson. 1993. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Conference*.

[56] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. 2011. Regression testing in the presence of non-code changes. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*. 21–30. https://doi.org/10.1109/icst.2011.60

[57] Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. 2020. Debugging the Performance of Maven's Test Isolation: Experience Report. In *Proceedings of the International Symposium on Software Testing and Analysis*. 249–259. https://doi.org/10.1145/3395363.3397381

[58] Raphael Noemmer and Roman Haas. 2020. An Evaluation of Test Suite Minimization Techniques. In *Software Quality: Quality Intelligence in Software and Systems Engineering*, D Winkler, S Biffl, D Mendez, and J Bergsmann (Eds.). Vol. 371. Springer,, 51–66. https://doi.org/10.1007/978-3-030-35510-4_4

[59] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 241–251. https://doi.org/10.1145/1029894.1029928

[60] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi Min Wang. 2011. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the Conference on Computer Systems*. 153–168.

[61] Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*. 324–336. https://doi.org/10.1145/3395363.3397383

[62] Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Nachiappan Nagppan. 2019. FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services. In *Proceedings of the International Conference on Software Engineering*. 408–418. https://doi.org/10.1109/icse.2019.00054

[63] Hari Pulapaka, Andrey Shedel, and Gopikrishna Kannan. 2019. DTrace on Windows. https://techcommunity.microsoft.com/t5/windows-kernel-internals/dtrace-on-windows/ba-p/362902

[64] Yihao Qin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F. Bissyandé. 2021. On the Impact of Flaky Tests in Automated Program Repair. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. 295–306. https://doi.org/10.1109/SANER50967.2021.00035

[65] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210. https://doi.org/10.1145/248233.248262

[66] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. 2000. Regression test selection for C++ software. *Software Testing, Verification and Reliability* 10, 2 (2000), 77–109. https://doi.org/10.1002/1099-1689(200006)10:2<77::AID-STVR197>3.0.CO;2-E

[67] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. 1998. An Empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Press, 34–43. https://doi.org/10.1109/icsm.1998.738487

[68] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: an empirical study. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Press, 179–188. https://doi.org/10.1109/icsm.1999.792604

[69] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *Proceedings of the International Symposium on Software Reliability Engineering*. 228–238. https://doi.org/10.1109/issre.2019.00031

[70] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the International Symposium on Software Testing and Analysis*. 12–22. https://doi.org/10.1145/3092703.3092709

[71] Andrew S. Tanenbaum and Herbert Bos. 2015. *Modern operating systems*. Pearson.

[72] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. 1998. Effect of test set minimization on fault detection effectiveness. *Software Prac. Experience* 28, 4 (1998), 347–369. https://doi.org/10.1002/(SICI)1097-024X(19980410)28:4<347::AID-SPE145>3.0.CO;2-L

[73] W. Eric Wong, Joseph R. Horgan, Aditya P. Mathur, and Alberto Pasquini. 1999. Test set size minimization and fault detection effectiveness: A case study in a space application. *Journal of Systems and Software* 48, 2 (1999), 79–89. https://doi.org/10.1016/S0164-1212(99)00048-5

[74] Tobias Wüchner, Martín Ochoa, and Alexander Pretschner. 2015. Robust and effective malware detection through quantitative data flow graph metrics. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Vol. 9148. 98–118.

[75] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Software Testing Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430

[76] Tingting Yu and Ting Wang. 2018. A Study of Regression Test Selection in Continuous Integration Environments. In *Proceedings of the International Symposium on Software Reliability Engineering*. 135–143. https://doi.org/10.1109/ISSRE.2018.00024

[77] Tarannum Shaila Zaman, Xue Han, and Tingting Yu. 2019. SCMiner: Localizing system-level concurrency faults from large system call traces. In *Proceedings of the International Conference on Automated Software Engineering*. 515–526. https://doi.org/10.1109/ASE.2019.00055

[78] Lingming Zhang. 2018. Hybrid regression test selection. In *Proceedings of the International Conference on Software Engineering*. 199–209. https://doi.org/10.1145/3180155.3180198

[79] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2013. FaultTracer: A spectrum-based approach to localizing failure-inducing program edits. *Journal of Software: Evolution and Process* 25, 12 (2013), 1357–1383. https://doi.org/10.1002/smr.1634

[80] Hua Zhong, Lingming Zhang, and Sarfraz Khurshid. 2019. TestSage: Regression test selection for large-scale web service testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. 430–440. https://doi.org/10.1109/icst.2019.00052

[81] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *Proceedings of the International Conference on Software Engineering*. 430–441.

[82] Celal Ziftci and Diego Cavalcanti. 2020. De-Flake Your Tests: Automatically Locating Root Causes of Flaky Tests in Code at Google. In *Proceedings of the International Conference on Software Maintenance and Evolution*. 736–745. https://doi.org/10.1109/ICSME46990.2020.00083

[83] Jörg Zinke. 2009. System call tracing overhead. http://www.linux-kongress.org/2009/slides/system{_}call{_}tracing{_}overhead{_}joerg{_}zinke.pdf