BinaryRTS: Cross-language Regression Test Selection for C++ Binaries in CI

Daniel Elsner
Technical University of Munich
Munich, Germany

Severin Kacianka
Technical University of Munich
Munich, Germany

Stephan Lipp
Technical University of Munich
Munich, Germany

Alexander Pretschner
Technical University of Munich
Munich, Germany

Axel Habermann
IVU Traffic Technologies
Berlin, Germany

Maria Graber
IVU Traffic Technologies
Berlin, Germany

Silke Reimer
IVU Traffic Technologies
Berlin, Germany

***Abstract*—Continuous integration (CI) pipelines are commonly used to execute regression tests before pull requests are merged. Regression test selection (RTS) aims to reduce the required testing effort and feedback time for developers. However, existing RTS techniques are imprecise for tests with cross-language links to compiled C++ binaries or unsafe if tests use external files. This is problematic because modern software in fact involves several programming languages and (non-)code artifacts such as configuration files. In this paper, we present BinaryRTS, a novel RTS technique that leverages dynamic binary instrumentation to collect the covered functions and accessed external files for each test. BinaryRTS then selects tests depending on changes issued to C++ binaries or external (non-)code artifacts. When evaluating BinaryRTS in our large-scale industrial context, we are able to exclude on average up to 74% of tests without missing real failures. We release BinaryRTS as the first publicly available RTS tool for software involving C++ code.**

***Index Terms*—Software testing, regression test selection, C++, cross-language links, multi-language software, non-code artifacts**

I. Introduction

Regression testing is a software testing activity that checks if changes have negatively impacted existing system behavior [1]. In modern development practices, continuous integration (CI) pipelines are commonly used to regularly build the software and run its regression test suite [2]–[4]. The most straightforward testing strategy is *retest-all*, which executes every test case after each change. However, if fast feedback for developers is crucial or testing resources are limited, executing all tests from a large test suite is often prohibitively costly [5], [6]. To address this problem, regression test selection (RTS) has been studied since the 1970s [7] to reduce the testing effort by only running a subset of test cases [2]–[4], [8]–[17]. An RTS technique is considered *safe*, if this subset of test cases contains all tests that potentially expose a fault [18].

At IVU Traffic Technologies[1], CI pipelines execute a regression test suite consisting of more than 25,000 unit, integration, and system tests written in C++ and Java before pull requests are merged into release branches. However, running the full test suite for each pull request yields intolerable feedback times of several hours, despite a high degree of test parallelization. Therefore, we have developed and successfully deployed a file-level RTS technique for Java tests at IVU in prior work [19]. Yet, due to the complex nature of the multi-language code base at IVU, two problems remained unsolved: first, the majority of the 13.5 million lines of code (LOC) and the test suite is written in C++. These C++ regression tests are not supported by our current RTS solution, as file-level techniques are impractical for languages that compile to large binary files [20]. Second, there exist several thousand Java tests that use the Java Native Interface (JNI) to interact with dynamic-link libraries (DLLs) built from the C++ code. Hence, if any C++ source file that is part of a DLL changes, every test accessing this binary file is selected. In short, file-level per-test execution traces are too imprecise when Java tests use cross-language links to C++ binaries [19].

Although several language-agnostic, yet inherently unsafe RTS approaches are reportedly used in industry [2], [4], [17], [21], research on safe RTS for C++ software is relatively sparse: while most early RTS research considered (binary) compiled languages such as C or C++ [8], [9], [18], [22]–[25], recently proposed RTS techniques focus on Java [11]–[13], [20], [26]–[28]. Since C++ itself, the size and frequency of regression testing, as well as development tool chains have significantly evolved, insights on the design and benefit of RTS in modern, large-scale industrial C++ software are largely missing [29]. To our knowledge, in the past decade, only two published studies proposed RTS techniques for C++ software [29], [30]. However, these techniques (1) are only suitable for C++ projects using the LLVM [31] compiler infrastructure, (2) do not cope with cross-language links to C++ binaries, (3) ignore changes to external files, *e.g.,* non-code artifacts, and (4) either do not

[1]IVU Traffic Technologies is one of the world's leading providers of public transport software solutions.

support dynamic linking of libraries or operating systems other than Linux [29], [30]. New approaches for RTS in modern C++ software and their industry-scale evaluation are therefore essential to address these gaps in research and practice [32].

In this paper, we present BINARYRTS, a novel RTS technique for software using C++ binaries throughout the testing process. The analyzed tests can be written in C++ or any other language with interoperability to native binaries, *e.g.,* Java tests with cross-language links using the JNI. BINARYRTS leverages dynamic binary instrumentation to collect (1) covered functions and (2) accessed external files for each test. This allows more accurate and reliable test selection, as changes to C++ binaries, non-code artifacts, or source files in other (domain specific) languages, can be properly attributed to affected tests. The instrumentation and analysis within BINARYRTS is compiler-agnostic, supports C and C++ binaries out-of-the-box, and can be transferred to different platforms as well as operating systems, and other compiled languages.

We evaluate BINARYRTS in IVU's large-scale CI infrastructure by analyzing 385 pull requests across two release branches covering more than 1,000 commits. To investigate saved testing effort with BINARYRTS, we measure the test selection ratio for the C++ test suite and the cross-language Java test suite. Our results show that BINARYRTS selects on average 26%–37% of C++ tests and 57%–64% of Java tests. BINARYRTS never fails to select tests that reveal actual regressions in the studied pull requests. Due to these promising results, IVU is currently deploying BINARYRTS to all release branches. We provide BINARYRTS as the first publicly available C++ RTS tool to foster regression testing research on C++[2].

II. Testing C++ Pull Requests at IVU

In the following, we explain the system under test and the testing process for C++ pull requests at IVU. We also elaborate on the few existing solutions for RTS in C++ software and their drawbacks in the given context.

A. System Description

The source code for the main IVU software resides in a monolithic repository. There are two major subtrees in the repository, one with mainly C++ (and some C) sources ($9.5M$ LOC) and one with mainly Java code ($4M$ LOC). Besides these two main general purpose programming languages (GPLs), IVU makes significant use of non-code artifacts, such as CSV or plain text files, as well as other GPLs and domain specific languages (DSLs), *e.g.,* TypeScript or XML.

While the Java source code is generally structured and built using Maven [33], the C++ sources are compiled through a self-maintained meta build tool (called BT hereafter). BT wraps Microsoft's C++ compiler toolchain [34], as most IVU software products target Microsoft Windows.

The C++ subtree contains code that compiles into 300+ executable binaries, including 200+ test executables and various applications. Alongside, 700+ binary DLLs are built from the subtree, which are linked against test executables and applications, both, at load-time and run-time[3].

For regression testing, unit, integration, and system tests are written in C++ or in Java. C++ tests are written using GoogleTest [35] and reside in the C++ subtree, whereas Java tests use JUnit [36] and reside in the Java subtree. However, since parts of the persistence logic are only implemented in C++, many Java tests heavily use the JNI to call C++ code for populating the database. In production, Java and C++ runtimes are usually separated. They are intertwined via JNI only to achieve two goals: (1) synchronization between the data generating parts in C++ and the data consuming parts in Java by using a single thread, and (2) allowing the test to run on an open database transaction which can easily be rolled back instead of having to commit and remove test data. For instance, a Java test might initiate a database session from within the Java virtual machine (JVM) process, hand the pointer to the session over to native code via JNI, load a few dozens DLLs at run-time and use them to generate test scenarios in the database, return control to the JVM to continue test case execution; then, update the test scenario by invoking native code again, return to the test case in the JVM, and finally clean up the database session from native code. This way, even complex test scenarios can be tested in a close-to-reality environment, which reduces the risk of late integration issues during release testing. However, these complex tests come at considerable costs: running the C++ and Java test suite takes up to 3 hours, despite high parallelization. This suggests to use RTS for more cost-effective, change-oriented testing.

In previous work, we discussed our build system aware multi-language RTS approach which we successfully deployed at IVU [19]. It uses system call analysis to trace file accesses and already takes into account that a test's outcome can be affected by changes to source code of multiple programming languages, non-code artifacts, and build system configuration files. However, the approach is limited to Java tests and imprecise in the case of C++ changes due to the analysis at file-level granularity: in case a DLL is changed, all Java tests that access the DLL during testing are selected, even if they do not execute the changed C++ code. At IVU, more than 16,000 regression tests are either part of binary executables or use DLLs at run-time. Hence, this paper focuses on precise RTS for C++ and Java tests that use C++ binaries, *i.e.,* executables or DLLs.

B. C++ Pull Request CI Pipeline

IVU usually provides support for the last ten released versions of their software products. Therefore, release branches are maintained next to the main development branch. Whenever developers want to integrate changes into any of these branches, they create a pull request. For each pull request, a CI pipeline is created that builds, analyzes, and tests the introduced changes.

If a pull request comprises changes related to the C++ subtree, the C++ code is analyzed, built, and tested through BT.

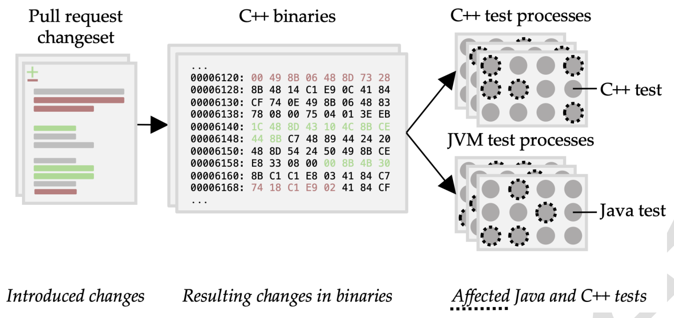
[2]BINARYRTS on GitHub: https://github.com/tum-i4/binary-rts

[3]We distinguish between *run-time* (during execution), *run time* (timespan taken by a run), and *runtime* (program execution environment) in this paper.

Since a full C++ build, even with high parallelization, takes roughly an hour, BT uses a shared remote compile cache and only compiles binaries depending on changed sources. BT also steers the C++ test execution and only runs those C++ test executables that are either directly or transitively affected by the changes. Yet, this *module-level* test selection is too coarse-granular: even very small changesets often require running thousands of tests.

Since many Java tests also make use of C++ binaries built from the C++ subtree, more precisely DLLs, these tests should also be executed for changes to the C++ subtree. In the case of changes to the C++ *and* Java subtree, the Java tests are currently selected by the established file-level RTS approach for Java tests [19]. However, for *C++-only* changesets, running selected Java tests has recently been deactivated due to the significant time overhead even for small changesets, since the selection is too imprecise (see Sec. II-A). Although developers are encouraged to check the main CI pipelines of the target branch within the next day after their pull request has been merged, this imposes the risk of missing failures and, even worse, bugs slipping into release branches.

Fig. 1 summarizes how both Java and C++ tests can be affected by changes to binaries built from the C++ code base.



```
...
00006120: 00 49 8B 06 48 8D 73 28
00006128: 8B 48 14 C1 E9 0C 41 84
00006130: CF 74 0E 49 8B 06 48 83
00006138: 78 08 00 75 04 01 3E EB
00006140: 1C 48 8D 43 10 4C 8B CE
00006148: 44 8B C7 48 89 44 24 20
00006150: 48 8D 54 24 50 49 8B CE
00006158: E8 33 08 00 00 8B 4B 30
00006160: 8B C1 C1 E8 03 41 84 C7
00006168: 74 18 C1 E9 02 41 84 CF
...
```

Fig. 1. C++ pull requests affecting Java and C++ tests

C. Existing C++ Test Selection Approaches

Over the past roughly 45 years, numerous RTS techniques have been proposed that harness static or dynamic program analysis at the level of basic blocks [9], [10], [37], functions [13], [26], [29], [30], classes or files [11]–[13], [19], [20], [27], [28], [38], or modules [14], [38], [39]. In the following, we iterate over RTS approaches for C++ software and outline why they are not applicable at IVU.

Early RTS research targeted compiled languages such as C [8], [22], [40] or C++ [9], [23]–[25], [41]. However, the described approaches were either never actually implemented [9], [41] or evaluated on small programs with a few unit tests rather than industrial-scale C++ projects [22], [24], [25]. Furthermore, most of the used analysis tools are not available today or cannot analyze modern C++ code bases, since the language and compilers have significantly evolved [29].

To our knowledge, in the past decade only two RTS approaches targeting C++ software have been presented, which have the following limitations in the given context: Fu *et al.* [29] propose RTS++, a static, function-level RTS technique. RTS++ relies on function call graphs constructed from LLVM bitcode and therefore can only analyze C++ projects targeting LLVM. Thus, RTS++ is not applicable to IVU's C++ code base, where compiling with `clang` [42] is possible, but linking can only be done using Microsoft's C++ compiler toolchain[4]. Yet, more importantly, RTS++ requires linking all libraries statically into a single binary test executable. While this may not be an issue in the comparatively small open-source projects which RTS++ has been evaluated on, it is infeasible at the scale of IVU, where many test executables and Java tests use the same hundreds of dynamically linked libraries.

To perform RTS for integration testing of distributed, large-scale C++ web services at Google, Zhong *et al.* [30] develop TESTSAGE, a dynamic, function-level RTS technique. Similar to RTS++, TESTSAGE is limited to LLVM-based projects that run on Linux or a few BSD descendants [43], whereas IVU targets the Windows operating system. TESTSAGE is further built on top of Google-internal infrastructure and code analysis tooling (*e.g.,* PIPER), which arguably limits transferability.

Besides, none of the proposed RTS techniques for C++ software has considered changes to non-code artifacts or source code of languages other than C or C++, even though they might affect test behavior [19], [20], [44]. Next to these conceptual and technical limitations, we did not find any publicly available tools that implement these RTS techniques.

In summary, we require an RTS technique that is (1) capable of analyzing tests which use arbitrary binaries at run-time, with the flexibility to support different compilers and operating systems; (2) aware of changes to non-code artifacts or source code of programming languages other than C or C++ that may affect test behavior. This motivates BINARYRTS, a novel dynamic RTS technique, which we describe in the next section.

III. BINARYRTS TECHNIQUE

This paper introduces BINARYRTS, the first RTS technique that harnesses dynamic binary instrumentation to reliably select affected regression tests that use C++ binaries or access external files, *e.g.,* source files from other languages or non-code artifacts. In the following, we first explain how BINARYRTS dynamically analyzes and instruments C++ and system binaries to generate per-test execution traces (*i.e.,* test traces), that include covered functions as well as accessed files. Second, we elaborate on the change-based test selection performed for C++ pull requests. Last, we explain how BINARYRTS has been integrated into IVU's CI test infrastructure.

A. Dynamic Binary Analysis

In order to implement any *dynamic* RTS technique, run-time information about tests is required, *i.e.,* per-test execution traces. To analyze the run-time behavior of a program, the target program needs to be *instrumented* or run in a monitored environment. Instrumentation refers to analysis code added to the program, which is executed as part of the normal program

[4]There is an ongoing effort to improve `clang`'s compatibility with MSVC projects: https://clang.llvm.org/docs/MSVCCompatibility.html

execution [45], [46]. Programs can either be instrumented statically, before the program runs, or dynamically, at program run-time. Instrumentation code can further be added through source code analysis or binary analysis. While the former is typically specific to the language and compiler, the latter is language- and compiler-agnostic but often more difficult to implement and more expensive in terms of run time overhead [45]. Yet, over the past two decades, instrumentation frameworks such as DynamoRIO [46], Intel PIN [47], or Valgrind [45] have evolved that ease the implementation of more efficient dynamic binary analysis (DBA) tools.

Nonetheless, existing dynamic RTS solutions rely on static source code analysis [30]. BinaryRTS is thus the first proposed RTS technique that leverages DBA to obtain per-test execution traces. In its current implementation, BinaryRTS relies on DynamoRIO [46], a popular and mature DBA framework [48], which supports a variety of operating systems and processor architectures, including Windows and x86-64, the primary target at IVU. DynamoRIO provides powerful application programming interfaces (APIs) to analyze and instrument basic blocks and to trace system call invocations. Therefore, DynamoRIO acts as a process virtual machine, by taking over control of the process executing the binary; it then creates and maintains a so-called *code cache* which contains a copy of the original code from the binary augmented by any added instrumentation code. Furthermore, DynamoRIO allows defining callback functions that are called whenever a new binary *module*[5] is pulled into the process [46]. This way, BinaryRTS can also analyze and instrument basic blocks from all DLLs that are dynamically loaded during execution. Since this flexibility naturally introduces run time overhead, we discuss performance considerations and implementation issues in Sec. IV-C.

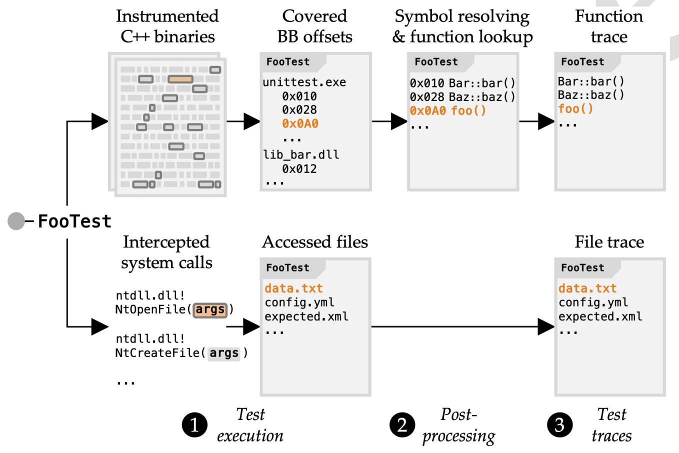


Fig. 2. Process for BinaryRTS instrumentation and test trace collection with **exemplary** covered basic block (BB) and accessed file

Fig. 2 illustrates how BinaryRTS obtains per-test execution traces using DBA: first, during *test execution* ❶, when a basic block is loaded into the code cache, BinaryRTS instruments it by (1) calculating its relative offset to the enclosing module's start address, (2) storing the triple $(module\ id, offset, hit\ count)$ in a table-like data structure, which keeps track of covered basic blocks, and (3) adding a single instruction at the beginning of the basic block to increment the hit count. Since DynamoRIO only loads basic blocks into the code cache when they are first executed or after cache invalidation, no irrelevant basic blocks are instrumented. Furthermore, BinaryRTS registers a *dump event listener* to write the basic block table to an output file (see *Covered BB offsets* in Fig. 2), either upon receiving the process exit event or a custom dump event. This can be triggered from the target program, *e.g.,* after each test case (see Sec. III-C). Alongside, BinaryRTS sets up interceptor functions that are called before system calls related to file accesses are invoked. BinaryRTS then extracts the requested file path from the provided system call arguments and stores it in a vector which is also written to a file by the dump event listener (see *Accessed files* in Fig. 2).

Second, in the *post-processing* stage ❷, all covered basic block offsets are resolved by querying the debug symbols using DynamoRIO's symbol access library. Note that BinaryRTS does not require debug builds, but merely debug symbols generated during compilation. These allow determining the source line a basic block offset corresponds to. Nevertheless, coverage will be more precise with debug builds. At IVU, we use release builds with function inlining disabled, to reliably detect all covered functions. Once the source line information has been obtained, source lines need to be mapped to C++ functions, both member or non-member functions. BinaryRTS currently uses the popular utility program `ctags` [49] to efficiently obtain C++ function declarations and definitions without the need for a preprocessor or compiler. We discuss extensions for compiler-specific function parsing in Sec. IV-C. BinaryRTS also supports resolving symbols during test execution, but shifting the work to a post-processing stage has been significantly more efficient at IVU.

Last, once each covered basic block has been resolved to its enclosing function, we generate per-test execution traces in a third step ❸. These traces contain the functions and external files a single test is associated with. Hereby, a function is stored with the attributes *file*, *signature* (*name* and *parameters*), *class* (optional), *namespace* (optional), and *meta-data* (*e.g.,* start/end line, `virtual`, `static`). A function's *identifier* is constructed by concatenating the optional scope attributes, namespace and class, and the function signature, *e.g.,* `ivu::Foo::bar(int x)`. These test traces can then be used to select tests in pull requests, as we describe next.

B. *Changed-based Test Selection*

Each pull request contains a set of changes, including additions, deletions, or modifications of files. Fig. 3 depicts that the analysis of a pull request changeset triggers the computation of affected functions and files. By combining these affected entities with the provisioned test traces, BinaryRTS can select the set of affected test cases.

[5]We stick to the DBA terminology, by collectively referring to binaries such as executables or DLLs as *modules*.

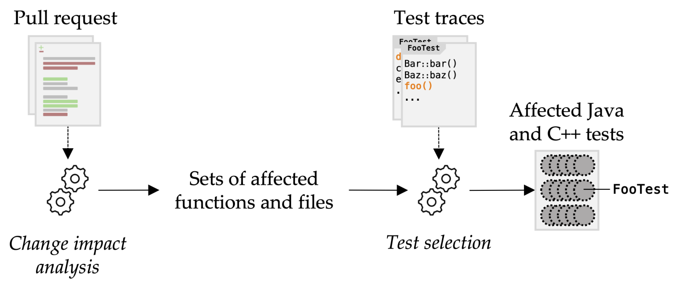


Fig. 3. BINARYRTS change impact analysis and test selection

The way BINARYRTS analyzes changesets is loosely inspired by Rothermel *et al.* [9] and Vokolos and Frankl [50], who use the Unix `diff` utility to locate differences between two program versions: we use `git diff` [51] to determine added, deleted, or modified files between the pull request branch (*new revision*) and the target release branch (*old revision*). Then, we run the change impact analysis shown in Algorithm 1 (simplified for presentation purposes) to compute affected functions and files, and finally derive the selected tests.

Algorithm 1: Change Impact Analysis and Test Selection

```
Input: Changeset, Test Traces
Output: Selected Tests
functions ← {}
files ← {}
foreach file ∈ changeset do
    if isCppFile(file) then
        if isAdded(file) then
            newFunctions ← getFunctions(file.newRev)
            /* account for build system changes */
            functions $\xleftarrow{+}$ stripFile(newFunctions)
        else if isDeleted(file) then
            functions $\xleftarrow{+}$ getFunctions(file.oldRev)
        else
            functions $\xleftarrow{+}$
              findAffectedFunctions(file.oldRev,
              file.newRev)
    else
        /* add external files, e.g., XML or SQL */
        files $\xleftarrow{+}$ file
/* query test traces for affected tests */
tests ← findAffectedTests(files, traces)
tests $\xleftarrow{+}$ findAffectedTests(functions, traces)
return tests
```

The algorithm iterates over each file in the changeset and checks if the file has a C++ file extension (*e.g.,* `.h` or `.cpp`). If not, the file is added to the set of affected files, since every test accessing this external file should be selected. If yes, the algorithm distinguishes between added, deleted, or modified files in the `git` repository. For added and deleted files, all functions in the file are added to the set of affected functions. Yet, for added files, these functions cannot exist in the test traces. Therefore, we strip the *file* attribute of the added functions to mark all functions with similar signature as affected. The rationale behind this over-approximation is that BINARYRTS aims to be agnostic about the build system. If a new source file F_{new} is added, which contains an implementation of function f, f could already be implemented in an existing source file F_{old}. Based on its configuration, the build system decides which source files to compile. Hence, if the configuration was changed to compile F_{new} instead of F_{old}, tests covering f will use the new implementation and are thus affected.

For modified files, more elaborate change impact analysis is required as depicted in Algorithm 2: first, we iterate over all functions from the new and old revision of the modified file to determine all *modified* functions. Therefore, we check whether the function body has changed by comparing the code inside the body for textual equivalence, excluding comments and whitespaces. Due to static and dynamic dispatch in C++, a newly *added* function can affect the run-time program behavior, even in the absence of other changes, such as a modification of an existing function:

- **Function Overloading:** If a new function is added that has the same name as an existing function, but different parameters (*e.g.,* `foo(int)` and `foo(short)`), the compiler determines and uses the most suitable function at each call site. BINARYRTS therefore marks functions with the same name as the added one as affected. We limit ourselves to functions in the *same file*, since marking *all* functions with the same name can lead to high imprecision.
- **Virtual Function Overriding:** If a new member function is added to class B that overrides a virtual member function in B's parent class A, due to dynamic dispatch, all uses of the parent's member function need to be marked as affected. BINARYRTS thus marks all member functions with similar signature of any class (or struct) as affected.
- **(Scope) Function Overriding:** C++ allows defining functions in global, class, namespace, or local scopes and if multiple functions with similar signature exist in different scopes, it is up to the compiler to decide at each call site which function to call. Thus, if a new non-global function is added, BINARYRTS will by default mark all functions with similar signature as affected.

We also mark all *deleted* functions as affected to select all tests that previously executed the deleted function.

By handling the scenarios for static and dynamic dispatch as described, we deliberately design BINARYRTS to prefer safety over precision. We still remain flexible by not requiring more elaborate (and costly) compiler-specific static analysis which might be more accurate [9]. However, BINARYRTS provides run-time options to skip these over-approximations to trade increased RTS precision for reduced safety.

Algorithm 2: Finding Affected Functions

```
Input: Old (oldRev) and New (newRev) File Revision
Output: Affected Functions
Function findAffectedFunctions(oldRev, newRev):
    affected ← {}
    /* find modified or newly added functions */
    foreach $f_{new} \in$ getFunctions(newRev) do
        isAddedFunction ← true
        foreach $f_{old} \in$ getFunctions(oldRev) do
            if $f_{old}$.identifier = $f_{new}$.identifier then
                /* functions with changed body */
                if hasBodyChanged($f_{old}$, $f_{new}$) then
                    affected $\overset{+}{\leftarrow}$ $f_{new}$
                isAddedFunction ← false
                break
        if isAddedFunction then
            /* new overloading function */
            if hasParameters($f_{new}$) then
                affected $\overset{+}{\leftarrow}$ stripParameters($f_{new}$)
            /* new virtual overriding function */
            if isVirtualOverride($f_{new}$) then
                affected $\overset{+}{\leftarrow}$ replaceClass($f_{new}$, $*$)
            /* new scope overriding function */
            else if ¬ hasGlobalScope($f_{new}$) then
                affected $\overset{+}{\leftarrow}$ stripScope($f_{new}$)
    /* find deleted functions */
    foreach $f_{old} \in$ getFunctions(oldRev) do
        isDeletedFunction ← true
        foreach $f_{new} \in$ getFunctions(newRev) do
            if $f_{old}$.identifier = $f_{new}$.identifier then
                isDeletedFunction ← false
                break
        if isDeletedFunction then
            affected $\overset{+}{\leftarrow}$ $f_{old}$
    return affected
```

In addition, BINARYRTS has a run-time option to handle changes to *non-functional* code entities (*e.g.,* macros, global/member variables) [22]: we use `ctags` to locate all non-functional entities inside a C or C++ file, determine if they have changed, and then, similar to White *et al.* [25], perform a text-based lookup for (calling) functions that use the changed entity. These functions are then added to the set of affected functions. We further discuss this run-time option in Sec. IV-C1.

We list all run-time options in Sec. IV-A3 and evaluate them regarding their impact on safety and precision in our empirical study at IVU (see Sec. IV).

Finally, the affected tests are computed by querying the test traces with the affected functions and files. BINARYRTS uses efficient hash tables to minimize the time for finding tests that use affected functions or files. We provide measurements for the run time of the change impact analysis and test selection in Sec. IV-C3.

C. Integration into Pull Request CI at IVU

We integrated BINARYRTS into IVU's infrastructure as follows: to obtain test traces, we created new test tracing pipelines for each release branch considered in our evaluation (see Sec. IV). These tracing pipelines run all C++ and Java tests during off-peak hours (at night or on the weekend) with BINARYRTS's instrumentation enabled. For C++ tests, we add a GoogleTest test listener to BINARYRTS, which triggers a dump event after test setup and for every test case. BINARYRTS supports all GoogleTest test case types, including value- or type-parameterized tests [29], accounts for changes to (global) test setup code, and always selects newly added test cases. For Java tests, we use a Java agent [52] to attach BINARYRTS to the JVM process before the JUnit test suite starts. Similar to prior research [19], [20], [53], [54], we run each JUnit test suite in a forked JVM process for better test isolation and reliability. As the covered basic blocks will be dumped upon receiving the JVM process exit event, we do not need to trigger custom dump events for Java tests.

Once the test traces have been created for a release branch, they are serialized and stored on a network drive and can then be used inside pull requests for this release branch.

IV. EVALUATION

To evaluate BINARYRTS in a real-world industrial context, we perform a large-scale study in IVU's CI infrastructure. Our goal is to empirically determine the cost-effectiveness of BINARYRTS in terms of saved testing effort and how many real test failures BINARYRTS fails to select. In addition to the commonly used retest-all baseline, we compare BINARYRTS to IVU's internal module-level C++ test selection (BT) and our DLL-level Java test selection from prior work [19]. We further aim to understand precision and safety trade-offs for different run-time options of BINARYRTS (see Sec. III-B). Overall, we seek to answer the following research questions (RQs):

- **RQ**$_1$: How much testing effort can BINARYRTS save for C++ tests compared to retest-all and module-level RTS?
- **RQ**$_2$: How much testing effort can BINARYRTS save for cross-language Java tests compared to DLL-level RTS?
- **RQ**$_3$: How safe is BINARYRTS for changesets of pull requests in terms of real missed test failures?

A. Experimental Setup

1) Evaluation Branches: To conduct our experiments, we first pick two release branches, one rather old release branch receiving mainly maintenance changes (R_M) and one recent release branch with ongoing feature development (R_D). As shown in prior work, there can be significant differences in RTS effectiveness depending on the type of the release branch [19]. For both branches, we set up separate test tracing CI pipelines which run in off-peak hours, as described in Sec. III-C. Then, we modify the pull request pipelines for R_M and R_D: inside each pull request run[6] we invoke BINARYRTS to compute selected C++ and Java tests using the most recent test traces for the respective target release branch.

2) Evaluation Metrics: For each pull request run, we measure the reduction in testing effort by comparing (1) the number of selected tests and (2) their cumulative duration against the set of tests selected by a baseline regression testing strategy. Related research usually compares RTS techniques against a retest-all testing strategy [19], [29], [30], [55], which we adopt for RQ_1. However, since the state-of-practice at IVU is better reflected by BT's module-level RTS strategy for C++ tests (RQ_1) and our DLL-level RTS strategy for Java tests (RQ_2), we add these as more realistic baseline strategies.

For RQ_1, we also report how often BINARYRTS excludes entire test executables. IVU's integration tests require a costly (global) database setup, which is performed when the process is started. Thus, skipping an entire test executable can significantly reduce overall test time.

Regarding RQ_2, recall that for *C++-only* pull requests, executing Java tests has currently been deactivated in the pull request CI pipelines, due to high execution times even with DLL-level RTS (see Sec. II-B). Yet, to perform our evaluation, we require the actual test verdicts and run time of Java tests. Therefore, we run the missing Java tests for *C++-only* pull requests in off-peak hours (at night and on the weekend) as selected by our DLL-level RTS strategy.

An RTS technique is considered safe, if it selects all tests that potentially expose a fault [18]. While safety for existing RTS techniques has been (semi-)formally proven under the assumption of code changes [9], [20], [27]–[29], prior research has shown that outdated test traces [19], [30] or changes related to non-code artifacts or cross-language links [14], [20], [44], [56] can compromise RTS safety. Hence, to perform a fair evaluation of RTS safety, we need to inspect all test failures that were not selected by BINARYRTS to understand if they actually reflect real regressions introduced in the respective pull requests. We discuss practical challenges in distinguishing between real regressions and flaky failures in Sec. IV-B.

3) BINARYRTS Configuration: Prior research has not yet investigated how testing effort and safety are affected by different levels of change impact analysis depth (see our discussion on static and dynamic dispatch in Sec. III-B). We are particularly interested in these trade-offs, as they provide practical guidelines on how to calibrate BINARYRTS during operation. Therefore, we run and compare BINARYRTS with the following run-time option configurations:

- B_{slim}: Disables all run-time options
- $B_{overload}$: B_{slim} + function overloading analysis
- $B_{override}$: B_{slim} + function overriding analysis
- $B_{virtual}$: B_{slim} + virtual function overriding analysis
- $B_{non\text{-}functional}$: B_{slim} + non-functional entity analysis
- B_{full}: BINARYRTS default; enables all run-time options

B. Results

We collect the dataset for evaluating BINARYRTS across four weeks of development, covering a total of 385 pull requests with 587 CI pipeline runs and more than 1,000 commits. Pull requests for R_D are more common (288) and have a larger median changeset size of 9 files than pull requests on R_M (97) with a median of 2 changed files.

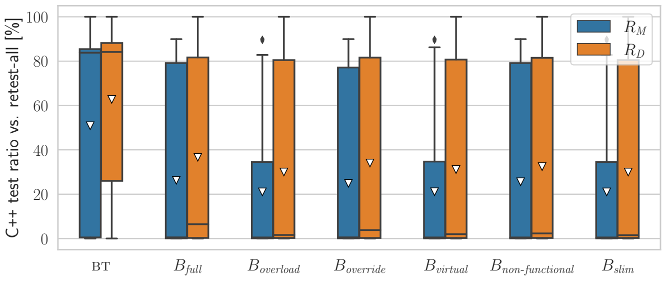


Fig. 4. Distribution of selected C++ test ratio of BINARYRTS configurations and BT compared to retest-all across all pull request runs

RQ_1: C++ Testing Effort: Fig. 4 shows the distribution of the ratio of selected C++ tests compared to retest-all across all pull request runs on the two branches R_M and R_D for different BINARYRTS configurations as well as BT. Note that while pull request CI pipelines only execute C++ tests selected by the static module-level RTS strategy of BT, BT still outputs a retest-all test report after execution. It can do so since it caches test results from unaffected test cases and can thereby report test verdicts from all executed plus cached tests. We compute the distribution plots for BT and all BINARYRTS configurations from these retest-all reports and the selected tests from BT and BINARYRTS, respectively. The results indicate that BT selects on average 51% (R_M) and 63% (R_D) of tests, whereas BINARYRTS selects on average 26% and 37% of tests with B_{full} configuration and 21% and 30% with B_{slim}. Moreover, the median selection ratios are 84% and 84% (BT), 0.5% and 6.5% (B_{full}), and 0.5% and 1.5% (B_{slim}).

In addition to the ratio of selected C++ tests, we compare the relative test duration for the different strategies against retest-all: the average relative test durations are 58% and 70% (BT), 32% and 44% (B_{full}), and 26% and 36% (B_{slim}).

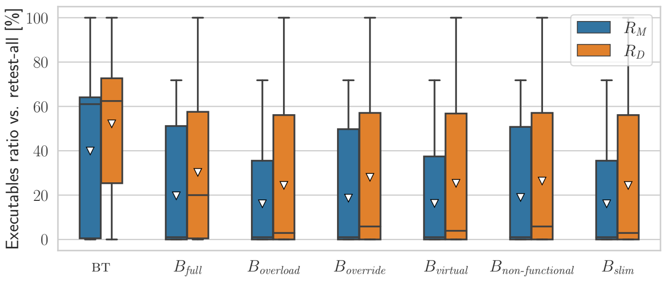


Fig. 5. Distribution of selected test executables ratio of BINARYRTS configurations and BT compared to retest-all across all pull request runs

[6] Recall that a new CI pipeline *run* is triggered whenever the pull request is updated, *i.e.,* if one or more commits are added.

The test execution costs are not only driven by the test duration, but also by the time required for process initialization and global (database) test setup for each test executable. Hence, if C++ test executables are skipped because no tests inside have been selected, end-to-end execution time for testing decreases. Therefore, we report the distribution of the ratio of selected test executables across all pull requests in Fig. 5.

Overall, we find that BINARYRTS saves considerably more testing effort than BT for both branches. Similar to our previous observations [19], the achieved savings for the maintenance branch (R_M) are higher than for the development branch (R_D). The difference between BINARYRTS configurations is small regarding the average test selection ratio ($\leq$7 pp), but becomes more visible when looking at the median test selection ratio where B_{full} selects more than four times as many tests as B_{slim} for R_D. We discuss the impact on safety in RQ$_3$. BINARYRTS further selects on average only 20% (R_M) and 30% (R_D) of C++ test executables with B_{full}, whereas BT selects 40% and 52% test executables, respectively.

SUMMARY RQ$_1$. *We find that* BINARYRTS *selects on average 26% (R_M) and 37% (R_D) of all C++ tests with B_{full} on two release branches, whereas* BT *selects 51% and 63%, respectively. In 50% of the pull request runs on R_D, B_{full} selected more than four times as many test cases as B_{slim}.* BINARYRTS *further reduces the number of C++ test executables by 80% (R_M) and 70% (R_D).*

RQ$_2$: Java Testing Effort: In our prior study [19], we were able to save on average 42% of execution time for cross-language Java tests compared to retest-all using a file-level RTS strategy. However, in case of C++ changes, we encountered problems of imprecise test selection, since a test that uses a DLL is selected whenever changes are made to that DLL, regardless of whether the test covers the changed code or not.

To investigate the effectiveness of BINARYRTS for cross-language Java tests at IVU, we compare BINARYRTS against our file-level—or rather, DLL-level—RTS strategy. Similar to this pre-existing strategy, BINARYRTS selects Java tests at the level of JUnit test suites [19].

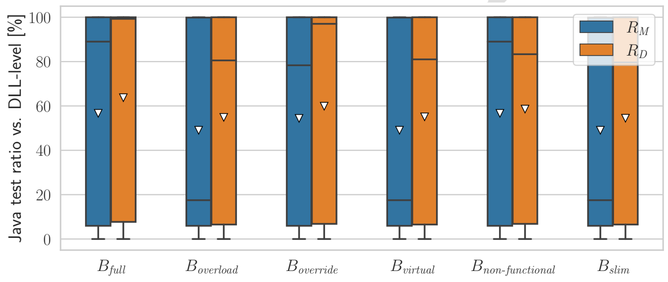


Fig. 6. Distribution of selected Java test ratio of BINARYRTS configurations compared to DLL-level RTS across all pull request runs

Fig. 6 depicts the test selection ratios for the different configurations of BINARYRTS. The results show that BINARYRTS with B_{full} selects on average 57% (R_M) and 64% (R_D) of the Java tests selected by the pre-existing DLL-level RTS strategy. The median selection ratios are higher at 89% and 99% (B_{full}), and 18% and 80% (B_{slim}).

In summary, we find that while BINARYRTS can be much more effective than DLL-level RTS for Java tests, DLL-level RTS selects almost the same number of tests in roughly 50% of pull request runs as B_{full} for R_D. We expect the reason to be that Java tests often use largely similar parts of the C++ code for their test setup and, therefore, are selected altogether for changes to core C++ components.

SUMMARY RQ$_2$. *We find that* BINARYRTS *(B_{full}) selects on average 57% (R_M) and 64% (R_D) of the Java tests selected by DLL-level RTS on two release branches. DLL-level RTS still performs comparatively well in roughly 50% of pull request runs for R_D.*

RQ$_3$: Safety: To assess the safety of BINARYRTS for different configurations, we compare the failed tests in a pull request run to the tests that BINARYRTS would have selected. If a test has failed in a pull request run and has not been selected by BINARYRTS, we need to check if this missed failure represents a real regression introduced in the pull request. Therefore, we check in the following order if the test (1) already failed on the target release branch, (2) failed due to infrastructure issues, (3) is a known flaky test, *i.e.,* a test with non-deterministic result, or (4) a newly detected flaky test which both, fails and passes within 100 reruns. We manually validate all missed failures, since especially test failures of categories (2) and (3) are sometimes difficult to automatically recognize from stack traces and often require discussion with IVU developers. If steps (1)–(3) do not provide a clear indication, we rely on rerunning tests (4) to detect flakiness, as is common practice in industry [57]–[59].

In total, we manually checked 179 pull request runs with missed C++ and Java test failures. The majority of missed failures either have already existed on the target branch or can be attributed to infrastructure issues, mainly database access problems. Most flaky tests we encounter are already known, but we still find a handful of new flaky tests that we reported to the responsible developers. Interestingly, we almost exclusively find flaky tests on R_D, indicating that flaky tests seem to get fixed eventually before a software version release.

Overall, we do not find any missed failures that are related to the changes introduced in the corresponding pull request for B_{full}. For B_{slim}, we also do not find any real missed failures. Despite these results, there are several sources for potential unsafe RTS behavior which we discuss in Sec. IV-C1.

SUMMARY RQ$_3$. *We find that* BINARYRTS *detects all real regressions in the considered pull requests with all configurations.*

C. Discussion

Next, we discuss weaknesses and possible improvements regarding safety, precision, and efficiency of BINARYRTS, and share feedback from IVU developers.

1) Safety: We expect BINARYRTS to be safe for changes to C++ functions if all run-time options are enabled (B_{full}), as BINARYRTS was designed following the example of safe function-level RTS techniques for C++ [29], [30]. In contrast with BT, which is only safe for changes to C++ files, BINARYRTS is further aware of changes to external files, thus ruling out a common source of unsafe RTS behavior [20], [56].

Yet, similar to existing techniques [19], [30], BINARYRTS can be unsafe if test traces are outdated. At IVU, we run the tracing CI pipelines to update test traces for release branches in off-peak hours, but at least once per week. Another source for potential safety violations is that BINARYRTS ($B_{overload}$) only marks overloaded functions within the *same* file as affected by static dispatch (see Sec. III-B). Moreover, since BINARYRTS operates at run-time, expressions evaluated at compile-time, *e.g.,* macros or `constexpr`, are only considered if analysis for non-functional changes is enabled ($B_{non\text{-}functional}$). However, the text-based search to find usages of non-functional entities can become expensive for large C++ code bases if all parent directories of the changed source file are recursively searched. Therefore, BINARYRTS provides a parameter to control how many levels of parent directories to visit during the search. Safety violations can possibly occur if this parameter is set too low and thus functions making use of the changed non-functional entity are not marked as affected. Based on IVU conventions for the use of non-functional entities, we set the parameter to 2. Additionally, BINARYRTS allows defining a regular expression to match files that should trigger a retest-all strategy to anticipate context-specific safety challenges.

Due to these limitations and because prior research has revealed safety violations of supposedly safe RTS solutions [56], we conduct the safety trade-off experiments for RQ$_3$.

2) Precision: Since prior RTS research has studied more coarse-grained and less precise RTS at the level of files for Java and C# projects [11], [19], [20], [27], [38], we also investigate how our results from RQ$_1$ change if we aggregate our function-level test traces to file-level test traces. The results show that BINARYRTS with file-level analysis selected on average 41% (R_M) and 49% (R_D) of C++ tests. Hence, function-level gives better precision than file-level analysis.

We further see potential for improvement in how BINARYRTS deals with static and dynamic dispatch (see Sec. III-B). Currently, we resort to over-approximation approaches as we rely on the compiler-agnostic, yet simple analysis tool `ctags` to locate and parse functions. However, with more elaborate static analysis from C++ compilers, we could reduce the set of affected functions due to static and dynamic dispatch.

3) Efficiency: BINARYRTS is specifically designed to have low overhead inside CI pipelines of pull requests to compute the set of selected tests, commonly called the *analysis phase* of RTS systems [13], [29]. The time for selecting tests with B_{full}, the configuration with maximum overhead, was on average roughly 30 seconds, where reading and deserializing test traces from disk took most of the time. This could be further improved by using a database to minimize involved I/O.

For the so-called *collection phase*, *i.e.,* when collecting per-test execution traces in dedicated CI pipelines, we observe relatively high instrumentation overhead of roughly a factor of 2–3. While allowing compiler-agnostic code and system call instrumentation, the overhead introduced by dynamic binary instrumentation is generally expected to be higher than for static source code instrumentation (often several times slower than the original program) [45], [60], [61]. Other instrumentation tools for C or C++, such as `OpenCppCoverage` or `CodeCoverage.exe` by Microsoft, exhibit similarly high overhead [62]. Moreover, DynamoRIO has been shown to have significant performance impact in other contexts as well [47], [63]. It also lacks support for efficiently instrumenting dynamically generated code [64]. Therefore, we need to disable the JVM's just-in-time compiler when tracing Java tests which further increases instrumentation overhead. However, since the CI pipelines that collect the per-test execution traces are executed *offline* [13], meaning in off-peak hours independently of any pull requests, the instrumentation overhead does not impact the development process. To further increase the tracing frequency, we envision improvements related to using more lightweight binary instrumentation solutions (BINARYRTS has experimental support for the DBA tool `Frida` [65]) or implementing source code instrumentation through source-to-source transformation using `clang`. If the latter is properly implemented, compiling and linking the transformed source code would still be possible with any compiler and linker.

4) Developer Feedback: We have continuously discussed the design of BINARYRTS and its evaluation with IVU engineers to establish broad support among developers and testers. As they see great value in the proposed RTS solution, we are currently integrating BINARYRTS into all release branches. Moreover, developers have suggested to implement further developer-aiding tools for test coverage visualization and test gap analysis based on the test traces collected with BINARYRTS.

D. Threats to Validity

1) External Validity: BINARYRTS has been designed to suit the context-specific challenges at IVU and, therefore, our findings do not necessarily generalize to other software projects inside and outside of IVU. Moreover, even though BINARYRTS also supports Linux and multiple platforms, our evaluation at IVU was performed on C++ software built with Microsoft's compiler toolchain to x86-64 binaries on Windows. Nevertheless, our results confirm prior RTS research on C++ software that reported significant savings in testing effort [29], [30]. We publish the source code of BINARYRTS to ease transferability to other projects beyond the context of IVU.

Another threat to validity emerges from the fact that, similar to previous studies [14], [19], [55], we use test durations from test reports to measure test execution time. GoogleTest measures test durations only in millisecond resolution, which may distort results as some fast-running unit tests sometimes take less than one millisecond to execute. To address this threat, we also report the ratio of selected tests and test executables.

The latter provides an indication regarding savings in (global) test setup costs, which can be substantial at IVU.

2) Internal Validity: Internal threats stem from the implementation of BinaryRTS, mainly related to using DynamoRIO for analyzing and instrumenting C++ binaries, and `ctags` for parsing C++ source files. To address these threats, we manually validated selection results with IVU engineers and wrote automated unit and integration tests for BinaryRTS.

V. Related Work

We have referenced several RTS techniques throughout this paper that have motivated and partly inspired BinaryRTS (see Sec. II-C). Below, we list studies targeting C or C++ software that we consider to be most relevant for this work.

Early related RTS research was primarily on C software. In 1994, Chen *et al.* [22] presented TestTube, an RTS technique for C programs. Similar to BinaryRTS, it tracks covered functions and non-functional entities per test case. TestTube employs source code instrumentation and static analysis, and selects a test if any of its covered entities has changed. Rothermel and Harrold [8], [40] proposed DejaVu, an RTS technique for C which uses static control flow graphs and edge-level test traces obtained through source code instrumentation to compute affected tests. Using this fine-grained analysis, DejaVu is safe for C code modifications [66], [67].

Later, Rothermel *et al.* [9] extended DejaVu to object-oriented C++ software. Therefore, they combine interprocedural and class control flow graphs with edge-level test traces. Their proposed RTS technique also accounts for dynamic dispatch and polymorphism, but, due to the lack of adequate C++ analysis and instrumentation tools at the time, it was not actually implemented. In 1995, Kung *et al.* [23] presented an RTS technique for C++ based on static class dependency graphs. Using these graphs they compute a *class firewall* (see also Leung *et al.* [68]), that is the set of classes affected by changes, and derive which tests need to be selected to retest affected classes. Jang *et al.* [24] and White *et al.* [25] extended the class firewall approach for C++ software to improve precision and safety by adding fine-grained change impact analysis and data flow analysis, respectively. The class firewall approach has also inspired the development of DejaVOO [10] and Starts [12], [28], two class-level static RTS techniques for Java.

In the past decade, only two studies on RTS in C++ software were published, which we deem as most related to this work: Fu *et al.* [29] presented RTS++, a static RTS technique operating on function call graphs. RTS++ targets modern C++ programs that compile to LLVM bitcode and use the GoogleTest testing framework. Fu *et al.* evaluated RTS++ on 11 open-source projects and find that the number of selected tests is on average reduced by 61% compared to retest-all. RTS++ is not applicable at IVU, as it requires all libraries to be statically linked into a single executable binary.

To address integration and system testing in large-scale C++ web services at Google, Zhong *et al.* [30] developed TestSage, a dynamic RTS technique for distributed systems. When deploying TestSage to Google testing infrastructure, they achieved up to 50% reduction in testing time. TestSage also targets LLVM-based projects, as it relies on a customized version of XRay, a function instrumentation tool for LLVM. TestSage further uses Google's internal version control system and code analysis tool Piper to perform change impact analysis for test selection. In contrast, BinaryRTS is based on publicly available tools and frameworks, and, due to the employed binary instrumentation, works with different compilers, operating systems, and binary formats.

In two previous studies, we have studied unsafe and safe RTS techniques to implement RTS for the multi-language code base at IVU [19], [55]. Unsafe RTS is typically language-agnostic, as it relies only on readily available CI and version control system (VCS) metadata [55]. We found that the best performing unsafe RTS technique saved on average 19.8% of testing time while 93.4% of failures were still detected on the main development branch. Since unsafe RTS is not suitable for pull requests to *release* branches, we developed a safer RTS approach for Java [19]. This new approach is akin to Ekstazi [11], [27] and RTSLinux [20], two file-level RTS techniques for Java, and tracks opened files for each test through system call analysis. Similar to BinaryRTS, this makes it safe for changes to external files, such as configuration files, as well as source files in other programming languages, such as SQL or XML. Furthermore, the approach is build system aware, meaning it accounts for changes to the build system configuration, and selectively builds only those Java modules required for testing; this resulted in a an end-to-end CI pipeline time reduction for Java by 50%–63% on average. However, we also found that the file-level analysis granularity was too coarse grained when Java tests accessed DLLs, as it results in most tests being selected upon any C++ changes. BinaryRTS is built on top of these insights and provides a practical RTS solution for tests that use C++ binaries either directly (C++ tests) or through cross-language links (Java tests).

To summarize, no prior RTS research analyzes regression tests which use arbitrary C++ binaries, accounts for multi-language source files and non-code artifacts, or performs dynamic binary instrumentation for RTS. We are the first to evaluate C++ RTS for pull requests in industry-scale CI.

VI. Conclusion

In this paper, we present BinaryRTS, a dynamic RTS technique for reliably selecting tests that use C++ binaries during execution. It harnesses dynamic binary instrumentation to monitor covered functions and accessed files for each test at run-time. This way, BinaryRTS is also aware of cross-language links to source files in other programming languages and non-code artifacts used during testing. We evaluate BinaryRTS in IVU's large-scale CI infrastructure on roughly 16,000 C++ and Java tests, some of which cover code from hundreds of C++ binaries. Our results indicate that BinaryRTS excludes on average 63%–74% of C++ tests and 36%–43% of Java tests, thereby reducing test duration by on average up to 68% against a naive retest-all baseline. The improved testing time directly translates to faster feedback in CI testing, boosting developer efficiency and satisfaction, which is why IVU is currently deploying BINARYRTS to all release branches. To foster RTS research on languages other than Java, we publish BINARYRTS and its source code as the first publicly available RTS tool for C++ software.

ACKNOWLEDGMENTS

We thank Dennis Bracklow, René Dammer, Stefan Golas, Maximilian Pohl, and Stefan Sieber for their support. This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant Q-Soft 01IS22001B. The responsibility for this article lies with the authors.

REFERENCES

[1] H. K. Leung and L. White, "Insights into regression testing," in *Proceedings of the International Conference on Software Maintenance*, 1989, pp. 60–69.

[2] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2014, pp. 235–245.

[3] A. A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagppan, "Fastlane: Test minimization for rapidly deployed large-scale online services," in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 408–418.

[4] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, 2019, pp. 91–100.

[5] K. Fischer, F. Raji, and A. Chruscicki, "A methodology for retesting modified software," in *Proceedings of the National Telecommunications Conference*, 1981, pp. 1–6.

[6] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing Verification and Reliability*, vol. 22, pp. 67–120, 2012.

[7] K. F. Fischer, "A test case selection method for the validation of software maintenance modifications," in *Proceedings of International Computer Software and Applications Conference*, 1977, pp. 421–426.

[8] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 173–210, 1997.

[9] G. Rothermel, M. J. Harrold, and J. Dedhia, "Regression test selection for c++ software," *Software Testing, Verification and Reliability*, vol. 10, pp. 77–109, 2000.

[10] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2004, pp. 241–251.

[11] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *Proceedings of the International Conference on Software Engineering*, 2015, pp. 713–716.

[12] O. Legunsen, A. Shi, and D. Marinov, "Starts: Static regression test selection," in *Proceedings of the International Conference on Automated Software Engineering*, 2017, pp. 949–954.

[13] L. Zhang, "Hybrid regression test selection," in *Proceedings of the International Conference on Software Engineering*, 2018, pp. 199–209.

[14] A. Shi, P. Zhao, and D. Marinov, "Understanding and improving regression test selection in continuous integration," in *Proceedings of the International Symposium on Software Reliability Engineering*, 2019, pp. 228–238.

[15] E. Knauss, M. Staron, W. Meding, O. Soder, A. Nilsson, and M. Castell, "Supporting continuous integration by code-churn based test selection," in *Proceedings of the International Workshop on Rapid Continuous Software Engineering*, 2015, pp. 19–25.

[16] B. Busjaeger and T. Xie, "Learning for test prioritization: An industrial case study," in *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2016, pp. 975–980.

[17] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2017, pp. 12–22.

[18] G. Rothermel and M. J. Harrold, "A framework for evaluating regression test selection techniques," in *Proceedings of the International Conference on Software Engineering*, 1994, pp. 201–210.

[19] D. Elsner, R. Wuersching, M. Schnappinger, A. Pretschner, M. Graber, R. Dammer, and S. Reimer, "Build system aware multi-language regression test selection in continuous integration," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 87–96.

[20] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, "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*, 2017, pp. 809–820.

[21] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," *Empirical Software Engineering*, vol. 27, pp. 1–34, 2022.

[22] Y. F. Chen, D. S. Rosenblum, and K. phong Vo, "Test tube: a system for selective regression testing," in *Proceedings of the International Conference on Software Engineering*, 1994, pp. 211–220.

[23] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, "Class firewall, test order, and regression testing of object-oriented programs," *Journal of Object-Oriented Programming*, vol. 8, pp. 51–65, 1995.

[24] Y. K. Jang, M. Munro, and Y. R. Kwon, "An improved method of selecting regression tests for c++ programs," *Journal of Software Maintenance and Evolution*, vol. 13, pp. 331–350, 2001.

[25] L. White, K. Jaber, B. Robinson, and V. Rajlich, "Extended firewall for regression testing: An experience report," *Journal of Software Maintenance and Evolution*, vol. 20, pp. 419–433, 2008.

[26] L. Zhang, M. Kim, and S. Khurshid, "Faulttracer: A spectrum-based approach to localizing failure-inducing program edits," *Journal of Software: Evolution and Process*, vol. 25, pp. 1357–1383, 2013.

[27] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2015, pp. 211–222.

[28] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2016, pp. 583–594.

[29] B. Fu, S. Misailovic, and M. Gligoric, "Resurgence of regression test selection for c++," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2019, pp. 323–334.

[30] H. Zhong, L. Zhang, and S. Khurshid, "Testsage: Regression test selection for large-scale web service testing," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2019, pp. 430–440.

[31] LLVM, "Llvm compiler infrastructure." [Online]. Available: https://llvm.org/

[32] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, 2018, pp. 1–23.

[33] A. Maven, "Maven." [Online]. Available: https://maven.apache.org

[34] Microsoft, "Msvc c++ toolset." [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/projects-and-build-systems-cpp

[35] Google, "Googletest." [Online]. Available: https://google.github.io/googletest/

[36] JUnit, "Junit 5," 2021. [Online]. Available: https://junit.org/junit5

[37] M. J. Harrold, A. Orso, J. A. Jones, T. Li, M. Pennings, S. Sinha, A. Gujarathi, D. Liang, and S. A. Spoon, "Regression test selection for java software," *ACM SIGPLAN Notices*, vol. 36, pp. 312–326, 2001.

[38] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric, "File-level vs. module-level regression test selection for .net," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2017, pp. 848–853.

[39] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, and J. Czerwonka, "Optimizing test placement for module-level regression testing," in *Proceedings of the International Conference on Software Engineering*, 2017, pp. 689–699.

[40] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 1993, pp. 358–367.

[41] ——, "Selecting regression tests for object-oriented software," in *Proceedings of the International Conference on Software Maintenance*, 1994, pp. 14–25.

[42] Clang, “Clang compiler.” [Online]. Available: https://clang.llvm.org/
[43] LLVM, “Llvm xray function call tracing.” [Online]. Available: https://llvm.org/docs/XRay.htm
[44] D. Elsner, R. Wuersching, M. Schnappinger, and A. Pretschner, “Probe-based syscall tracing for efficient and practical file-level test traces,” in *Proceedings of the International Conference on Automation of Software Test*, 2022, pp. 126–137.
[45] N. Nethercote, “Dynamic binary analysis and instrumentation or building tools is easy,” Ph.D. dissertation, University of Cambridge, 11 2004.
[46] D. Bruening, “Efficient, transparent, and comprehensive runtime code manipulation,” Ph.D. dissertation, MIT, 9 2004.
[47] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.
[48] DynamoRIO, “Dynamorio.” [Online]. Available: https://dynamorio.org
[49] ctags, “Universal ctags.” [Online]. Available: https://ctags.io
[50] F. I. Vokolos and P. G. Frankl, *Pythia: A regression test selection tool based on textual differencing*. Springer, 1997.
[51] git, “git.” [Online]. Available: https://git-scm.com
[52] “Java agent api,” 2017. [Online]. Available: https://docs.oracle.com/javase/9/docs/api/java/lang/instrument/package-summary.html
[53] J. Bell and G. Kaiser, “Unit test virtualization with vmvm,” in *Proceedings of the International Conference on Software Engineering*, 2014, pp. 550–561.
[54] P. Nie, A. Celik, M. Coley, A. Milicevic, J. Bell, and M. Gligoric, “Debugging the performance of maven’s test isolation: Experience report,” in *Proceedings of the International Symposium on Software Testing and Analysis*, 2020, pp. 249–259.
[55] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, “Empirically evaluating readily available information for regression test optimization in continuous integration,” in *Proceedings of the International Symposium on Software Testing and Analysis*, 2021, pp. 491–504.
[56] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, “A framework for checking regression test selection tools,” in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 430–441.
[57] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, “Deflaker: Automatically detecting flaky tests,” *Proceedings of the International Conference on Software Engineering*, pp. 433–444, 2018.
[58] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, “Understanding flaky tests: The developers perspective,” in *Proceedings of the ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, Inc, 8 2019, pp. 830–840.
[59] W. Lam, K. Muslu, H. Sajnani, and S. Thummalapenta, “A study on the lifecycle of flaky tests,” in *Proceedings of the International Conference of Software Engineering*, 2020, pp. 1471–1482. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-study-on-the-lifecycle-of-flaky-tests/
[60] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The art, science, and engineering of fuzzing: A survey,” *IEEE Transactions on Software Engineering*, 2019.
[61] A. Engelke and M. Schulz, “Instrew: Leveraging llvm for high performance dynamic binary instrumentation,” in *Proceedings of the International Conference on Virtual Execution Environments*, 2020, pp. 172–184.
[62] CQSE, “Performance impact of c++ profilers.” [Online]. Available: https://docs.teamscale.com/howto/setting-up-profiler-tga/cpp/#performance-impact
[63] M. A. B. Khadra, D. Stoffel, and W. Kunz, “Efficient binary-level coverage analysis,” in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1153–1164.
[64] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao, “Optimizing binary translation of dynamically generated code,” in *Proceedings of the International Symposium on Code Generation and Optimization*, 2015, pp. 68–78.
[65] Frida, “Frida.” [Online]. Available: https://frida.re/
[66] G. Rothermel and M. J. Harrold, “Analyzing regression test selection techniques,” *IEEE Transactions on Software Engineering*, vol. 22, pp. 529–551, 1996.
[67] G. Rothermel, “Efficient, effective regression testing using safe test selection techniques,” Ph.D. dissertation, Clemson University, 5 1996.
[68] H. K. Leung and L. White, “A study of integration testing and software regression at the integration level,” in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Press, 1990, pp. 290–301.