2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)

TestSage: Regression Test Selection for Large-scale Web Service Testing

Hua Zhong
Google Inc.
The University of Texas at Austin
Mountain View, California USA
hzhong@utexas.edu

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

Sarfraz Khurshid
The University of Texas at Austin
Austin, Texas USA
khurshid@utexas.edu

***Abstract*—Regression testing is an important but expensive activity in software development. Among various types of tests, web service tests are usually one of the most expensive (due to network communications) but widely adopted types of tests in commercial software development. Regression test selection (RTS) aims to reduce the number of tests which need to be retested by only running tests that are affected by code changes. Although a large number of RTS techniques have been proposed in the past few decades, these techniques have not been adopted on large-scale web service testing. This is because most existing RTS techniques either require direct code dependency between tests and code under test or cannot be applied on large scale systems with enough efficiency. In this paper, we present a novel RTS technique, TestSage, that performs RTS for web service tests on large scale commercial software. With a small overhead, TestSage is able to collect fine grained (function level) dependency between test and service under test that do not directly depend on each other. TestSage has also been successfully applied to large complex systems with over a million functions. We conducted experiments of TestSage on a large scale backend service at Google. Experimental results show that TestSage reduces 34% of testing time when running all AEC (Analysis, Execution and Collection) phases, 50% of testing time while running without collection phase. TestSage has been integrated with internal testing framework at Google and runs day-to-day at the company.**

***Index Terms*—Regression Test Selection, Web Service Testing, Regression Testing, System Testing**

I. Introduction

Regression testing [1] plays an important role at software development life-cycle by running a predefined group of tests against updated revisions of a project to ensure that the new changes do not break existing functionalities of the software. Although crucial, it is very costly to run the full set of regression tests in practice due to the large number of tests and software revisions. For example, Google reported that the company runs over 100 Million tests each day, consuming a large number of resources [2], [3].

Web service testing is usually adopted by developers to verify a major component of complex distributed systems. However, web service testing could be extremely expensive if the underlying system is enormous and complex. This is because: (1) building and bringing up various components on servers is expensive, (2) a test case tends to take more time and resources to run due to network latency, and (3) different revisions of components can also create more combinations to test. In addition, if web service testing is adopted at pre-submit stage, which means the tests are executed before submitting a change, it could result in an even larger number of revisions. At Google, many development teams try to avoid running large web service tests at pre-submit stage to reduce testing load, with the compromise of finding certain bugs at post-submit stage.

Regression Test Selection (RTS) [1], [3]–[11] is a promising technique to optimize regression testing. RTS runs a subset of tests by skipping those that are not affected by code changes, as the skipped tests should produce the same results with prior runs. A traditional RTS technique collects dependencies for each test at previous revision and then runs tests if their dependencies are modified in the new revision. An RTS technique is *safe* if it guarantees all tests whose behaviors are impacted in the newer revision are selected for execution [12]. An RTS technique is *precise* if unaffected tests are not selected.

Many RTS techniques have been proposed in the past few decades. These techniques differ in the mechanism they use to collect test dependency (static [13], [14] vs dynamic [4], [6], [9], [15]–[17]), and the granularity on which they collect dependencies (basic block level [4], [12], [15], method level [9], [16], file level [13], [17] and etc.). Recent studies [17], [18] on real-world programs demonstrate that RTS collecting dynamic dependencies at the file or mixed file-method granularity are cost-effective as they make a good tradeoff between selection precision and runtime overhead, and thus representing state-of-the-art RTS.

Despite recent progress made in RTS techniques, none of them can be effectively applied on large-scale web service testing. This is because: (1) most traditional dynamic and static RTS techniques require direct code dependency between test and System Under Test (SUT) to collect test dependency and use this dependency to select affected tests. Web service tests, on the other hand, run in a separate environment from SUT and such dependency cannot be directly obtained; (2) enormous dependency data produced by a large test case can create non-trivial overhead for an RTS technique, thus slowing down or even crashing the tool; and (3) web service tests and SUT may even have distributed binaries implemented under different languages.

In this paper, we present TestSage, the first dynamic RTS approach which collects function granularity test dependency for large-scale web service tests. TestSage works in its own environment, separating from tests and SUT, and remotely instruments SUT and runs tests to collect dependency. TestSage works for web service tests that (1) interact with SUT through Remote Procedure Calls (RPC); (2) are written in different languages from SUT; and (3) are built on separate environments, platforms, and machines, or even in different GEO locations from SUT. We implemented TestSage as a

978-1-7281-1736-2/19/$31.00 ©2019 IEEE
DOI 10.1109/ICST.2019.00052

IEEE computer society

web service to interact with tests and SUT through different RPCs. In our current implementation, TESTSAGE requires the SUT to be written in C/C++ as the prototype instrumentation tool adopted in TESTSAGE is implemented in C++; however, the basic idea of TESTSAGE applies to other languages.

To evaluate the proposed technique, we applied TESTSAGE to collect dependency for Google Assistant [19], a major Google backend system that consists of dozens of self-contained components. TESTSAGE was enabled for test-class-level RTS in both pre-submit and post-submit tests of the SUT since December 2017 and has been serving continuously at Google. It collects dependency for around 200 changed test classes (the exact number varies from day-to-day, consisting of about 10,000 test methods) that are implemented to validate the SUT. We measured executions of TESTSAGE in a randomly picked 32 day window. In 11028 pre-submit executions and 360 post-submit executions, TESTSAGE skipped 55.26% of pre-submit tests and 41.29% of post-submit tests on average. TESTSAGE also saved 49.96% and 33.77% execution time for the two types of tests, respectively. Note that to speed up the pre-submit executions, we turned down collection phase in pre-submit stage and used dependency data collected at post-submit stage. This drastically increased the performance of pre-submit testing but made the selection potentially unsafe. Interestingly, we have not discovered any bugs leaked into post-submit stage due to unsafe selection at the time when we draft this paper.

This paper makes the following contributions:

1) Design and implementation of the first function-level dynamic RTS technique, TESTSAGE, that is easy to integrate into large-scale web service tests.
2) Experimental results (11388 executions) of TESTSAGE on the large-scale Google Assistant backend service, demonstrating the effectiveness, efficiency and scalability of dynamic RTS on large and complex real-world software systems.

II. BACKGROUND AND EXAMPLE

This section introduces the real-world web service used in our experiment, walks through a sample test case, illustrates the limitations of existing RTS techniques, and presents a novel RTS technique that is applicable on large scale web service test, TESTSAGE.

A. System Under Test and Sample Test

System Under Test. The system under test is the backend service of Google Assistant [19], the virtual assistant powered by artificial intelligence and users can interact with the system through natural voice. The Assistant is able to search the Internet, answer questions, schedule events and alarms, adjust hardware settings on the user device, and show information from the user Google account. The backend system accepts a RPC request containing user voice input and returns a voice response to the user. The system contains multiple individual sub services and connects to other backends services. The system has been widely used in practice, e.g., it was installed on more than 400 million devices in 2017 [20]. Although the detailed size of the system cannot be shown due to the Google policy, it is among one of the largest systems developed in Google.

Sample Test Case. A set of web service tests send RPC calls with pre-defined voice queries to the system and validate the responses. The test is categorized into different test classes based on the type of queries. Consider a sample test class which verifies knowledge based questions answering of the system. One test method of the class sends out a question "How far is the moon?" to the server and verifies the received response. TESTSAGE computes a dynamic function-level dependency with more than 45,000 functions for this single test. The size of the dependency data is 9.4 Megabytes.

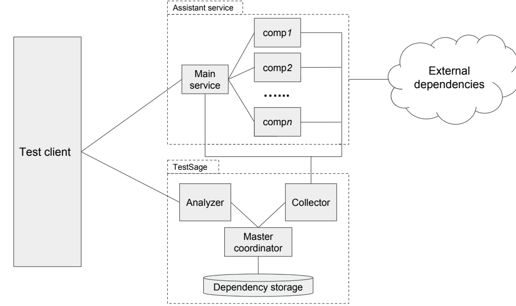

Fig. 1: Interactions between Google Assistant service, TestSage and the test client.

B. Existing RTS Techniques and TestSage

Existing RTS techniques. Given no direct dependency between test and SUT, static RTS techniques [13], [14] are unable to compute dependency for tests. Most of existing dynamic RTS techniques also require tests to run on the same platform with SUT. Although dynamic RTS techniques have been proposed before [10], [21]–[24] for web services, they require extensive analysis (both dynamic and static) of the service application and tests to model the application. These techniques are based on the assumption that the updated models of web service should be sufficient to highlight the changes that can cause regression testing. Some of the proposed techniques are unsafe [21], others incur non-trivial overhead to model the underlying web services. However, in a complex system like the one used in our experiment, generating fine-grained model of the system is non-trivial and is impractical to adopt. On the other hand, coarse-grained models on the system are insensitive to small changes and cannot be used to perform RTS for those changes. We discuss the details of these techniques in Section VII.

TestSage. TESTSAGE is able to obtain function-level dependencies for tests without the need to modify existing system. TESTSAGE does this by instrumenting the source code of the web service with a function call tracing system called XRAY [25], [26]. XRAY combines compiler-inserted instrumentation points and a run-time library that can dynamically enable and disable the instrumentation on a running service. XRAY is part of the LLVM compiler infrastructure [26], [27] and works with all code written in C/C++ and Objective-C/C++. At runtime, TESTSAGE interacts with XRAY through RPC calls to enable/disable dependency collection for individual tests. It also uses RPC to download the dependency into its own dependency storage. TESTSAGE works on a modified version of XRAY which instruments and profiles web service with minimum overhead and updates test dependency at nearly full speed. Fig. 1 shows the interactions of TESTSAGE with test client and assistant service. Note that TESTSAGE is connected with each service component of the

```
void foo() {
  printf("Hi, XRay!");
}
```

C++ function

```
.Lfunc_begin0:
  .file 22 "test.cc"
  .loc 22 6 0
  .cfi_startproc
  .p2align 1, 0x90
.Lxray_sled_0:
  .ascii "\353\t"
  nopw 512(%rax,%rax)
.Ltmp0:
  pushq %rbp
.Ltmp1:
  .cfi_def_cfa_offset 16
...
...
.Ltmp4:
  .loc 22 6 48 prologue_end
  movb $0, $al
  callq printf
  .loc 22 6 74 is_stmt 0
  movl %eax, -4(%rbp)
  addq $16, %rsp
  popq %rbp
  .p2align 1, 0x90
.Lxray_sled_1:
  retq
  nopw %cs:512(%rax,%rax)
.Ltmp5:
.Lfunc_end0:
...
```

Assembler with *no-op sleds*.

Fig. 2: XRAY instrumentation

system so it can collect dependency of a test case on all services within the system.

C. XRay

Since XRAY plays a key role in TESTSAGE, before digging into the overall architecture of TESTSAGE, we first introduce XRAY instrumentation in this section.

XRAY **instrumentation.** As mentioned above, XRAY [25], [26] is a function call tracing system that instruments C/C++ based binaries to debug performance issues. To instrument the code, XRAY inserts a few bytes of code that do nothing in function entry/exits at compile time. These inserts are referred as *no-op sleds* in the rest of this paper. The locations of these no-op sleds are encoded and stored in the object files. XRAY tracing can be disabled/enabled at runtime. If disabled, the no-op sleds are executed as-is and introduce minimal execution overhead. On the other hand, once enabled, the XRAY runtime library overwrites the no-op sleds with calls to instrumentation code that records function entry/exit information and stores the information to in-memory buffers. Additional information like cycle-counter time stamps are also stored so that XRAY can reconstruct the program operation at post-processing time.

XRAY works for fully multithreaded programs. When XRAY tracing is enabled at runtime, it inserts instrumentation at the right sections of the program code. Disabling tracing undoes the changes. Program semantics can be kept intact when XRAY tracing is being turned on. Therefore, it is not required to force single processor execution or stop execution of the program while XRAY instrumentation is being added and enabled. To further reduce the overhead of the no-op sleds inserted at function entry and exit points, XRAY employs heuristics to determine which functions to instrument. XRAY only instruments a function if it has at least N "machine" instructions or it has a loop after performing all optimizations. N is a configurable parameter that can be set at compile time.

Instrumentation Point Insertion at Compile-time. XRAY instrumentation can be enabled by passing flag *-fxray-instrument* to Clang. Upon receiving this flag, Clang emits LLVM Intermediate Representation (IR) code which signals the LLVM backend to XRAY instrument functions based on the heuristic. At link-time, the flag will add additional dependencies to the XRAY runtime. An example of such instrumentation point insertion is shown in Fig.2.

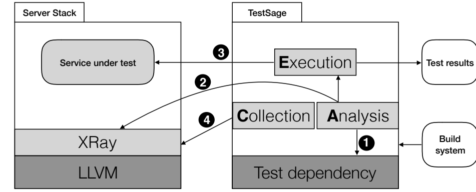

Fig. 3: Design overview of TESTSAGE

In the LLVM back-end compiler, the LLVM IR code is translated into machine code with the XRAY instrumentation points and the XRAY instrumentation map. The instrumentation map is generated at the lowest level of the stack when writing out the instrumentation points for functions that are XRAY instrumented. The map is loaded as an array in memory with pointers to the code locations where XRAY instrumentation points are located. These maps are written out per object file, and are concatenated by the linker.

Function Tracing at Runtime. The XRAY runtime provides platform-specific functions for patching and un-patching the instrumentation points embedded in functions at runtime. XRAY runtime enables instrumentation before main starts and it records all function entry and exit events which may result in a lot of records in the resulting trace. To allow customized profiling, XRAY runtime also provides interfaces through which new implementations of the event handling functionality can be created. TESTSAGE leverages on these interfaces and implements a *"Coverage Mode"* bundled with the default runtime library. This new mode self-registers to the XRAY runtime, and is installed through a flag.

III. TECHNIQUES AND IMPLEMENTATION

This section illustrates the architecture of TESTSAGE, and breaks it down into different components. Like most RTS techniques, TESTSAGE has three main phases: (1) an analysis phase (**A**) which selects what tests to run in a given revision; (2) an execution phase (**E**) that runs the selected tests; and (3) a collection phase (**C**) collects information for the updated revision.

The overall design of TESTSAGE is shown in Fig. 3. We integrate TESTSAGE with *Blaze*, the internal build system at Google [28] so TESTSAGE can be invoked like regular tests. In the analysis phase, upon receiving the new program revision from build system, TESTSAGE first computes the updated functions in the new revision and retrieves the list of affected tests from dependency (❶). Next, before sending the affected tests to test runner, TESTSAGE first calls XRAY to start a trace for each test on the server (❷). If multiple services are included in the SUT, TESTSAGE initiates calls to all services and handle them in batch process. TESTSAGE then initiates the execution phase to run the selected tests and monitors the status of the execution (❸). Once a test finishes running, TESTSAGE reports the result, then calls XRAY to stop the trace of the test on server(s). Finally, collection phase starts to download the updated dependency of the test

Algorithm 1: Coverage Handler Algorithm

```
1  Function CoverageHandler(func, coverage_mode):
2      if coverage_mode then
           // turn off tracing for later calls
3          unpatch_function(func)
4          log_function_call_record(func, EntryType::Enter)
5      else
6          if EntryType::Enter then
7              log_function_call_record(func, EntryType::Enter)
8          else if EntryType::Exit then
9              log_function_call_record(func, EntryType::Exit)
10         else
11             unknown(func)
12         end
13     end
14     return
```

(❹). If collection phase is turned off, TESTSAGE serves only as a filter for the build system to skip unaffected tests.

In rest of this section, we provide more details on each phase and their integration with the selected testing framework. Since TESTSAGE is integrated with internal systems built at Google, it relies on a number of infrastructures at Google to function.

A. TestSage Customization of XRay

TESTSAGE relies on XRAY to obtain test coverage on underlying SUT. However, the default XRAY runtime library produces more information than required, at a cost of undesired latency. For example, the default library stores both function entry and exit plus additional timestamps to debug performance issues. It also stores information that can be used to generate function call graph in post-processing. These information significantly increase the overhead of XRAY, and they are not useful to TESTSAGE to understand test coverage of the program.

1) Coverage Event Handler: To resolve the above issue, TESTSAGE implements a custom event handler and register it in *Coverage Mode*. Upon entering or exiting a function, the default XRAY handler logs a corresponding entry or exit record. Multiple entry/exit records are produced if a function is visited multiple times during a run. In contrast, the customized coverage handler logs only one entry record for a function, regardless of the access type of the visit. Additionally, the coverage handler immediately un-patches the instrumentation points embedded in the function to disable the instrumentation of the function. All following visits to the function will be run at full speed and no additional records are produced. As a result, the coverage handler still emits enough trace data for TESTSAGE to compute test coverage while producing a smaller overhead that is almost un-noticeable. The detailed algorithm of coverage handler is illustrated in Algorithm 1.

2) Coverage Tracing: The coverage mode can be run in two ways: (1) collect coverage of program from start to end; (2) collect coverage over a certain interval in the middle of execution; TESTSAGE adopts the second option to gather test coverage data for different test cases. By executing different tests at different intervals on a running system, TESTSAGE is able to obtain clean, isolated coverage data for each individual test. The granularity of a test entity can also be easily configured by adjusting the group of tests (i. e. a test class) to run inside a single interval. TESTSAGE parses the raw XRAY trace data of an interval and produces lists of functions executed (*"live"*) and not executed (*"dead"*). The control mechanism of the interval is explained in Section III-E.

B. Change computation

TESTSAGE relies on the Google version control system, *Piper* [29], to compute the change information from two software revisions. *Piper* offers change information at different granularity (file, function, statement, etc.) and we are not going to cover the details of the system in this paper. However, TESTSAGE cannot directly use the change information provided by *Piper* to compute affected tests. Before we can dig into the root cause, we first need to define a few terms. We follow the work by Chen et al. [30] to divide a program into program elements called *functional* and *non-functional* code entities. *Functional entity* includes directly executable unit of code like a function or a statement. *Non-functional entity* includes non-executable unit of code like a global variable, an instance variable or a macro. The change information returned by *Piper* contains both *functional* and *non-functional* code entities. We know that XRAY only instruments functions and the coverage data does not include *non-functional* code entities. To bridge this coverage gap, we adopt an algorithm similar to the one proposed in [30] to statically compute the transitive closure of references to all *non-functional entities* that are reachable from *functional entities*. To illustrate the method, lets say we have a test case $t \in T$, where T is the test suite. At revision v, a list of *functional entities* FE are computed by XRAY for t. For every entity $fe \in FE$, we perform static analysis to obtain the set of *non-functional entities* referenced by the fe (denoted as NE_{fe}). After iterating through FE, we obtain all *non-functional entities* for t (i.e., $\bigcup_{fe \in FE} NE_{fe}$), which together with FE form the complete dependency data for t. As a matter of fact, *Piper* pre-computes such information for some revisions and we can obtain this information for free at those revisions. Once TESTSAGE obtains change information from *Piper*, it moves on to analysis phase to compute the affected tests.

Note that for web service testing, the change information of tests needs to be handled separately as it resides in a different environment. TESTSAGE also computes change information for tests, but at a coarser granularity. It simply checks if a test is added, updated or deleted in the new revision (including all the dependencies of the test).

C. Analysis Phase

As mentioned in Section II-C, TESTSAGE computes the function dependencies of each test through XRAY tracing. Once the dependent *functional* and *non-functional* code entities are computed, TESTSAGE stores all entities as keys in a lookup table. The name of the test is stored as value to a entity key if it executed the entity in previous revision. In analysis phase, TESTSAGE first calls *Piper* to obtain the updated code entities in the latest revision. The affected tests

can be retrieved by simply querying the lookup table with the entities. However, there is one interesting side note. With the help of Google infrastructure, we can also derive file-level dependencies from the lookup table by intersecting the code entities of a file with entities covered by a test. Google builds function indices of its code base and TESTSAGE can retrieve code entities in a source file through internal source code index service. We consider a test depends on a file if the intersection between the two is non-empty. We will explore this direction in our future work.

To build a safe RTS, TESTSAGE also needs to analyze the change information of a test. A modified test could have different end-to-end coverage on a new revision, even when there is no change in its dependency at the revision. Besides utilizing test dependency to select affected tests like traditional RTS, TESTSAGE also selects a test if it is added or modified in the new revision. TESTSAGE skips the test if it is removed in the new revision. Since TESTSAGE is designed for web service tests, in which case the SUT and test framework are in distributed binaries, TESTSAGE operates a RPC service to serve the analysis logic.

D. Execution Phase

TESTSAGE is integrated with an internal integration test framework at Google which dispatches gRPC [31] requests to servers to validate their responses. Before executing any tests, the test framework first calls the analysis RPC in TESTSAGE to determine what tests not to run for the new revision. This reduces unnecessary overhead as loading and building integration test cases is costly. TESTSAGE requires change information of both the services and the test to analyze which tests can be skipped in the new revision.

Unlike traditional RTS techniques, TESTSAGE does not need to have its own execution component to run tests. However, to obtain clean XRAY coverage for each test, TESTSAGE needs to coordinate test executions on servers. We illustrate this control mechanism in details in Section III-E.

Typical RTS techniques can choose to use *one-pass* execution or *two-pass* execution to run selected tests and update test dependency. In *one-pass* system, the selected tests are executed once, producing test outcomes and updating dependency information of selected tests at same time. This is usually preferred as it simplifies both execution phase and collection phase. However, collecting dependency has extra overhead. Even though XRAY coverage mode introduces minimum latency when updating test dependency, the interval mechanism of XRAY limits the number of tests that can be run in parallel. To enable maximum test execution parallelization, TESTSAGE uses *two-pass* in pre-submit testing. The test framework kicks off two runs of tests at same time. One runs on a server that is not XRAY instrumented with maximum parallelization to produce test outcome as fast as possible, and one runs on a XRAY instrumented server sequentially to update test dependencies. In actual production setup the test framework also brings up dozens of duplicated XRAY instrumented services for the second run so dependency update can also run tests in parallel.

E. Collection Phase

The collection phase creates XRAY trace files for the executed tests. Depending on the number of duplicated

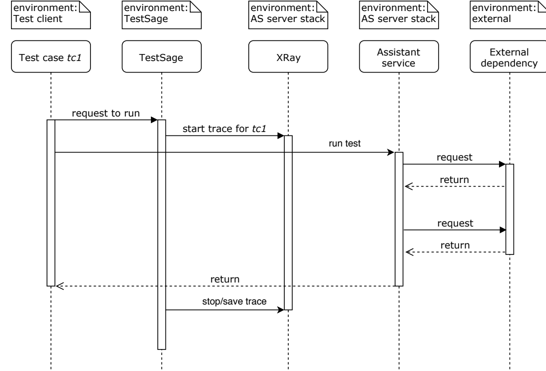

Fig. 4: Test execution with TESTSAGE

services running in parallel, TESTSAGE divides the number of selected tests into same number of partitions, and runs each partition sequentially on a service. Since the system under test can be loosely coupled and composes of multiple distributed services running on different environments, TESTSAGE appends a RPC service on every single service to control XRAY tracing intervals (Fig. 1). Before running a test, TESTSAGE first calls all services to start tracing intervals. It then runs and waits for the test on SUT. After the test finishes running, TESTSAGE dispatches calls to all services to stop the traces on them. A final batch of RPC calls are made to those services to download the raw trace files to cloud storage. Fig. 4 shows the execution sequence of a sample test case.

After all trace files are downloaded from the services, TESTSAGE parses the files and extracts test dependencies to update the lookup table. To illustrate test dependency extraction for a test case, TESTSAGE first parses the raw XRay trace files from all services and extracts fully qualified function names from them, TESTSAGE then iterates through the functions to compute non-functional code entities for every function. All entities are then stored into a lookup table using Bigtable [32]. Bigtable is a distributed storage system developed at Google. It manages structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers.

IV. EVALUATION

We implement TESTSAGE with the purpose of saving resources and testing time by safely skipping unaffected tests that run on large web service systems. This task cannot be handled by traditional RTS techniques. To learn the effectiveness of TESTSAGE, we applied the tool on two integration test suites that validate Google Assistant backend service. We access the performance of TESTSAGE by answering the following research questions:

1) **RQ1:** How many tests do TESTSAGE skip on average? Given a large web service test case usually executes an end-to-end scenario and can cover a large portion of the system, we want to learn if RTS techniques are still meaningful in the area of integration testing.
2) **RQ2:** How much time can TESTSAGE save on average? One key benefit of RTS techniques is reducing testing time by skipping unaffected test cases. Large web

TABLE I: Number/ratio of test classes selected

Date	$TestP_{Sel}$[#]	$TestP_{All}$[#]	Diff [#]	$TestP_{Sel}$/ $TestP_{All}$[%]	$TestC_{Sel}$[#]	$TestC_{All}$[#]	Diff [#]	$TestC_{Sel}$/ $TestC_{All}$[%]
2018-09-02	1381	3168	1787	43.59%	847	2259	1412	37.49%
2018-09-03	8589	18392	9803	46.70%	885	2268	1383	39.02%
2018-09-04	26607	71096	44489	37.42%	1758	2268	510	77.51%
2018-09-05	21613	52118	30505	41.47%	1669	2268	599	73.59%
2018-09-06	30319	58542	28223	51.79%	1345	2268	923	59.30%
2018-09-07	37071	70557	33486	52.54%	1343	2250	907	59.69%
2018-09-08	2643	6100	3457	43.33%	839	2349	1510	35.72%
2018-09-09	3950	8015	4065	49.28%	855	2349	1494	36.40%
2018-09-10	26432	62936	36504	42.00%	1680	2403	723	69.91%
2018-09-11	28505	59904	31399	47.58%	1737	2403	666	72.28%
2018-09-12	25882	57921	32039	44.69%	1545	2403	858	64.29%
2018-09-13	24006	49016	25010	48.98%	1461	2403	942	60.80%
2018-09-14	26260	53253	26993	49.31%	1596	2403	807	66.42%
2018-09-15	1819	3672	1853	49.54%	982	2394	1412	41.02%
2018-09-16	2124	4968	2844	42.75%	924	2394	1470	38.60%
2018-09-17	19054	49468	30414	38.52%	1590	2394	804	66.42%
2018-09-18	27587	64587	37000	42.71%	1599	2394	795	66.79%
2018-09-19	27220	58319	31099	46.67%	1737	2367	630	73.38%
2018-09-20	21396	49054	27658	43.62%	1578	2376	798	66.41%
2018-09-21	19614	52738	33124	37.19%	1631	2367	736	68.91%
2018-09-22	639	1908	1269	33.49%	751	2376	1625	31.61%
2018-09-23	2355	5406	3051	43.56%	798	2376	1578	33.59%
2018-09-24	21963	53867	31904	40.77%	1679	2385	706	70.40%
2018-09-25	22159	48785	26626	45.42%	1743	2385	642	73.08%
2018-09-26	24225	51376	27151	47.15%	1562	2403	841	65.00%
2018-09-27	21551	48151	26600	44.76%	1596	2403	807	66.42%
2018-09-28	21682	43298	21616	50.08%	1778	2403	625	73.99%
2018-09-29	1020	2280	1260	44.74%	912	2394	1482	38.10%
2018-09-30	1726	3990	2264	43.26%	883	2394	1511	36.88%
2018-10-01	19455	42243	22788	46.05%	1781	2394	613	74.39%
2018-10-02	26292	52509	26217	50.07%	1657	2394	737	69.21%
2018-10-03	13029	30523	17494	42.69%	1805	2394	589	75.40%
Max	37071	71096	44489	52.54%	1805	2403	1625	77.51%
Min	639	1908	1260	33.49%	751	2250	510	31.61%
Avg	17443	38693	21250	44.74%	1392	2365	973	58.81%

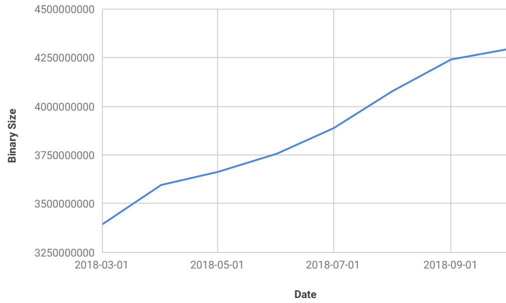

Fig. 5: Binary-size growth of the studied system in six months

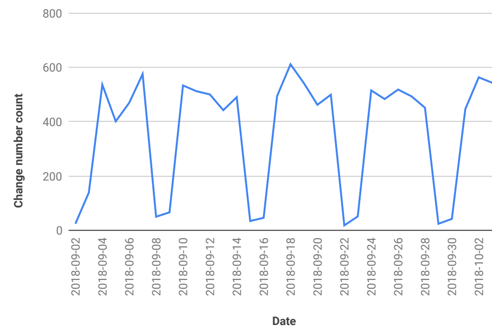

Fig. 6: Daily submitted changes to the system

service test is usually one of the slowest types of tests to run.

3) **RQ3:** What is the overhead of TESTSAGE in updating dependencies and what is the trade-off between the above two metrics and overhead introduced by TESTSAGE? Since TESTSAGE is a dynamic code-instrumentation-based RTS technique, the overhead of TESTSAGE could potentially triumph the benefit we gain from it.

In the remaining part of this section, we describe the projects and tests used in the experimental evaluation, the experimental setup, and report the results of experiment to answer the above questions.

A. Projects Description

We used the Google Assistant backend system as our main project. This system performs natural language understanding, conversational AI support and is tightly integrated with Google Search systems. It is the back-bone system of Google Assistant, the virtual assistant powered by artificial intelligence. It is among one of the biggest systems developed at Google, and is being actively updated. Fig. 5 shows the size and growth of the system within a 7 month window. Note that the detailed unit of the size is hidden intentionally due to the policy of Google. From Fig. 6 we learn that on average developers submit ~400 changes in a day to the system. Each submission could invoke regression tests multiple times due to different reasons (failed submission attempt, local client sync, etc.). Fig. 7 shows the size of a test suite and its growth in a month. Note that each test class contains a varies number of test methods, ranging from 5 to 1000.

We performed our evaluation on two suites of tests that validate the system: (1) a pre-submit test suite that runs

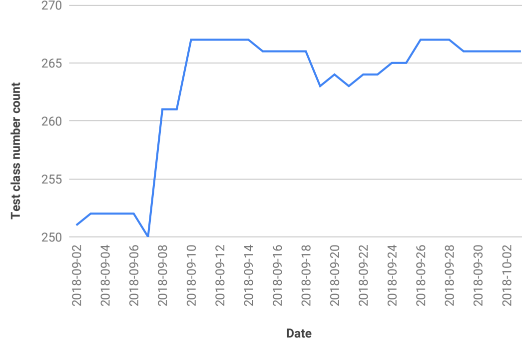

Fig. 7: Growth of test class number in the TestC suite

before every change gets submitted to the code base (**TestP**); and (2) a continuous test suite which runs every 2 hours to validate the latest head version of the code base (**TestC**). TESTSAGE was enabled back in December 2017 at Google, and has been serving the above two test suites ever since. In this paper, we randomly selected 11028 revisions of TestP and 360 revisions of TestC (executions in 32 days) to analyze the performance of the technique.

B. Experimental Setup

Before we begin with the experimental setup, it is worth mentioning that TESTSAGE executes with different setup in TestP and TestC. TestP runs much more frequent than TestC as it is triggered before every submit. Fig. 6 shows there are around 400 submits every day. Besides, developers often sync to different revisions before submitting the change and it is almost impossible to compute test dependency for that many revisions within a short time. Therefore, to reduce the pre-submit test time, the analysis phase of TestP uses the latest test dependency computed in TestC to select tests. We understand this approach is not safe, thus we only integrate this configuration with pre-submit tests. Additionally, for large web service test, it is very unlikely to have two changes submitted within the 2 hour time window with the former change introducing unsafe selection for the latter one. As a matter of fact, we did not notice any occurrences of such issue during our experiment phase.

The goal is to evaluate how the two types of test suites behave with TESTSAGE enabled and analyze the overhead of TESTSAGE dependency update. For every revision in TestP, we run TESTSAGE with two scenarios: (1) TestP$_{All}$ executes all tests in TestP, and (2) TestP$_{Sel}$ executes tests in TestP with TESTSAGE but without dependency update. For revisions in TestC, we also run TESTSAGE in two scenarios: (1) TestC$_{All}$ executes all tests in TestC, and (2) TestC$_{Sel}$ executes tests in TestC with TESTSAGE and also updates dependency data.

The metrics we measure are the number of executed tests for every setup and its testing time. TestP and TestC adopt different execution parallelization method and trigger different number of server stacks (TestP brings up 2 stacks as it is executed more often; TestC brings up 15 stacks instead). We report two types of testing time, elapsed time (testing time) and executor days. Elapsed time is the actual time taken from the start of the execution to the end. Note that if different execution parallelization setups are adopted in two executions, elapsed time could be an unfair comparison as one execution runs more tests in parallel. In contrast, executor days sum up the total time of each test executor.

C. Results Analysis

We ran all experiments on Borg [33] – Google's cluster management system. For a fair comparison, we requested same amount of resources for each test stack.

1) RQ1: How many tests do TESTSAGE *skip on average:* Table I shows the number of test classes selected by TESTSAGE, the total number of test classes in TestP$_{All}$ and TestC$_{All}$, the number of test classes skipped and the percentage of the selected tests in TestP and TestC. One concern we had before conducting the experiment was the performance of RTS techniques on large web service test. Given that a large web service test case usually covers an end-to-end scenario, it might not be very effective to perform RTS on it. However, TESTSAGE skipped a large portion of tests on average. We believe this is due to the nature of the Google Assistant system. A "what is the weather in Mountain View" test and a "navigate to Googleplex" test could trigger different code paths in the system, thus making the two tests have relatively distinct function dependencies. We also noticed that TESTSAGE is less effective for changes that modify the core logic of the system, which most tests depend on. Almost all test classes are selected by TESTSAGE in those changes. In general, we can see that the average percentage of skipped pre-submit tests across 32 days is 55.26%. Note that this number would be much higher if we filter out non-binary submits. Since TESTSAGE dependency can only be computed on source code, we make TESTSAGE blindly select all tests if non-binary changes are detected in a submit for sake of safety. If we count only binary submits, the average percentage of skipped tests is around 90%.

2) RQ2: How much time can TESTSAGE *save on average:* In Table II we count the daily average execution time of both studied test suites (TestP, TestC) in four test configurations (TestP$_{Sel}$, TestP$_{All}$, TestC$_{Sel}$, TestC$_{All}$) across 32 days. Recall that TestP is a pre-submit test suite and TestC is a continuous test suite. $_{Sel}$ means the suite is executed with RTS and $_{All}$ means the suite runs all its tests blindly. Note that even though we report the execution time of TestP and TestC in one table, they are not comparable with each other as they use different configurations (execution parallelization, assigned resources, etc). In the initial setup, we also noticed that the execution time of same test varies drastically in different runs. This is due to the internal testing infrastructure at Google automatically re-run failed test cases up to a threshold. To eliminate this noise, we ignore all executions that logged test retries. It is not surprising that TESTSAGE performs well in TestP as TESTSAGE skips dependency collection phase in its execution. What is more interesting is TESTSAGE outperforms existing solution in TestC where all phases are included and it still reduced ~34% testing time.

Besides testing time analysis, we also measure the average resource consumption of TestP calculated by Borg. Table III shows the average resource consumption of pre-submit testing suites (i.e., TestP$_{Sel}$ and TestP$_{All}$). Note that we did not present the resource consumption information for continuous testing suites (i.e., TestC$_{Sel}$ and TestC$_{All}$), because they run much less frequently than pre-submit testing and their

TABLE II: Average Execution time reduction.

Date	TestP$_{Sel}$[s]	TestP$_{All}$[s]	TestP$_{Sel}$/ TestP$_{All}$[%]	TestC$_{Sel}$[s]	TestC$_{All}$[s]	TestC$_{Sel}$/ TestC$_{All}$[%]
2018-09-02	1817	3813	47.65%	2610	3660	71.31%
2018-09-03	2102	3945	53.28%	2673	3696	72.32%
2018-09-04	1775	4008	44.29%	2710	3712	73.01%
2018-09-05	1996	4270	46.74%	2781	3779	73.59%
2018-09-06	1956	4146	47.18%	2716	3861	70.34%
2018-09-07	2238	4061	55.11%	2774	3693	75.12%
2018-09-08	1714	3920	43.72%	2849	3968	71.80%
2018-09-09	2026	3911	51.80%	2810	3939	71.34%
2018-09-10	2116	4006	52.82%	3079	4874	63.17%
2018-09-11	1826	4122	44.30%	3174	4629	68.57%
2018-09-12	1799	3937	45.69%	3057	4740	64.49%
2018-09-13	2207	4038	54.66%	3087	4941	62.48%
2018-09-14	2182	4128	52.86%	3169	4705	67.35%
2018-09-15	1936	3840	50.42%	3018	4565	66.11%
2018-09-16	1964	3983	49.31%	2948	4541	64.92%
2018-09-17	2389	3945	60.56%	3072	4834	63.55%
2018-09-18	2305	4460	51.68%	3228	4910	65.74%
2018-09-19	2080	4272	48.69%	3006	4834	62.18%
2018-09-20	2220	4263	52.08%	2936	4636	63.33%
2018-09-21	1985	4151	47.82%	3008	4694	64.08%
2018-09-22	1955	3900	50.13%	2846	4528	62.85%
2018-09-23	2023	3932	51.45%	2871	4531	63.36%
2018-09-24	2252	4059	55.48%	3138	4846	64.75%
2018-09-25	2264	4153	54.51%	3050	5046	60.44%
2018-09-26	2079	4256	48.85%	3209	4532	70.81%
2018-09-27	2135	4199	50.85%	2846	4595	61.94%
2018-09-28	2329	3931	59.25%	2822	5097	55.37%
2018-09-29	2019	4062	49.70%	2978	4742	62.80%
2018-09-30	1871	3944	47.44%	3095	4598	67.31%
2018-10-01	2205	4001	55.11%	2929	4856	60.32%
2018-10-02	1809	4309	41.98%	3034	4366	69.49%
2018-10-03	1404	3923	35.79%	3104	4768	65.10%
Max	2329	4309	60.56%	3209	5097	75.12%
Min	1404	3813	35.79%	2610	3660	55.37%
Avg	2030.56	4059	50.04%	2957.09	4491.13	66.23%

TABLE III: Resource consumption of TestP$_{All}$ and TestP$_{Sel}$. (Units are executor days)

Date	TestP$_{Sel}$	TestP$_{All}$	TestP$_{Sel}$/ TestP$_{All}$[%]
2018-09-26	1595	3468	45.99%
2018-09-27	2066	5897	35.03%
2018-09-28	5180	7886	65.69%
2018-09-29	5819	6673	87.20%
2018-09-30	4236	6287	67.38%
2018-10-01	4546	7474	60.82%
2018-10-02	4564	6866	66.47%
2018-10-03	4108	7799	52.67%
Max	5819	7886	87.20%
Min	1595	3468	35.03%
Avg	4014.25	6543.75	61.34%

resource consumption is not a main concern for Google. The units reported in the table is "executor days", which is roughly the test wall time plus setup time on a single executor. One executor day roughly means an executor is occupied 24 hours by the requesting job. Please note that the executor day is not proportional to the actual test execution time as the execution is paralleled and one execution may occupy hundreds of executors at the same time. Therefore, the average reduced resource consumption (~39%) is slightly less than the average reduced execution time(~50%).

3) RQ3: What is the overhead of TESTSAGE *dependency update and what is the trade-off between the above two metrics and overhead introduced by* TESTSAGE*:* The cost of TESTSAGE mainly comes from three parts: (1) latency introduced by querying the dependency lookup table, (2) the overhead of running tests on XRAY instrumented server, and (3) dependency update for selected tests. The lookup table query latency is usually less than 1 second and it is a one time access for TESTSAGE in a run (Bigtable processes queries in batch). Comparing to the execution time of an entire test class (usually between 5 to 10 minutes), this table query latency is almost negligible. To understand the test execution latency caused by XRAY, we randomly picked a test class and measured its execution time on XRAY instrumented and non-instrumented servers, respectively. In 1000 runs, we measured a 2% increase in running time but the data comes with a high variance. Given that most of the test classes run in 5 to 10 minutes, TESTSAGE could potentially adds a 10 seconds latency to each test class. In order to collect test dependency, TESTSAGE needs to issue three RPC calls to each service under test. The first RPC asks a service to start a tracing interval, the second RPC asks the service to stop the tracing interval, and the final RPC downloads the trace data to client. It also needs to write the updated dependency to the look-up table. Among the four operations, the second and third RPCs are most expensive, which cost 5 seconds to run on average. Therefore, TESTSAGE could potentially introduce 2% increase in testing time. More significantly, TESTSAGE blindly adds ~10 seconds overhead to collect dependency for each test entity. This limits the usage of TESTSAGE on short-running tests as they usually finish within a few seconds. Applying TESTSAGE on fine-grained tests(test method) is also limited because of this fixed per test entity overhead. Therefore, we adopt test-class level RTS in our experiment.

V. DISCUSSION

One-Pass vs. Two-Pass. TESTSAGE can be configured to use either *one-pass* or *two-pass* executions. However, TESTSAGE collects dependency of a test by running the test exclusively

during a XRAY trace interval on a server. This limits the concurrent executions of tests on a server. Therefore, we only adopted *two-pass* executions in practice.

RTS granularity. TESTSAGE collects function-level dependency using XRAY. We can also derive file-level dependency using code index. Based on the evaluation result, function-level granularity is still applicable to large-scale projects. Depending on the tool used to collect test coverage, more granularity levels can be explored in the future.

RTS techniques can also be categorized based on the granularity of *test entity*. A *test entity* can be a test method, a test class, or even a test suite. The nature of TESTSAGE makes it very easy to adjust the granularity of test entity. We can simply run an entire test class or a test method in a trace interval to obtain dependency for the class. What is more interesting is that this feature allows TESTSAGE to choose different test entity granularity for different test. For example, we break down some heavy test classes into multiple sub-groups in practice, and collect dependency separately for them. However, choosing a finer granularity of a test entity in TESTSAGE might not be practical, as starting/stopping traces come with a fixed amount of overhead (shown in Section IV-C3).

Safety. TESTSAGE adopts the architecture of classic function-level dynamic RTS approaches and assumes that XRAY instrumentation is accurate and correct. The safety of RTS techniques have been studied and semi-proved by previous studies. We also discussed the correctness of XRAY in Section II-C. However, note that TESTSAGE is unsafe in one of its experiment setup. In TestP setup (pre-submit test) of our experiment, TESTSAGE selects test using latest computed dependency from TestC. The change information passed to TESTSAGE is computed using developer local change and the code base synced on his/her client. This could potentially make TESTSAGE select tests on outdated dependency. We made this compromise due to the large number of revisions (~400 in a day) and resource/time constraint of pre-submit testing at Google. We mitigate the issue by narrowing the time gap of revisions and running TESTSAGE with updated dependency at post-submit time.

Deployment. Deploying TESTSAGE on an existing web service test is non-trivial and we highlight the following two phases that require most of the work. The first phase is instrumentation. While XRAY instrumentation is straight forward but the compiled binary size could increase by 20% with XRAY. This might cause issues when building an enormous binary on a build environment that has constraints on resources like memory or binary size. The second phase is the network connectivity between TESTSAGE and SUT. During test execution time, TESTSAGE requires network connection with all the service components in SUT as it needs to interact with them through RPC calls. However, a real-world system might hide some of its service components behind a firewall due to privacy and security concerns. Since the original test client only requires connection with the main service component, extra work might be required when deploying TESTSAGE on an existing test.

VI. THREATS TO VALIDITY

Threads to External Validity. The main threat to external validity is that results measured in our study might not be generalized to other projects. To mitigate this threat, we picked a large size web service project consisting of multiple running binaries at Google. Besides, we selected all project revisions within a randomly picked 32 days window. To the best of our knowledge, this is the largest RTS study on a single project. However, because our implementation depends on Google internal infrastructure, it is still unclear if our study can generalize to projects outside of Google.

Threads to Internal Validity. The main threats to internal validity lies in the implementation of TESTSAGE and XRAY. To reduce the threat, we conducted a large number of manual checks for the results. Further more, given the size of the project, we also collect and analyze developer reported bugs of the tool continuously at the company. To verify the correctness of XRAY updates (XRAY is being consistently updated), we also implemented a simple service and tests to collect test dependency computed by the updated XRAY. We compare the result with human generated dependency data on each updated version of XRAY.

Threads to Construct Validity. The main threat to construct validity is the methods we used to evaluate the proposed RTS technique. To mitigate the threat, we measure our study with the three widely used metrics in RTS, i.e., the selected test ratio, the offline testing time (i.e., AE time), and the online testing time (i.e., AEC time).

VII. RELATED WORK

Regression test selection. Regression test selection has been studied for several decades [1], [3], [5]–[11], [34]. Depending on the way the dependency is collected, Prior work on RTS techniques can be categorized into dynamic RTS techniques and static RTS techniques. Rothermel and Harrold [6] proposed the pioneer dynamic RTS for C programs at the basic-block granularity. Harrold [4] then reported a study of basic-block dynamic RTS on Java programs. Orso et al. [15] presented a novel two-phase RTS approach for large Java programs. The first phase is called *partitioning*, in which the RTS technique identifies affected classes and interfaces (a partition of the program) using CFG analysis. The second phase is referred as *selection*, in which the RTS technique does more detailed CFG analysis on the partition to select test cases.

To mitigate the large overhead incurred by finer-grained analysis, a number of studies have been done at coarser-granularies. Ren et al. [16] and Zhang et al. [9] proposed function-level RTS using code analysis. They collect function/field dependency for tests at old revision and select tests in the new revision if any of the depended function/field are updated. Our study builds on top their ideas and introduces a mechanism to collect function/field test dependency for distributed binaries. Besides function-level RTS techniques, even coarser grained RTS have also been explored. Gligoric et al. [17] proposed file-level RTS for Java programs. Their technique does fast checksum computation for class files, and collects test dependency on the encoded class files. If a file checksums is updated in a new revision, all tests depend on the file are selected. More recently, Zhang [18] observed that a hybrid file and method level RTS can be even more cost-effective. Vasic et al. [35] also compared file-level RTS with module-level RTS for .NET programs. They found out that file-level RTS is more cost-effective than module-level RTS. Celik et al. [36] designed dynamic file-level RTS

across JVM boundaries. Note that our work is different from their work: (1) the system in their RTS technique is still confined to a single OS kernel; (2) the system in our study is distributed in nature, loosely coupled, and reside at separate locations. Despite the smaller overhead, file-level RTS may be imprecise for web service testing due to the coarse-grained analysis and end-to-end coverage of test cases. In our work we find that a significant number of source files are accessed by all tests and these files are insensitive to file-level RTS, e.g., an *util.cpp* file may contain different utility functions for different tests. Therefore, in our work, we adopt function-level RTS and prove that it is feasible to collect dependency for large web service projects at this granularity.

In addition to dynamic RTS, static RTS techniques using static analysis have also been proposed to make up for the shortcoming of dynamic RTS techniques. Kung et al. [14] proposed the pioneer static RTS for C++ programs based on *class firewall*. Their technique aims to find the affected classes for the new revision of the software using class-level static analysis. Ryder and Tip [37] studied static RTS at the function level using static call graphs. Recently, Legunsen et al. [13], [38] push static RTS techniques forward and make the testing time of static RTS techniques comparable state-of-the-art dynamic file-level RTS. But this technique is unsafe due to reflections. Although static RTS techniques cannot be directly applied on web service testing, it is still an interesting direction to explore as it does not require code instrumentation. Instrumenting a large number of distributed binaries is challenging and therefore we exclude some dependencies out of our SUT for TESTSAGE.

Regression test selection on web service. RTS technique for web services test is more challenging than traditional RTS because the tests and services are inherently distributed in nature and are loosely-coupled. Besides, web services are usually composed of other services, making the dependency collection difficult for RTS. Therefore, RTS for web services [10], [21]–[24] have not been fully explored in prior studies. We briefly summarize the prior work as follows.

Xu et al. [21] firstly proposed an RTS technique for web applications based on slicing [39]. The technique assumes that changes on a web application can be divided into a list of basic classes on a static HTML page. An HTML page in an web application can be a data or a hyperlink dependency on other pages in the application. The technique then computes an extended system dependence graph (SDG) model of the web application using the data dependencies. RTS selects tests that execute the potentially affected web elements. Though precise, SDG-based slicing RTS techniques are unsafe [1]. Tarhini et al. [24] proposed a safe RTS technique for web applications. The technique models a web application in two hierarchical levels. Given two models representing two revisions of the web application, the RTS technique generates two test suites for each revision and finds the difference between the initial test suite and the new test suite. The differential set of test cases are selected for regression testing. This technique is safe because it selects every test case that produces a different behavior in the modified system. However, the technique is not applicable on large and complex web services simply because there are too many states of the system to model. We cannot handle the overhead to model the system and to generate the test cases. Lin et al. [22] proposed a safe RTS technique for Java web services by merging the client side code and server side code into a single program. The technique uses Web Service Description Language (WSDL) specifications to create local proxy objects that simulate communications between the client application and the web service on the server side. It then applies an existing RTS technique [4] on the transformed program. Ruth et al. [10], [23] proposed a generalized RTS technique for web services based on analysis of control flow models. The technique traverses the global CFGs for the old and the updated services and identify the nodes of the graph which are changed. The test cases which execute the control flow edges that can be reached from the modified nodes are selected. Both of the above two RTS techniques suffer from the same reason, the non-trivial overhead to model complex web services. Regardless whether manual effort is required to model the service and to generate the test for these techniques, providing a model for each update of a complex web service is unsustainable. TESTSAGE differs from the prior work as it computes the dynamic function-level test dependency. The granularity of TESTSAGE offers a good trade-off between selection precision and overhead, making it scalable to large-scale systems in Google.

VIII. CONCLUSIONS

This paper presents the first dynamic RTS technique based on function-level dependency collected on web service test, TESTSAGE. Unlike existing dynamic RTS techniques, TESTSAGE is able to collect white-box function-level test dependency for test runs in a different environment from the code under test. TESTSAGE does this by instrumenting the server side code with a C/C++ function tracing tool called XRAY. At runtime, TESTSAGE runs tests with their own traces on server, and converts the trace data into test dependency. TESTSAGE stores the dependency data in a separate storage to select tests in new revision. TESTSAGE was enabled on Google Assistant backend service, a single web service project consisting of multiple running binaries in December 2017, and has been serving continuously at the company. We gathered TESTSAGE metrics from a randomly picked 32 days window to evaluate the proposed approach. For the 11028 revisions of pre-submit testing, TESTSAGE skips 55% of the available tests on average; for the 360 revisions of post-submit testing, TESTSAGE skips 41% of the available tests on average. The study shows that fine-grained analysis is feasible for large-scale web-service testing. Our work also demonstrate the possibility of integrating dynamic RTS techniques into large-scale web-service testing.

ACKNOWLEDGMENT

We thank Dean Michael Berris, Eric Anderson and Ning Wang for their advise and feedback on XRay, Moyang Zhang and Thanapong Lertpanyavit for their help on implementing TestSage. The project is partially supported by National Science Foundation grants CCF-1763906 and CCF-1718903.

REFERENCES

[1] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: a survey,” *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120.

[2] “Tools for continuous integration at google scale,” 2011. https://www.youtube.com/watch?v=b52aXZ2yi08.

[3] “Testing at the speed and scale of google,” 2011. http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

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

[5] E. Engström, P. Runeson, and M. Skoglund, “A systematic review on regression test selection techniques,” *IST*, vol. 52, no. 1, pp. 14–30, 2010.

[6] G. Rothermel and M. J. Harrold, “A safe, efficient regression test selection technique,” vol. 6, (New York, NY, USA), pp. 173–210, ACM, Apr. 1997.

[7] E. Engström and P. Runeson, “A qualitative survey of regression testing practices,” in *Product-Focused Software Process Improvement* (M. Ali Babar, M. Vierimaa, and M. Oivo, eds.), (Berlin, Heidelberg), pp. 3–16, Springer Berlin Heidelberg, 2010.

[8] R. Feldt, J. K. Gorantla, C. White, W. Zhe, and G. Wikstrand, “Dynamic regression test selection based on a file cachean industrial evaluation,” in *2009 International Conference on Software Testing Verification and Validation(ICST)*, vol. 00, pp. 299–302, 04 2009.

[9] L. Zhang, M. Kim, and S. Khurshid, “Localizing failure-inducing program edits based on spectrum information,” in *ICSM*, pp. 23–32, IEEE, 2011.

[10] M. Ruth, S. Oh, A. Loup, B. Horton, O. Gallet, M. Mata, and S. Tu, “Towards automatic regression test selection for web services,” in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 2, pp. 729–736, July 2007.

[11] A. Srivastava and J. Thiagarajan, “Effectively prioritizing tests in development environment,” in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA ’02, (New York, NY, USA), pp. 97–106, ACM, 2002.

[12] G. Rothermel and M. J. Harrold, “A safe, efficient regression test selection technique,” *TOSEM*, vol. 6, no. 2, pp. 173–210, 1997.

[13] 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 *FSE*, pp. 583–594, ACM, 2016.

[14] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, “Class firewall, test order, and regression testing of object-oriented programs,” *JOOP*, vol. 8, no. 2, pp. 51–65, 1995.

[15] A. Orso, N. Shi, and M. J. Harrold, “Scaling regression testing to large software systems,” in *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 241–251, ACM, 2004.

[16] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, “Chianti: a tool for change impact analysis of java programs,” in *ACM Sigplan Notices*, vol. 39, pp. 432–448, ACM, 2004.

[17] M. Gligoric, L. Eloussi, and D. Marinov, “Practical regression test selection with dynamic file dependencies,” in *ISSTA*, pp. 211–222, ACM, 2015.

[18] L. Zhang, “Hybrid regression test selection,” in *ICSE*, pp. 199–209, 2018.

[19] “Google assistant,” 2016. https://en.wikipedia.org/wiki/Google_Assistant.

[20] “Google assistant on more than 400 million devices in 2017,” 2018. https://ppc.land/google-assistant-on-more-than-400-million-devices-in-2017/.

[21] L. Xu, B. Xu, Z. Chen, J. Jiang, and H. Chen, “Regression testing for web applications based on slicing,” in *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, pp. 652–656, Nov 2003.

[22] F. Lin, M. Ruth, and S. Tu, “Applying safe regression test selection techniques to java web services,” in *International Conference on Next Generation Web Services Practices*, pp. 133–142, Sept 2006.

[23] S. Tu and M. Ruth, “A safe regression test selection technique for web services,” in *Internet and Web Applications and Services, International Conference on(ICIW)*, vol. 00, p. 47, 05 2007.

[24] A. Tarhini, H. Fouchal, and N. Mansour, “Regression testing web services-based applications,” in *IEEE International Conference on Computer Systems and Applications, 2006.*, pp. 163–170, March 2006.

[25] D. M. Berris, A. Veitch, N. Heintze, E. Anderson, and N. Wang, “Xray: A function call tracing system,” tech. rep., 2016. A white paper on XRay, a function call tracing system developed at Google.

[26] “Llvm xray,” 2016. https://llvm.org/docs/XRay.html.

[27] C. Lattner and V. Adve, “Llvm: a compilation framework for lifelong program analysis amp; transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, March 2004.

[28] F. Henderson, “Software engineering at google,” *CoRR*, vol. abs/1702.01715, 2017.

[29] R. Potvin and J. Levenberg, “Why google stores billions of lines of code in a single repository,” *Commun. ACM*, vol. 59, pp. 78–87, June 2016.

[30] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo, “Testtube: A system for selective regression testing,” in *Proceedings of the 16th International Conference on Software Engineering*, ICSE ’94, (Los Alamitos, CA, USA), pp. 211–220, IEEE Computer Society Press, 1994.

[31] “grpc.” https://en.wikipedia.org/wiki/GRPC.

[32] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” in *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 205–218, 2006.

[33] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at google with borg,” in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys ’15, (New York, NY, USA), pp. 18:1–18:17, ACM, 2015.

[34] 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, no. 12, pp. 1357–1383, 2013.

[35] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric, “File-level vs. module-level regression test selection for .net,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), pp. 848–853, ACM, 2017.

[36] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, “Regression test selection across jvm boundaries,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), pp. 809–820, ACM, 2017.

[37] B. G. Ryder and F. Tip, “Change impact analysis for object-oriented programs,” in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 46–53, ACM, 2001.

[38] O. Legunsen, A. Shi, and D. Marinov, “Starts: Static regression test selection,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 949–954, Oct 2017.

[39] M. Weiser, “Program slicing,” in *Proceedings of the 5th International Conference on Software Engineering*, ICSE ’81, (Piscataway, NJ, USA), pp. 439–449, IEEE Press, 1981.