Regression Test Selection for TizenRT

Ahmet Celik
University of Texas at Austin (USA)
ahmetcelik@utexas.edu

Young Chul Lee
Samsung (South Korea)
yc207.lee@samsung.com

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

ABSTRACT

Regression testing – running tests after code modifications – is widely practiced in industry, including at Samsung. Regression Test Selection (RTS) optimizes regression testing by skipping tests that are not affected by recent code changes. Recent work has developed robust RTS tools, which mostly target managed languages, e.g., Java and C#, and thus are not applicable to large C projects, e.g., TizenRT, a lightweight RTOS-based platform.

We present Selfection, an RTS tool for projects written in C; we discuss the key challenges to develop Selfection and our design decisions. Selfection uses the objdump and readelf tools to statically build a dependency graph of functions from binaries and detect modified code elements. We integrated Selfection in TizenRT and evaluated its benefits if tests are run in an emulator and on a supported hardware platform (ARTIK 053). We used the latest 150 revisions of TizenRT available on GitHub. We measured the benefits of Selfection as the reduction in the number of tests and reduction in test execution time over running all tests at each revision (i.e., RetestAll). Our results show that Selfection can reduce, on average, the number of tests to 4.95% and end-to-end execution time to 7.04% when tests are executed in the emulator, and to 5.74% and 26.82% when tests are executed on the actual hardware. Our results also show that the time taken to maintain the dependency graph and detect modified functions is negligible.

CCS CONCEPTS

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

KEYWORDS

Regression test selection, TizenRT, static dependency analysis

ACM Reference Format:
Ahmet Celik, Young Chul Lee, and Milos Gligoric. 2018. Regression Test Selection for TizenRT. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3236024.3275527

1 INTRODUCTION

Regression testing – running available tests to check correctness of recent code changes – is widely practiced in industry, including at Samsung. Despite the widespread use, regression testing is costly due to a large number of tests and large number of changes [1, 2]. The high cost of regression testing impacts developers' productivity, and developers may miss bugs if they *manually* select to run only a subset of available tests [7, 15].

Regression test selection (RTS) techniques optimize regression testing by automatically detecting and skipping to rerun a subset of tests whose behavior is not affected by recent code changes [10, 21, 22, 31]. Traditionally, RTS techniques track dependencies, for each test, on code elements (e.g., statements, basic blocks, functions, or files) and skip from the run (in a new project revision) those tests that do not depend on any of the modified code elements.

RTS has been studied for over three decades, and researchers and practitioners have developed RTS *techniques* for various programming languages, including C/C++ [9, 22], Java [14, 19, 32], C# [30], etc. However, there are only a few RTS *tools* available that implement these techniques. Most of the available tools target managed code, i.e., languages that compile to Java bytecode [14, 19] or .NET CLR [30]. Our work is mainly motivated by the lack of an RTS tool for the C programming language and numerous questions about potential benefits that such tools could provide.

We present the design, implementation, and evaluation of Selfection, a novel RTS tool for projects written in C, which are compiled to ARM ELF. Selfection uses static analysis to collect dependencies for each test by utilizing the call graph, i.e., each test depends on *functions* that might be transitively invoked from the test. In a new project revision, Selfection detects modified functions, by comparing the current checksum with the old checksum for each function, and propagates the information about non-modified functions to identify tests to skip. Selfection performs the analysis on an executable image by extending objdump and readelf tools.

To evaluate the benefits of Selfection, we integrated the tool in TizenRT [27], a lightweight runtime operating system developed by Samsung. We used the latest 150 revisions and replayed the code changes. We measured the benefits of Selfection as the reduction in the number of executed tests, as well as the reduction in end-to-end execution time compared to *RetestAll* (i.e., running all tests at each revision). To execute the tests we used two environments: Qemu emulator and an actual hardware board; these environments are used by TizenRT developers, and the set of tests that can run in each environment differs, e.g., network tests only run on the board.

Our results for runs with the Qemu emulator show that Selfection reduces, on average, the number of executed tests and test execution time *to* 4.95% and 7.04%, respectively. Our results for runs on the board show that Selfection reduces, on average, the number of executed tests and test execution time *to* 5.74% and 26.82%, respectively. Finally, our results show that time taken to maintain the dependency graph and select tests is negligible.

2 SELFECTION

Selfection follows traditional RTS tools, most notably TestTube [9], and includes three phases: analysis, execution, and collection. We

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3275527

describe the task of each phase, the way we implement these phases, and the reasoning behind our implementation decisions.

2.1 Phases

Analysis phase (A-Phase). The goal of the *analysis phase* is to select tests to be run in a new project revision. In other words, the goal is to detect tests that are *affected* by recent code changes. To detect affected tests, Selfection analyzes the executable image, extracts functions, and computes the checksum of each function; our current implementation uses Adler32 algorithm to compute the checksum, but any other algorithm can be used. Once modified functions are detected, Selfection computes the transitive closure using a dependency graph collected in prior C-Phase (see below) to find affected tests. Selfection uses symbols instead of absolute addresses for function calls and global variables referenced in a function body, and relative addresses for branches inside a function. If these absolute addresses were not ignored, even a simple change, e.g., adding a line of code could change checksum of every function.

Execution phase (E-Phase). The goal of the execution phase is simply to run selected tests. Although this step is rather trivial for projects that use testing frameworks, such as JUnit, xUnit, or similar, because those frameworks support test filtering (i.e., excluding a subset of tests), there is no straightforward way to exclude some tests in C projects, as those tests are frequently explicitly invoked from the main function. To enable selection of some tests we pass, as arguments to main, the list of test functions that should be skipped. This approach ensures that newly added tests are always run. Any project that would like to utilize Selfection would have to adjust its test code to invoke our filtering library instead of invoking tests directly. We automatically modified TizenRT test code to include necessary invocations for the sake of evaluation; we describe the details of our experiment setup in Section 3.

Collection phase (C-Phase). The goal of the C-Phase is to collect dependencies for each test, which will be used in the next test run (and next A-Phase). To collect dependencies, Selfection statically analyzes the executable image and builds a function call graph, which is then used to find transitive dependencies for each test. Selfection uses objdump and readelf tools to build the call graph. The dependency data is maintained in the root directory of the project. If Selfection is integrated in a continuous integration service, e.g., TravisCI, the dependency data could be kept either in the cache or as an external repository on GitHub. In the dependency data, we associate checksum with each function. The persisted data is used in the A-Phase for the next revision.

2.2 Design/Implementation Decisions

Source vs. binary analysis. Selfection, as mentioned earlier, analyzes an executable image in A-Phase and C-Phase. An alternative approach would be to analyze source code of the project, e.g., via a compiler plugin, to build the call graph and dependencies for tests. Both approaches have advantages and disadvantages. Binary analysis may be seen as more generic, because any language (e.g., OCaml) that compiles to the same executable format would be supported. However, depending on the architecture and compiler used, the binaries frequently differ. On the other hand, analyzing source code would require dependencies on a specific compiler platform. Although our preference would be a compiler plugin, because it would simplify the implementation, we chose to analyze binaries simply because the compiler used to compile TizenRT (gcc-arm-none-eabi-4_9-2015q3) does *not* support compiler plugins. Additionally, GCC compiler plugin infrastructure, in general, is poorly documented.

Static vs. dynamic analysis. Selfection *statically* analyzes binaries in A-Phase and C-Phase; static analysis overapproximates the set of dependencies [19]. An alternative would be to *dynamically* collect dependencies for each test. In other words, while a test is running, we could collect dependencies on functions that are executed, which would improve precision of the technique, i.e., test would depend only on functions that are actually used. There are several (technical) reasons why we chose static analysis. First, dynamic approach would require code instrumentation. Considering that our target project – TizenRT – is run in restricted environment, using standard instrumentation frameworks, e.g., Dyninst, would not be feasible. Second, dynamic instrumentation would require extra memory to maintain dependencies and store those dependencies to disk. Extra memory for keeping dependencies could be too large for the environment used to run TizenRT tests (e.g., ARTIK 053). Finally, transferring collected dependencies from a board (and even from an emulator) at the end of each test run would introduce additional technical challenges and cost.

Table 1: Test Suites Available in TizenRT, Execution Time of each Test Suite on ARTIK 053, and Number of Tests in each Test Suite; Note That the Execution Time Depends on Platform/Environment and can be Much Longer

Test Suite	Time [s]	#Test
Arastorage I-Tests	2.02	54
Arastorage U-Tests	1.01	46
Drivers Tests	3.02	26
Filesystem Tests	23.21	76
Sys IO U-Tests	4.04	90
Network Tests	2.02	180
Kernel Tests	136.26	405
$\sum$	171.58	877

3 CASE STUDY

To assess the benefits of Selfection, we answer the following research questions:

RQ1: How many tests does Selfection skip on average across a large number of revisions?

RQ2: What is the reduction, on average, in end-to-end test execution time across a large number of revisions?

RQ3: How does time for A-Phase, E-Phase, and C-Phase compare to other build steps?

We first describe the subject used in our case study, the experiment setup, and then answer the research questions.

3.1 Subject

We use TizenRT [27] developed by Samsung as the main case study. At the latest revision (0a3d2deb), available at the time of our study, TizenRT has 5049 functions and 877 test functions/cases. Table 1 shows, for each test suite, the number of test cases and execution time. Note that a set of test suites differs for various platforms, and

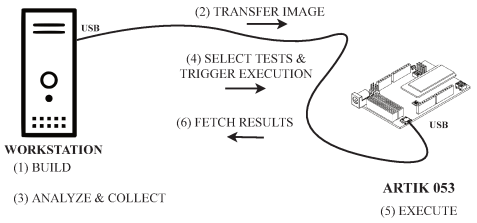


Figure 1: Experiment setup with ARTIK 053

the numbers reported in Table 1 are obtained by running tests on ARTIK 053; execution time may be substantially different on other hardware platforms used at Samsung.

3.2 Experiment Setup

We briefly describe our experiment setup. Specifically, we describe the way we prepare TizenRT for evaluation, data collected during experiments (i.e., independent variables), metrics used to evaluate the benefits (i.e., dependent variables), and environments used to execute the experiments.

Annotated repository. Recall (Section 2) that SELFECTION skips non-selected tests by passing the list of test cases to skip to the main function. We expect that these changes would be performed by developers when they integrate SELFECTION into their projects. As we performed the evaluation retroactively on the TizenRT repository, those changes are not available, so our first step was to rewrite the repository and insert the appropriate filtering code. The result of rewriting is a new repository that contains the same files and changes as the original repository, but also includes code that guards/skips tests. Specifically, we perform the following steps:

a) Clone the latest revision of the repository from GitHub; we will call this repository OriginalRepo;
b) Go 150 revisions back into history; we consider, as in recent work on RTS, only revisions that are on the master branch (i.e., `git log --first-parent`);
c) Create a new repository, named AnnotatedRepo, that will be used to host annotated code;
d) Copy all the files from OriginalRepo to AnnotatedRepo;
e) Annotate all the tests by surrounding each test with code that will guard the test execution; we automate this step with several bash scripts;
f) Commit all files in AnnotatedRepo;
g) If the current revision in OriginalRepo is the latest revision, finish the process, otherwise checkout the next revision and go to step d).

Data collection. To answer aforementioned questions, we performed the steps below on AnnotatedRepo. We follow, as closely as possible, recent work on RTS [14, 19, 30].

a) Checkout the oldest revision (from the used set of revisions);
b) Execute tests (using RetestAll) and collect number of executed tests ($N_{RetestAll}$), as well as test execution time ($T_{RetestAll}$);
c) Run SELFECTION to select tests, execute selected tests, and collect new dependencies; we collect number of executed tests ($N_{\text{SELFECTION}}$), as well as execution time for all phases: $T_{A-Phase}$, $T_{E-Phase}$, and $T_{C-Phase}$;
d) If there are no more revisions, then finish the process; otherwise checkout the next revision and go to step b).

We will use $T_{\text{SELFECTION}}$ to denote the *end-to-end* execution time, i.e., $T_{A-Phase} + T_{E-Phase} + T_{C-Phase} + T_{etc.}$; $T_{etc.}$ denotes execution time for other build steps, e.g., compilation.

Dependent variables. Using the collected data we compute two variables. First, we compute *test selection ratio* (*Sel*[%]), as the ratio of the number of selected tests and the total number of tests, i.e., $Sel[\%] = N_{\text{SELFECTION}} / N_{RetestAll} * 100$.

Second, we compute savings in end-to-end execution time. Arguably, the most important metric for developers used to evaluate an RTS technique is the reduction in end-to-end execution time. We compute reduction in time as the ratio of end-to-end time taken by SELFECTION and end-to-end time taken by RetestAll, i.e., $time[\%] = T_{\text{SELFECTION}} / T_{RetestAll} * 100$.

Execution environments. We use two execution environments (QemuEnv and ArtikEnv) to run the experiments; both environments, and several other platforms, are used by TizenRT developers.

QemuEnv uses Qemu [20] to emulate necessary hardware and run the tests. Only a subset of tests – Kernel Tests (in Table 1) – is enabled on Qemu. We installed Qemu on a machine with an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz with 16GB of RAM, running Ubuntu 17.10. We set up our experiment to automatically build TizenRT, transfer the image to a Qemu instance, execute tests, and log the execution time and test results; this setup was not automated prior to our work. Note that A-Phase and C-Phase are run on the host machine, as already discussed in Section 2.

ArtikEnv uses the actual hardware – an ARTIK 053 board [3] – to execute tests. Therefore, all tests in Table 1 are enabled. Figure 1 illustrates ArtikEnv. We connected the board, via serial port, to the machine described in the previous paragraph. Our scripts automatically build the project, transfer the image to the board, select tests, initiate the test runs, and collect logs and test results. As in QemuEnv, A-Phase and C-Phase are run on the host machine.

For each test case run in either QemuEnv or ArtikEnv, we compute percentage of functions on which the test case depends. Figures 2a and 2b show distribution of the percentage of test dependencies. We can see that tests frequently depend on a small number of functions, which is an ideal scenario for using an RTS tool.

3.3 Results

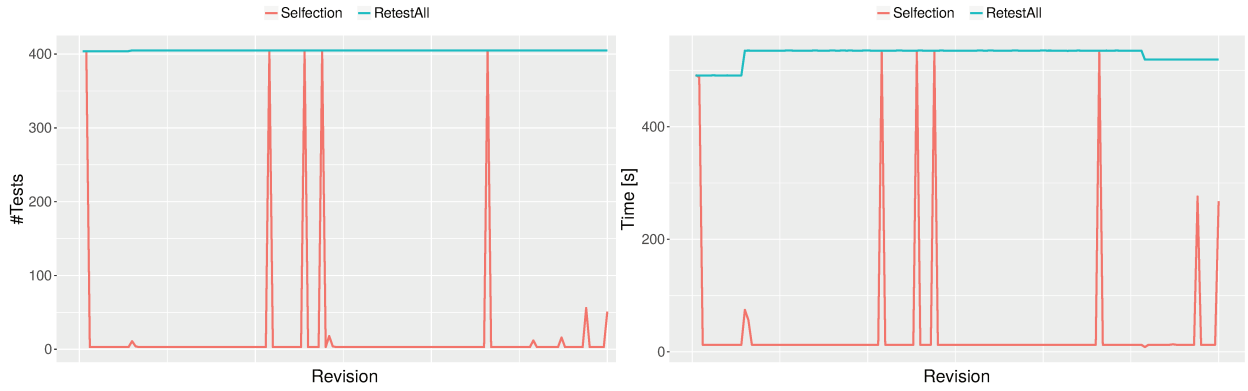
3.3.1 Average Savings in the Number of Tests. Plots in figures 3a and 4a show the number of executed tests, using RetestAll and SELFECTION, at each revision for QemuEnv and ArtikEnv, respectively. Note that the set of tests run in QemuEnv is not necessarily a subset of tests run on ArtikEnv, because the configuration in Makefiles differ. Also, we were unable to run builds for ArtikEnv for first 25 revisions. We can observe that for most revisions, SELFECTION selects very small number of tests (if any). For each revision, we compute *Sel*[%]. Our results show that *Sel*[%], on average across all revisions, is 4.95% and 5.74% for QemuEnv and ArtikEnv, respectively.

3.3.2 Savings in the Execution Time. Plots in figures 3b and 4b show end-to-end execution time, using RetestAll and SELFECTION, at each revision for QemuEnv and ArtikEnv, respectively. Clearly, execution with SELFECTION is substantially faster than using RetestAll[1]. As expected, at the first revision, SELFECTION takes equal or more

[1]Time for SELFECTION increases initially due to the lack of cleanup in `filesystem_tc`, which was added by developers in revision `07b740ae`.

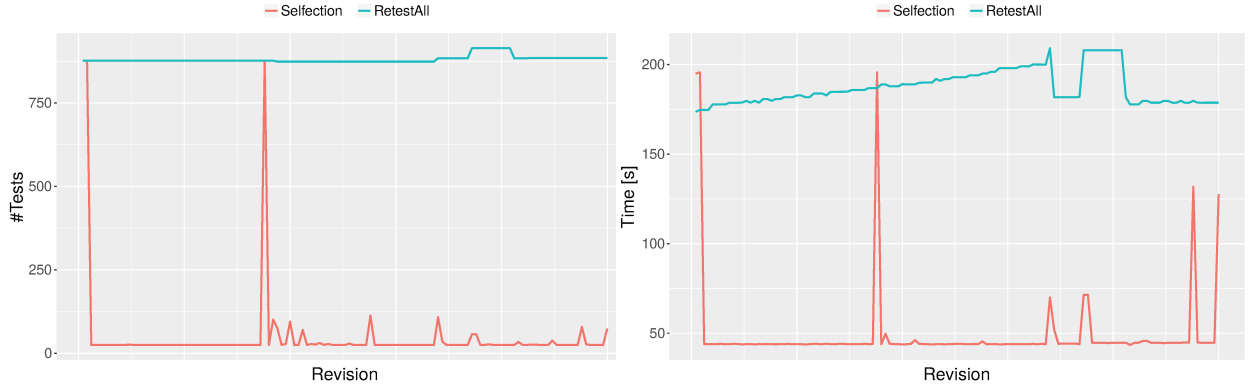
(a) QemuEnv (b) ArtikEnv

Figure 2: Distribution of the percentage of dependencies per test case



(a) Number of tests (b) End-to-end execution time

Figure 3: Number of executed tests (left) and end-to-end execution time (right) using RetestAll and SELFECTION when tests are run using QemuEnv



(a) Number of tests (b) End-to-end execution time

Figure 4: Number of executed tests (left) and end-to-end execution time (right) using RetestAll and SELFECTION when tests are run using ArtikEnv

time than RetestAll due to the analysis cost. For each revision we compute *time*[%]. Our results show that *time*[%], on average across all used revisions, is 7.04% and 26.82% of the RetestAll time for QemuEnv and ArtikEnv, respectively. The reduction in test time for ArtikEnv is lower than for QemuEnv because testing is not the only phase that dominates the build, as we discuss below.

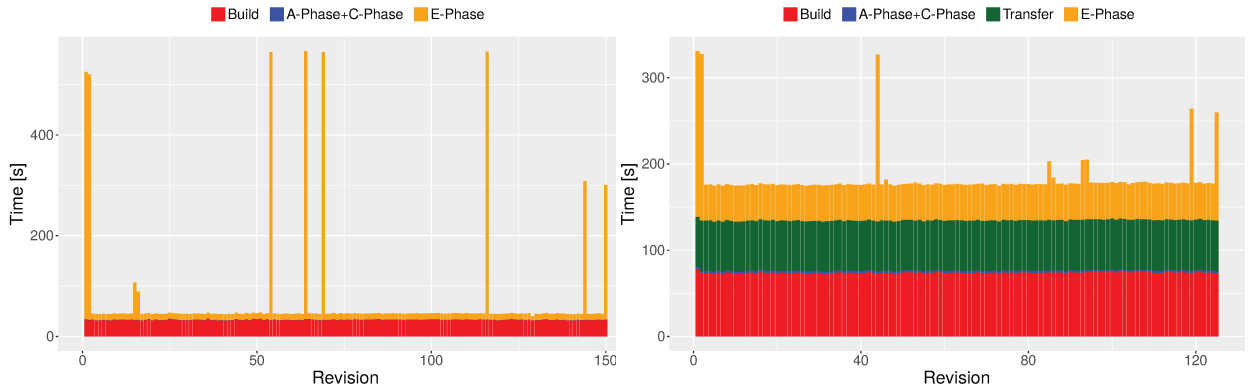
We also observed, in Figure 4b, an interesting case (between `f1f53f6f` and `d99f5451`) when execution time drops sharply. This happens due to a code change that removes several sleep statements.

3.3.3 Execution Time for Various Build Steps. We were curious not only about the end-to-end execution time, which was discussed in the previous answer, but also about the portion of time taken by various SELFECTION phases. Specifically we compare time for A-Phase+C-Phase, E-Phase, build time (except time to run SELFECTION and tests), and transfer time (in case of the ArtikEnv). Stacked plots in figures 5a and 5b show time for each step. We can observe that A-Phase+C-Phase takes negligible time. Interestingly, as only a few tests are selected on average, test execution time for TizenRT becomes faster than building the project and transferring the binary to the board. Future work could optimize the transfer time by incrementally patching previously transferred binaries [8].

4 DISCUSSION

Different sets of tests. We inspected several revisions used in our experiments to confirm the correctness of selection. For example, for the middle two revisions when SELFECTION selected many tests for QemuEnv (c560cf79 and 93b205), but only a few for ArtikEnv, we found that those changes mostly impact binaries that are run in the emulator. Specifically the change is in `up_assert.c` file, which is not included in the binary run in ArtikEnv. In other words, some part of code (or some files) are included in the binary depending on the target platform.

Test-order dependencies. The order in which tests are executed may impact the results of test execution [5, 16]. Therefore, selecting



(a) QemuEnv (b) ArtikEnv

Figure 5: Execution time for various build phases, at each revision, when tests are run using SELFECTION

only some tests may expose an unexpected behavior. We have encountered one of these cases in our experiments. Namely, we observed kernel panic due to an illegal memory access occasionally if only one test is selected in `arastorage_itc`. Interestingly, we found that there is a bug in the test case as the test should deinitialize the database after removing relations (rather than the other way around). Our patch was accepted by Samsung developers [28].

Precision and safety. RTS is considered *safe* if it guarantees that all affected tests are selected and *precise* if it selects only those tests that are affected (but no other tests). We currently provide no strong guarantees about safety of SELFECTION, because we are aware that specific type of changes – a change to a global variable of non-primitive type and a change to an array without symbolic name – may not be detected by our tool. We leave it as future work to improve and test safety of the tool; combining analysis of ELF and source code would solve most of the problems. SELFECTION could also be more precise if we were to track dependencies on individual statements; however, considering the current positive results, improving precision is not an immediate goal of our work.

Excluded tests and always executed tests. Ten test cases in the `sysio_itc` test suite are excluded (i.e., commented out) from our experiments as their block forever even *without* SELFECTION. We believe that this has no impact on our conclusions, as the number of these test cases are rather small compared to the total number of tests. We are in touch with Samsung developers about this issue, and we hope to enable experiments with those tests in the near future. On the other hand, we always run some tests in the `network_tc` and `arastorage_utc` test suites. This was necessary as there are test-order dependencies, e.g., a test that is querying a database depends on another test that is initializing the database.

Future work. Although we made substantial progress towards practical RTS tool for projects written in C, there are several directions for future work. We plan to (1) improve safety of our tool by collecting dependencies on global variables, detecting usage of function pointers, etc.; (2) evaluate SELFECTION on other projects; and (3) optimize transfer of binaries to the board.

5 THREATS TO VALIDITY

External. Our results may not generalize beyond TizenRT. We claim no such generalization, and the evaluation to other C projects remains as a future work. Our goal in this paper was to report on our experience on building RTS to support TizenRT in the first place and document design decisions and challenges.

For each environment – QemuEnv and ArtikEnv – we use only a single host machine. The savings could different on different platforms. Additionally, although TizenRT tests could be run on different hardware boards, Samsung developers confirmed that ARTIK 053 is among the most relevant at the moment, as they are heavily using this platform.

Internal. SELFECTION and the scripts we wrote to perform the experiments may contain bugs. To mitigate this threat, we extensively tested the tool and inspected the results of our experiments. Interestingly, by inspecting one of the outliers, we discovered the bug in TizenRT that was described in Section 4.

Construct. We evaluated SELFECTION on 150 revisions in QemuEnv and 125 revisions in ArtikEnv. Using different number of revisions or a different sequence of revisions could lead to different conclusions. To mitigate this threat, we used the latest available revisions (at the time of our experiments), and we went into history as far as we could before the build would start failing or our scripts for rewriting the repo would be invalidated with changes in the repo. In the future, we plan to further expand the sequence of revisions, although building old revision is known to be challenging [29].

In our experiments, we compare SELFECTION only to RetestAll, although many RTS techniques have been proposed over the years. To the best of our knowledge, no RTS tool is available for C/C++ projects. Even if there was a tool for C/C++ projects, it would likely not be readily applicable to TizenRT. Our contribution is the experience on bringing RTS into TizenRT.

6 RELATED WORK

We briefly discuss the most related work on RTS, evaluation of regression testing tools in industry, and work on build systems.

RTS techniques. Several recent survey papers extensively discuss work and progress on RTS [6, 11, 12, 31]. Rothermel and Harrold [23, 24] presented a test selection algorithm based on control dependency graph. TestTube [9] combines static and dynamic analysis and builds dependencies of tests on functions. Although SELFECTION is directly inspired by TestTube, SELFECTION does not use code instrumentation for reasons described earlier. Recently,

there has been substantial effort to enable RTS with coarse-grained dependencies to work with managed languages and perform large scale evaluations. Gligoric et al. [14] presented Ekstazi, an RTS tool for projects that compile to Java bytecode. Ekstazi tracks dynamic dependencies on class level [25]. Several studies have shown that Ekstazi can reduce end-to-end time by over 50%, and several companies and open-source projects adopted the tool. Work by Legunsen et al. [19] implemented and evaluated RTS with static class dependencies. Their results showed similar savings to Ekstazi, with small negative impact on safety of the technique. The idea behind Ekstazi was reimplemented for .NET and evaluated together with Microsoft developers [30]. Unlike most of the recent work on RTS, our focus was on C projects and specifically on evaluating RTS for TizenRT.

Regression testing tools in practice. Srivastava and Thiagarajan [26] implemented Echelon, a test case prioritization tool that analyzes binaries and prioritizes tests based on the number of basic blocks that they cover. Elbaum et al. [10] proposed an approach to perform (unsafe) RTS in the pre-submit phase; this RTS selects a set of tests that failed in the given time window. Herzig et al. [17] introduce THEO, a tool for accelerating testing process based on a cost model. Our work differs, as our goal was to make a step towards an RTS technique for C.

Build systems and continuous integration systems. Many modern build systems, e.g., Bazel [4], Condor [13], etc., compute static (file) dependencies for each target (or those dependencies are explicitly provided by developers). As these systems keep dependencies for each target, they are commonly imprecise, i.e., they may run more tests than necessary. Our work improves precision for C projects, as we detect changes on function level. Hilton et al. [18] studied usage, cost, and benefits for continuous integration (CI). We plan to integrate Selfection in TizenRT to run as part of CI.

7 CONCLUSIONS

We presented Selfection, a novel regression test selection (RTS) tool for projects written in C. Selfection implements static function-level RTS and analyzes binaries to collect dependencies and find affected tests. To evaluate Selfection, we integrated the tool in the latest 150 revisions of TizenRT, an open-source project developed by Samsung. We measured savings in terms of the number of executed tests and test execution time compared to RetestAll (i.e., running all tests from scratch for each revision). We used two environments to execute tests: Qemu emulator and the actual hardware board (ARTIK 053). Our results for Qemu emulator show that Selfection reduces the number of tests and end-to-end execution time to 4.95% and 7.04%, on average, compared to RetestAll. Our results for ARTIK 053 show that Selfection reduces the number of tests and time to 5.74% and 26.82%, on average, compared to RetestAll. We are currently working closely with Samsung developers to deploy Selfection at the company. We believe that extending Selfection with support for other binary formats, or creating a variant that works as a compiler plugin, can result in a valuable tool for many other C developers.

Acknowledgments. We thank Owolabi Legunsen, Pengyu Nie, Karl Palmskog, and Chenguang Zhu for their feedback on this work. This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1566363, CCF-1652517, CCF-1704790, and by a Samsung Global Research Outreach Award.

REFERENCES

[1] Testing at the speed and scale of Google. http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.
[2] Tools for Continuous Integration at Google Scale. http://www.youtube.com/watch?v=b52aXZ2yi08.
[3] Artik Home Page. Samsung ARTIK IoT Platform. https://www.artik.io/modules/artik-05x.
[4] Bazel Home Page. Build and test software of any size, quickly and reliably. https://bazel.build.
[5] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient Dependency Detection for Safe Java Test Acceleration. In *FSE*. 770–781.
[6] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. 2011. Regression Test Selection Techniques: A Survey. *Informatica (Slovenia)* 35, 3 (2011), 289–321.
[7] Vincent Blondeau, Anne Etien, Nicolas Anquetil, Sylvain Cresson, Pascal Croisy, and Stéphane Ducasse. 2017. What are the Testing Habits of Developers? A Case Study in a Large IT Company. In *ICSME*.
[8] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build System with Lazy Retrieval for Java Projects. In *FSE*. 643–654.
[9] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. 1994. TestTube: A System for Selective Regression Testing. In *ICSE*. 211–220.
[10] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *FSE*. 235–245.
[11] Emelie Engström and Per Runeson. 2010. A Qualitative Survey of Regression Testing Practices. In *PROFES*. 3–16.
[12] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. *IST* 52, 1 (2010), 14–30.
[13] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's Distributed and Caching Build Service. In *ICSE, SEIP*. 11–20.
[14] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *ISSTA*. 211–222.
[15] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. 2014. An Empirical Evaluation and Comparison of Manual and Automated Test Selection. In *ASE*. 361–372.
[16] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: detecting state-polluting tests to prevent test dependency. In *ISSTA*. 223–233.
[17] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The Art of Testing Less without Sacrificing Quality. In *ICSE*. 483–493.
[18] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In *FSE*. 197–207.
[19] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *FSE*. 583–594.
[20] Qemu Home Page. QEMU - the FAST! processor emulator. https://www.qemu.org.
[21] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A tool for change impact analysis of Java programs. In *OOPSLA*. 432–448.
[22] Gregg Rothermel and Mary Jean Harrold. 1993. A safe, efficient algorithm for regression test selection. In *ICSM*. 358–367.
[23] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *TSE* 22, 8 (1996), 529–551.
[24] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *TOSEM* 6, 2 (1997), 173–210.
[25] Mats Skoglund and Per Runeson. 2007. Improving Class Firewall Regression Test Selection by Removing the Class Firewall. *IJSEKE* 17, 3 (2007), 359–378.
[26] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively Prioritizing Tests in Development Environment. In *ISSTA*. 97–106.
[27] TizenRT Home Page. TizenRT - Lightweight RTOS-based platform for low-end IoT devices. https://github.com/Samsung/TizenRT.
[28] TizenRT Pull1368. Should deinitialize database after removing relations. https://github.com/Samsung/TizenRT/pull/1368.
[29] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and Back Again: Can you Compile that Snapshot? *JSEP* (2017).
[30] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. 2017. File-Level vs. Module-Level Regression Test Selection for .NET. In *FSE, industry track*. 848–853.
[31] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *STVR* 22, 2 (2012), 67–120.
[32] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *ICSM*. 23–32.