When Automated Program Repair Meets Regression Testing—An Extensive Study on Two Million Patches

YILING LOU, Fudan University, Shanghai, China
JUN YANG, University of Illinois Urbana-Champaign, Champaign, IL, USA
SAMUEL BENTON, The University of Texas at Dallas, Richardson, TX, USA
DAN HAO, Peking University, Beijing, China
LIN TAN, Purdue University, West Lafayette, IN, USA
ZHENPENG CHEN, Nanyang Technological University, Singapore, Singapore
LU ZHANG, Peking University, Beijing, China
LINGMING ZHANG, University of Illinois Urbana-Champaign, West Lafayette, IN, USA

In recent years, Automated Program Repair (APR) has been extensively studied in academia and even drawn wide attention from the industry. However, APR techniques can be extremely time consuming since (1) a large number of patches can be generated for a given bug, and (2) each patch needs to be executed on the original tests to ensure its correctness. In the literature, various techniques (e.g., based on learning, mining, and constraint solving) have been proposed/studied to reduce the number of patches. Intuitively, every patch can be treated as a software revision during regression testing; thus, traditional Regression Test Selection (RTS) techniques can be leveraged to only execute the tests affected by each patch (as the other tests would keep the same outcomes) to further reduce patch execution time. However, few APR systems actually adopt RTS and there is still a lack of systematic studies demonstrating the benefits of RTS and the impact of different RTS strategies on APR. To this end, this article presents the first extensive study of widely used RTS techniques at different levels (i.e., class/method/statement levels) for 12 state-of-the-art APR systems on over 2M patches. Our study reveals various practical guidelines for bridging the gap between APR and regression testing, including: (1) the number of patches widely used for measuring APR efficiency can incur skewed conclusions, and the use of inconsistent RTS configurations can further skew the conclusions; (2) all studied RTS techniques can substantially improve APR efficiency and should be considered in future APR work; (3) method- and statement-level RTS outperform class-level RTS substantially and should be preferred; (4) RTS techniques can substantially outperform state-of-the-art test prioritization techniques for APR, and combining them can further improve APR efficiency; and (5) traditional Regression Test Prioritization (RTP) widely studied in regression testing performs even better than APR-specific test prioritization when combined with most RTS techniques. Furthermore, we also present the detailed impact of different patch categories and patch validation strategies on our findings.

Authors' Contact Information: Yiling Lou, Department of Computer Science, Fudan University, Shanghai, China; e-mail: yilinglou@fudan.edu.cn; Jun Yang, Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, IL, USA; e-mail: jy70@illinois.edu; Samuel Benton, Department of Computer Science, The University of Texas at Dallas, Richardson, TX, USA; e-mail: samuel.benton1@utdallas.edu; Dan Hao, School of Computer Science, Peking University, Beijing, China; e-mail: haodan@pku.edu.cn; Lin Tan, Department of Computer Science, Purdue University, West Lafayette, IN, USA; e-mail: lintan@purdue.edu; Zhenpeng Chen (Corresponding author), School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore; e-mail: zhenpeng.chen@ntu.edu.sg; Lu Zhang, School of Computer Science, Peking University, Beijing, China; e-mail: zhanglucs@pku.edu.cn; Lingming Zhang, Department of Computer Science, University of Illinois Urbana-Champaign, IL, USA; e-mail: lingming@illinois.edu

ACM 1557-7392/2024/9-ART180
https://doi.org/10.1145/3672450

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**;

Additional Key Words and Phrases: Test selection, program repair, patch validation

ACM Reference format:
Yiling Lou, Jun Yang, Samuel Benton, Dan Hao, Lin Tan, Zhenpeng Chen, Lu Zhang, and Lingming Zhang. 2024. When Automated Program Repair Meets Regression Testing—An Extensive Study on Two Million Patches. *ACM Trans. Softw. Eng. Methodol.* 33, 7, Article 180 (September 2024), 23 pages.
https://doi.org/10.1145/3672450

1 Introduction

Automated Program Repair (APR) [25, 29, 32, 71, 79, 84] has been proposed to automatically generate patches for buggy programs so as to reduce manual debugging efforts. Modern APR techniques often follow a *generate-and-validate* procedure, leveraging test suites as the partial specification of the desired program behavior. More specifically, a test-based APR system repeatedly generates patches and validates them against the whole test suite until a patch that can pass all tests is found. To date, a large number of APR techniques have been proposed, effectively fixing a considerable number of real bugs and improving software quality/productivity [17, 24].

Although receiving wide attention from both academia and industry, APR techniques can be extremely time consuming [17, 19, 22, 44, 47, 70]: (1) for a given bug, there are a vast number of patches generated; (2) for each generated patch, it often takes non-trivial time to execute the original tests for correctness validation. Efficiency is essential for the practical usage of APR systems [44], as both the production cycle and the development cycle require low-latency debugging assistance from APR systems. In particular, given the computing resources are not always sufficient, parallel execution cannot always be available for APR systems.

To reduce overheads in the repair procedure, researchers have proposed various cost reduction approaches. For example, numerous APR techniques have been proposed to reduce the number of generated patches, including constraint-based [15, 53, 79], heuristic-based [24, 72, 84], template-based [32, 40, 41], and learning-based techniques [13, 37, 51]. However, APR still remains one of the most costly approaches in software engineering.

Regression Testing [83] reruns regression tests on every software revision to check whether it breaks previously working functionalities and have been widely adopted in practice. Meanwhile, rerunning all tests for each revision can be extremely time consuming [16, 56]. Therefore, researchers have proposed various regression testing techniques to speed up the process. For example, given a software revision, **Regression Test Selection (RTS)** [18, 36, 58, 61, 62] only selects/executes the tests affected by code changes for faster regression testing (because the other tests should have the same outcomes on the original and new revisions). To date, RTS techniques have been shown to significantly accelerate regression testing and has been widely incorporated into build systems of open source projects [18, 86] and commercial systems [10, 87].

In fact, *APR shares a similar procedure with regression testing that the buggy program is repeatedly modified (by APR systems rather than developers during regression testing) and each generated patch can be treated as a software revision to be exhaustively validated during regression testing*. Therefore, regression testing techniques, such as RTS, can be naturally applied to accelerate APR. However, surprisingly, after systematically revisiting the APR literature, we find that such an important optimization receives little attention from existing APR work: *most existing APR systems do not apply any RTS, while the few APR systems doing so adopt RTS at different granularities without demonstrating their clear benefits*. For example, **Practical Program Repair (PraPR)** [17] adopts statement-level RTS while CapGen [71] uses class-level RTS in its implementation without any justification. Therefore, *it remains unknown that how important it is to adopt RTS in APR systems and how different RTS techniques affect APR efficiency.* As a result, 16 of the 17 APR systems proposed in recent 3 years have still not adopted any RTS at all (Section 2.3).

To bridge such a knowledge gap, we perform the first extensive study to investigate the impact of widely used RTS techniques on APR efficiency. More specifically, we first investigate the cost reduction achieved by widely used RTS techniques at different levels (i.e., class/method/statement levels); furthermore, we also study the joint impact of RTS and other popular regression testing techniques (i.e., various test prioritization techniques which reorder test executions and can also potentially speed up patch validation) on APR efficiency. Our study is conducted on 12 state-of-the-art APR systems with over 2M patches generated for the widely used Defects4J [28] benchmark.

Our study reveals various practical guidelines for the APR community. (1) The number of patches widely used for measuring APR efficiency can incur skewed conclusions, and the use of inconsistent RTS configurations can further skew the conclusion. (2) RTS can indeed reduce a significant portion of test executions on all studied APR systems and should be considered in future APR work. (3) The performance varies among different RTS strategies. In particular, statement- and method-level RTS can significantly outperform class-level RTS and are recommended for future APR work. (4) RTS techniques can substantially outperform state-of-the-art test prioritization techniques for APR, and combining them can further improve APR efficiency. (5) Traditional **Regression Test Prioritization (RTP)** techniques [39, 48] widely studied in regression testing perform even better than APR-specific test prioritization [59] when they are combined with most RTS techniques. Besides, we further present the detailed impact of different patch categories and patch validation strategies on our findings. Lastly, we also discuss the impact of RTS strategies on the repair effectiveness.

As the first extensive study on the impact of regression testing on APR, this article makes the following contributions:

- *Literature review*. Revisiting the literature and implementations of existing APR systems, highlighting RTS as an important optimization mostly neglected and inconsistently configured by existing work.
- *Extensive study*. Performing the first extensive study on the impact of different regression testing strategies on APR efficiency, exploring various representative RTS techniques (and RTP techniques) for 12 state-of-the-art APR systems on a well-established benchmark (Defects4J) with 395 real bugs, involving over 2M patches in total.
- *Practical guidelines*. Demonstrating the importance of regression testing for APR and revealing various practical guidelines regarding the adoption of regression testing for the APR community and future APR work.

2 Background and Motivation

2.1 RTS

RTS [18, 36, 58, 61, 62] accelerates regression testing by executing only a subset of regression tests. Its basic intuition is that the tests not affected by code changes would have the same results on the original and modified revisions. An RTS technique is regarded as *safe* if it selects all tests that may be affected by changed code, because missing any of such tests may fail to detect some regression bugs. A typical RTS strategy involves analyzing two dimensions of information: (1) *changed code elements* between the original and modified program revisions and (2) *test dependencies* on the original revision. RTS then selects the tests whose dependencies overlap with the changed code elements.

Based on the granularities of changed code elements and test dependencies, RTS techniques can be categorized as *class-level* [18, 36], *method-level* [60], and *statement-level* RTS [21, 58, 62]. For example, Gligoric et al. [18] proposed an efficient class-level RTS, and Zhang [86] proposed an RTS strategy of hybrid granularities. In addition, according to how test dependencies are analyzed, RTS techniques can be categorized as *dynamic* [18, 21, 58] and *static* RTS [33, 36]. For example, Legunsen et al. [36] compared the performance and safety of static RTS approaches in modern software evolution and found that class-level static RTS was comparable to dynamic class-level RTS. To date, RTS has been shown to substantially reduce the end-to-end regression testing costs [18, 86] and has been widely incorporated into build systems of open source projects [2, 4]. While existing work has extensively studied different RTS approaches in the traditional regression testing and other related scenarios (e.g., non-functional genetic improvement [20]), in this work, we perform the first extensive study of RTS in the APR scenario.

2.2 APR

APR [53, 67, 82] automatically fixes program bugs with minimal human intervention. Given a buggy program, APR generates patches and then validates them to check their correctness. A typical test-based APR system takes a buggy program and its test suite (with at least one failed test) as inputs and consists of three phases. (1) *Fault localization*: before the repair process, off-the-shelf fault localization [7] is leveraged to diagnose suspicious code elements (e.g., statements). (2) *Patch generation*: the APR system then applies repair operations on suspicious locations following the suspiciousness ranking list. Each modified program version is denoted as a *candidate patch*. (3) *Patch validation*: lastly, each candidate patch is validated by the test suite until a patch that can pass all tests is found, i.e., *plausible patch*. The whole patch validation is often terminated once finding plausible patches or reaching the time budget.

Existing work suggests that patch validation is very time consuming [22, 49], because (1) the number of generated patches is large and (2) each patch validation requires non-trivial time to execute the original tests. To date, various APR systems have been proposed to generate patches with different strategies, and they can be categorized into the following categories according to the way of patch generation. (1) Heuristic-based APR [24] (e.g., GenProg [84], ARJA-e [85], VarFix [75]) iteratively explore the search space of program modifications via certain heuristics; (2) constraint-based APR [15, 53, 79], e.g., Nopol [80], generates patches for conditional expressions by transforming them into constraint solving tasks; (3) template-based APR [32, 41], e.g., KPar [40], designs predefined patterns to guide patch generation; (4) learning-based APR [13, 37, 51], e.g., CocoNut [51], adopts machine/deep learning techniques to generate patches based on existing code corpus.

2.3 RTS in Patch Validation

In addition to the large number of generated patches, each patch also needs to be executed against the original tests, which can be very costly. In fact, the *generate-and-validate* procedure in APR is very similar to the regression testing scenario that patches are modifications of the original buggy program and each of them needs to be validated by the existing test suite. Therefore, it is intuitive to adopt regression testing techniques in patch validation, such as improving the repair efficiency by executing only the tests affected by the patch via RTS. We then systematically revisit RTS techniques adopted in existing APR systems. In particular, we consider APR systems targeting Java programs due to its popularity in the APR community and use two sources of information to collect representative APR systems (i.e., program-repair.org and the living review of APR [57]). For each APR system, we check which RTS strategy is used both in its paper and its implementation (if the source code is released). Table 1 summarizes the results of our literature review.

Table 1. Revisiting RTS Strategies on APR Systems

APR	Time	RTS	APR	Time	RTS	APR	Time	RTS
PAR [31]	2013	No	jGenProg [53]	2016	No	jKali [53]	2016	No
jMutRepair [53]	2016	No	DynaMoth [16]	2016	No	xPAR [36]	2016	No
HDRepair [36]	2016	No	NPEFix [15]	2017	No	ACS [80]	2017	No
Genesis [47]	2017	No	jFix/S3 [35]	2017	No	EXLIR [65]	2017	No
JAID [12]	2017	No	ssFix [79]	2017	No	SimFix [25]	2018	No
Cardumen [54]	2018	No	SketchFix [23]	2018	Stmt	LSRepair [44]	2018	No
SOFix [46]	2018	No	CapGen [72]	2018	Class	ARJA [85]	2018	Stmt
GenProg-A [85]	2018	Stmt	Kali-A [85]	2018	Stmt	RSRepair-A [85]	2018	Stmt
SequenceR [14]	2019	No	kPAR [41]	2019	No	DeepRepair [74]	2019	No
PraPR [18]	2019	Stmt	Hercules [66]	2019	No	GenPat [24]	2019	No
AVATAR [42]	2019	No	TBar [43]	2019	No	Nopol [81]	2019	No
ConFix [32]	2019	No	FixMiner [33]	2020	No	CocoNut [52]	2020	No
DLFix [38]	2020	No	CURE [27]	2021	No	Recoder [89]	2021	No
Reward [82]	2022	No	DEAR [39]	2022	No	AlphaRepair [78]	2022	No
ARJA-e [86]	2020	No	VarFix [76]	2021	No			

Stmt, statement.

Based on the table, we have the following findings. (1) Most existing APR systems, including the latest system proposed in 2022, have not adopted any RTS strategy to optimize the efficiency of patch validation. In other words, the whole test suite is repeatedly executed against each generated patch, including the tests that are not affected by the patch at all. Although some system (i.e., VarFix) adopts random sampling to reduce the test executions, it still executes the entire test suite after the sampled test are passing. (2) The few APR systems which adopt RTS are all based on dynamic RTS. This makes sense as prior regression testing studies show that static RTS can be imprecise and unsafe (e.g., due to Java Reflections) [36, 66]. (3) For the few APR systems using RTS, they adopt RTS at different granularities (highlighted by different colors in the table) without clear justification. For example, PraPR directly performs dynamic statement-level RTS, while CapGen uses class-level RTS.

> *Motivation:* RTS is neglected by most APR systems and different RTS strategies are randomly adopted for those few systems using RTS. Therefore, it remains unknown how necessary it is to adopt RTS in APR and how different RTS techniques affect APR efficiency. To bridge this knowledge gap, we perform the first extensive study on the impact of RTS on APR efficiency.

3 Study Design

3.1 Preliminaries

We first formally define key conceptions used in this article.

3.1.1 Patch Validation Matrix. Given a buggy program $\mathcal{P}_b$ and its test suite $\mathcal{T}$ (with at least one failed test), an APR system generates a list of candidate patches $\mathbb{P}$ after the patch generation phase.

Definition 3.1. Patch Validation Matrix $\mathbb{M}$ defines the execution results of all tests on all patches. In particular, each cell $\mathbb{M}[\mathcal{P}, t]$ represents the execution result of test $t \in \mathcal{T}$ on patch $\mathcal{P} \in \mathbb{P}$, which can have the following values: (1) ❍, if t has not been selected by RTS and thus skipped for execution, (2) ✗, if t fails on the patch $\mathcal{P}$, (3) ✓, if t passes on the patch $\mathcal{P}$, and (4) —, if t is selected by RTS but has not been executed (i.e., its execution result remains unknown).

The early-exit mechanism is a common efficiency optimization widely enabled in existing APR systems [17, 24], which stops validation for the current patch once it fails on any test. In this scenario, there are many tests whose results remain unknown, resulting in a validation matrix with “—” cells. We denote such a matrix as a *partial patch validation matrix* $\mathbb{M}_p$. Meanwhile, in the other scenario where the execution results of all tests are required [49], the early-exit mechanism is often disabled. In this scenario, all the selected tests would be executed even when there are some tests that already failed during patch validation, leaving no “—” cell in the matrix $\mathbb{M}$. We denote such a matrix as a *full patch validation matrix* $\mathbb{M}_f$. The following are examples for full and partial matrices when there is no RTS adopted in patch validation. For example, patch validation for $\mathcal{P}_1$ stops by the failure of t_1, and thus the execution results of t_2 and t_3 on $\mathcal{P}_1$ remain unknown in $\mathbb{M}_p$.

$$\mathbb{M}_f = \left[\begin{array}{cccc} & t_1 & t_2 & t_3 \\ \mathcal{P}_1 & \text{✗} & \checkmark & \checkmark \\ \mathcal{P}_2 & \text{✗} & \text{✗} & \checkmark \\ \mathcal{P}_3 & \checkmark & \checkmark & \checkmark \end{array}\right] \quad \mathbb{M}_p = \left[\begin{array}{cccc} & t_1 & t_2 & t_3 \\ \mathcal{P}_1 & \text{✗} & - & - \\ \mathcal{P}_2 & \text{✗} & - & - \\ \mathcal{P}_3 & \checkmark & \checkmark & \checkmark \end{array}\right] \tag{1}$$

3.1.2 Studied RTS Strategies. Based on the literature review, we focus on dynamic RTS strategies that utilize test coverage in buggy programs as test dependencies, since static RTS can be imprecise and unsafe [36, 66]. Given a candidate patch $\mathcal{P}$, $\mathcal{P}_\Delta$ denotes the set of code elements modified by $\mathcal{P}$, and $\mathbb{C}[\mathcal{P}_b, t]$ denotes the set of code elements in $\mathcal{P}_b$ that are covered by the test t.

Definition 3.2. Given a patch $\mathcal{P}$ and the whole test suite $\mathcal{T}$, a *dynamic RTS strategy for APR* selects a subset of tests $\mathcal{T}'$ for execution. In particular, for precise and safe RTS, each test in $\mathcal{T}'$ should cover at least one modified code element, i.e., $\forall t \in \mathcal{T}', \mathcal{P}_\Delta \cap \mathbb{C}[\mathcal{P}_b, t] \neq \emptyset$, while each test not in $\mathcal{T}'$ should not cover any modified element, i.e., $\forall t \notin \mathcal{T}', \mathcal{P}_\Delta \cap \mathbb{C}[\mathcal{P}_b, t] = \emptyset$.

Note that the above definition is simplified for the ease of understanding. Our actual implementation for ensuring RTS precision and safety is actually more complicated and handles all types of Java code changes (e.g., method-overriding hierarchy changes) following prior safe RTS work [18, 86]. According to the granularity of modified code elements, we study three RTS strategies: class-level (RTS_{class}), method-level (RTS_{method}), and statement-level RTS (RTS_{stmt}). In addition, we regard no RTS adoption as the baseline test selection strategy, denoted as RTS_{no}. In the example, if t_1 and t_3 cover the statements modified by $\mathcal{P}_1$, while t_1 and t_2 cover the statements modified by $\mathcal{P}_2$ and $\mathcal{P}_3$, the partial and full validation matrices associated with RTS_{stmt} can be represented as Equation (2). For example, since t_3 does not cover any statement modified by $\mathcal{P}_3$, RTS_{stmt} skips t_3 in the validation for $\mathcal{P}_3$. However, if t_3 happens to cover other statements in the same class as the modified statements (i.e., t_3 covers the classes modified by $\mathcal{P}_3$), RTS_{class} would select t_3 when validating $\mathcal{P}_3$. In this way, RTS at coarser granularities tend to select more tests.

$$\mathbb{M}_f = \left[\begin{array}{cccc} & t_1 & t_2 & t_3 \\ \mathcal{P}_1 & \text{✗} & \text{❍} & \checkmark \\ \mathcal{P}_2 & \text{✗} & \text{✗} & \text{❍} \\ \mathcal{P}_3 & \checkmark & \checkmark & \text{❍} \end{array}\right] \quad \mathbb{M}_p = \left[\begin{array}{cccc} & t_1 & t_2 & t_3 \\ \mathcal{P}_1 & \text{✗} & \text{❍} & - \\ \mathcal{P}_2 & \text{✗} & - & \text{❍} \\ \mathcal{P}_3 & \checkmark & \checkmark & \text{❍} \end{array}\right] \tag{2}$$

3.1.3 Efficiency Measurement. Recent work [44] on APR efficiency adopts the number of patches as the efficiency metric. However, in our work, simply counting the number of patches could be imprecise, because it treats each patch as equally costly and measures repair efficiency in an oversimplified way. For example, given $\mathbb{M}_p$ in Equation (1), both $\mathcal{P}_2$ and $\mathcal{P}_3$ would be regarded as one program execution, even if $\mathcal{P}_3$ actually executes more tests than $\mathcal{P}_2$. Therefore, in this study, we define the following metrics based on the number or time of test executions for precise efficiency measurement.

Definition 3.3. Given a patch validation matrix $\mathbb{M}$, its *accumulated number of test executions* $NT_{num}(\mathbb{M})$, sums up the number of test executions on all the validated patches, i.e.,

$NT_{num}(\mathbb{M}) = \sum \mathbb{M}[\mathcal{P}, t]$, if $\mathbb{M}[\mathcal{P}, t] \neq \bigcirc \wedge \mathbb{M}[\mathcal{P}, t] \neq -$; its *accumulated time of test executions* $NT_{time}(\mathbb{M})$, sums up the time of test executions on all the validated patches, i.e., $NT_{time}(\mathbb{M}) = \sum f_t(\mathbb{M}[\mathcal{P}, t])$, if $\mathbb{M}[\mathcal{P}, t] \neq \bigcirc \wedge \mathbb{M}[\mathcal{P}, t] \neq -$, and the function f_t returns the execution time of the given test t on the patch $\mathcal{P}$.

Definition 3.4. Given the patch validation matrix $\mathbb{M}$ without regression testing technique, and the matrix $\mathbb{M}^{RTS}$ generated by certain regression testing strategy, $Reduction_{num/time}$ measures the efficiency improvement achieved by the RTS strategy compared to no RTS, i.e., $Reduction_{num/time} = \frac{NT_{num/time}(\mathbb{M}) - NT_{num/time}(\mathbb{M}^{RTS})}{NT_{num/time}(\mathbb{M})}$.

For example, without any RTS (in Equation (1)), $NT_{num}(\mathbb{M}_f)$ is 9 and $NT_{num}(\mathbb{M}_p)$ is 5. Meanwhile, with RTS_{stmt} (in Equation (2)), $NT_{num}(\mathbb{M}_f^{RTS})$ is 6 and $NT_{num}(\mathbb{M}_p^{RTS})$ is 4. RTS_{stmt} achieves 33.33% /20.00% $Reduction_{num}$ in the full/partial matrix. Such efficiency difference cannot be captured by the number of patches. For example, if only considering the number of patches in $\mathbb{M}_p^{RTS}$ and $\mathbb{M}_p$, there is no reduction achieved by RTS at all.

In this work, we mainly present the results on the number of test executions. In fact, we find that results between the number or the time of test executions are consistent (more details in Section 5). While execution time is often dependent on many factors (e.g., specific implementations and test execution engines) unrelated to APR approaches [44], the number of test executions is more stable and more suitable for future work to reproduce. In addition, we also discuss the RTS impact on repair effectiveness in Section 5.

3.2 Research Questions (RQs)

- *RQ1: Revisiting APR efficiency.* We revisit the efficiency of APR systems by making the first attempt to (1) measure repair efficiency by the number of test executions and (2) compare the efficiency of different APR systems with consistent RTS configurations to eliminate the bias from RTS strategies.
- *RQ2: Overall impact of RTS strategies.* We investigate the efficiency improvement achieved by different RTS strategies based on their reduction of test executions.
- *RQ3: Impact on different patches.* We further study the impact of RTS on patches of different characteristics (i.e., fixing capabilities and fixing scopes) to find out what kinds of patches are more susceptible to RTS.
- *RQ4: Impact with the full patch validation matrix.* In RQ1–RQ3, we investigate APR efficiency with the default *partial* patch validation matrix (with early-exit), which is widely adopted in many APR systems for the sake of efficiency. In this RQ, we further study the impact of RTS strategies with the *full* validation matrix (without early-exit).
- *RQ5: Combining test selection with prioritization.* In addition to RTS strategies, when the early-exit is enabled, the execution order of tests jointly decides the number of test executions. Therefore, we further combine the studied RTS strategies with state-of-the-art test prioritization techniques and investigate their impact on APR efficiency.

3.3 Benchmark

We perform our study on the benchmark Defects4J V.1.2.0 [28], which is the version used most widely by existing APR work [17, 42, 79]. Defects4J V.1.2.0 includes 395 real-world bugs from 6 real-world software systems. We choose Defecst4J, as (1) some studied APR systems are only implemented for Java (e.g., SimFix), and (2) most studied systems are evaluated on Defects4J in their previous work while using a consistent widely used benchmark can better position our findings in the domain.

3.4 Studied APR Systems

Following the recent studies on APR [44], our study selects APR systems for Java program according to the following criteria. (1) The source code of the APR system is publicly available because we need to modify patch validation settings. (2) The APR systems can be successfully applied to Defects4J subjects. (3) The APR systems require no extra input data besides program source code and the available test suite. In this way, we successfully collect 16 APR systems fulfilling the requirements above, and all of them are also used as the studied APR systems in previous work [44]. However, these APR systems use test cases at different granularities: the Astor family [52] (including jMutRepair, jGenProg, jKali) and Cardumen [53] regard each test class as a test case, while the remaining systems all regard each test method as a test case. Since RTS results at different test-case granularities are incomparable, we use the more popular test-method granularity with 12 APR systems in our study, including (1) *constraint-based systems*: **Accurate Condition Synthesis (ACS)** and Dynamoth, (2) *heuristic-based systems*: Arja, GenProg-A, Kali-A, RSRepair-A, and SimFix, (3) *template-based systems*: AVATAR, FixMiner, kPar, PraPR, and TBar. In total, *the studied APR systems generate 2,532,915 patches for all the Defects4J subjects studied, while each patch can involve up to thousands of test executions.*

3.5 Implementation Details

To implement studied RTS strategies, following state-of-the-art Ekstazi [18] and HyRTS [86], we leverage on-the-fly bytecode instrumentation with ASM [5] and Java Agent [6] for lightweight test dependency collection at the class/method/statement level on buggy programs. For all studied APR systems, we modify their source code to collect patch/test execution results with different RTS and validation settings. We only keep the compilable patches (85% compilation rate) in our experiments as RTS strategies cannot be applied to un-executable patches. For each system, we try up to three re-executions for failed/error repair attempts. In all RQs, we schedule all originally failed tests prior to originally passed tests, which is also the common practice of all recent APR systems because the former are more likely to fail again on patches. For each APR system, we mainly follow the same configurations (such as timeout and JDK version) as its original publication. We manually modified all studied APR systems to collect the detailed patch execution information required by unified debugging and ensured our modified versions did not impact underlying tool functionality. Each system used original time settings suggested by the original papers. Each tool was executed using the same JDK version found in the tool's original publication, allowing us to obtain repair execution results as close as possible to the tool's original results.

From RQ1 to RQ4, within the originally failed tests or the originally passed tests, we prioritize them by their default lexicographical order to eliminate the impact of different test execution orders. Our data are available at [1], including patch results and impact on repair effectiveness of all studied APR tools.

3.6 Threat to Validity

Our study focuses on APR systems for Java programs, which might threaten generality of our findings. We mitigate this threat by performing our study on a large spectrum of APR systems belonging to different categories and a widely used benchmark with hundreds of real bugs [17, 24, 71]. In addition, to mitigate the threat in faulty implementations, we carefully review our RTS implementations and experimental scripts; the source code of all the studied APR systems was directly obtained from their authors or original open source repositories; we further ensured that each APR system performs consistently before and after our modifications.

Table 2. Number of Test Executions among APR Systems

Subject	RTS Strategy	PraPR	SimFix	AVATAR	kPar	TBar	FixMiner	Dynamoth	ACS	Arja	Kali-A	GenProg-A	RSRepair-A
Lang	#Patch	381.27	275.98	4.63	4.79	4.76	4.56	0.16	1.00	522.97	26.92	628.85	289.23
	#Test (RTS_{no})	2,117.61	8,802.36	321.41	163.19	437.52	233.15	1,256.86	1,140.80	68,448.64	799.43	86,910.25	10,945.44
	#Test (RTS_{class})	447.31	291.91	19.48	18.30	24.63	18.88	25.71	64.40	1,571.25	59.54	2,746.19	620.69
	#Test (RTS_{method})	414.15	244.86	12.11	11.89	12.63	11.88	2.86	1.80	1,196.83	51.54	1,376.81	556.39
	#Test (RTS_{stmt})	412.59	244.79	11.44	11.59	11.96	11.50	2.00	1.20	1,176.53	50.63	1,369.64	534.58
Math	#Patch	1,483.12	469.45	5.84	5.12	5.01	4.76	0.23	1.00	804.09	10.11	858.19	411.58
	#Test (RTS_{no})	11,879.77	13,723.01	168.78	237.05	640.65	239.38	2,042.74	3,071.33	104,804.90	705.64	73,174.10	9,661.55
	#Test (RTS_{class})	1,711.16	679.05	12.92	46.66	34.66	9.47	50.79	241.33	4,828.21	22.00	5,231.08	828.93
	#Test (RTS_{method})	1,573.21	403.17	8.20	7.25	8.18	7.03	32.53	14.83	2,105.64	15.97	1,610.38	498.28
	#Test (RTS_{stmt})	1,557.41	379.67	7.81	6.86	7.25	6.23	27.00	6.33	1,382.11	14.10	1,419.85	463.63
Time	#Patch	1,466.54	396.50	2.50	2.00	2.23	1.08	0.23	1.00	335.92	13.23	380.62	107.46
	#Test (RTS_{no})	7,782.19	3,754.42	1,065.00	4.73	359.09	490.00	1,303.67	3,894.00	34,310.82	724.00	141,456.00	2,652.50
	#Test (RTS_{class})	2,513.88	658.67	232.55	4.73	80.82	107.38	537.33	2,042.00	7,337.91	325.70	29,401.40	635.00
	#Test (RTS_{method})	1,848.62	372.08	9.73	4.73	6.55	5.25	8.67	52.00	3,822.82	55.30	10,596.50	413.90
	#Test (RTS_{stmt})	1,808.12	368.08	9.73	4.73	6.55	5.25	8.67	27.00	1,781.91	38.50	1,903.60	291.50
Chart	#Patch	784.88	588.21	5.00	4.56	4.40	4.48	0.40	-	627.76	42.36	729.72	333.16
	#Test (RTS_{no})	5,517.96	2,459.42	272.84	107.61	367.74	102.12	1,640.80	-	202,236.87	2,035.27	104,090.08	9,552.46
	#Test (RTS_{class})	1,053.80	529.25	8.26	8.00	8.79	7.12	77.30	-	27,627.48	133.45	5,694.38	758.38
	#Test (RTS_{method})	868.24	485.96	6.89	6.89	6.37	6.59	16.70	-	6,853.48	61.45	2,645.12	450.67
	#Test (RTS_{stmt})	863.96	485.92	6.89	6.89	6.37	6.59	16.30	-	2,356.57	56.82	1,746.42	386.71
Closure	#Patch	14,725.00	511.03	2.96	2.61	2.94	0.42	-	-	-	-	-	-
	#Test (RTS_{no})	86,651.19	7,995.31	342.78	378.73	478.09	2.89	-	-	-	-	-	-
	#Test (RTS_{class})	47,374.48	3,229.28	140.81	154.94	197.02	2.89	-	-	-	-	-	-
	#Test (RTS_{method})	28,476.71	922.37	6.17	9.90	6.67	2.89	-	-	-	-	-	-
	#Test (RTS_{stmt})	21,825.07	662.63	4.49	3.92	4.40	2.89	-	-	-	-	-	-
Mockito	#Patch	2,307.92	-	1.67	2.00	1.81	1.67	-	-	-	-	-	-
	#Test (RTS_{no})	3,753.19	-	267.75	41.27	200.27	4.62	-	-	-	-	-	-
	#Test (RTS_{class})	2,857.81	-	97.56	26.00	146.47	4.62	-	-	-	-	-	-
	#Test (RTS_{method})	2,555.61	-	7.62	5.80	48.00	4.62	-	-	-	-	-	-
	#Test (RTS_{stmt})	2,547.64	-	7.38	5.80	7.33	4.62	-	-	-	-	-	-

4 Results Analysis

4.1 RQ1: Revisiting APR Efficiency

Table 2 presents the number of test executions with different RTS strategies. We also present the number of validated patches in the table (i.e., #Patch is the average patch number of each bug). Note that cells are empty when there is no valid patch generated. Based on the table, we have the following observations.

First, measuring APR efficiency by a more precise metric (i.e., the number of test executions) can draw totally different conclusions from the number of validated patches advocated by prior work [44]. For example, for each buggy version of Math, SimFix (469.45 patches) validates many more patches than RSRepair-A (411.58 patches), showing that SimFix is less efficient than RSRepair-A in terms of the number of validated patches. Such a conclusion is consistent with the finding in previous work [44], which also measures APR efficiency by the number of validated patches. However, the conclusion can be opposite if these systems are compared by the number of test executions. For example, with class-level RTS, SimFix executes 679.05 tests and RSRepair-A executes 828.93 tests per buggy version of Math, indicating that SimFix is more efficient than RSRepair-A. The reason is that although RSRepair-A generates fewer patches than SimFix, each patch of RSRepair-A executes many more tests than SimFix. Such inconsistencies between the number of test executions and patches are prevalent. Hence, it is imprecise to measure APR efficiency by simply counting the number of patches, since different patches can execute totally different number of tests. Future work on APR efficiency should also consider the number of test executions.

Second, comparing APR efficiency with different RTS strategies can also deliver opposite conclusions. For example, on subject Time, RSRepair-A executes 413.90 and 291.50 tests with RTS_{method} and RTS_{stmt}, while SimFix executes 372.08 and 369.08 tests with RTS_{method} and RTS_{stmt}. When both systems are configured with method-level RTS, RSRepair-A is considered as less efficient than SimFix; however, when they are compared under statement-level RTS, RSRepair-A is actually more efficient than SimFix. Moreover, if APR systems are compared under different RTS strategies

Table 3. Reduction of the Number of Test Executions by Different RTS Strategies

Subject	RTS Strategy	PraPR	SimFix	AVATAR	kPar	TBar	FixMiner	Dynamoth	ACS	Arja	Kali-A	GenProg-A	RSRepair-A
Lang	RTS_{class}	33.33%	35.66%	17.92%	10.57%	21.48%	14.80%	69.99%	56.37%	36.67%	22.16%	26.11%	30.12%
	RTS_{method}	34.51%	36.31%	18.35%	10.95%	22.05%	15.22%	71.31%	59.94%	38.16%	22.50%	26.68%	31.17%
	RTS_{stmt}	34.55%	36.31%	18.40%	10.99%	22.09%	15.26%	71.36%	59.97%	38.17%	22.52%	26.70%	31.20%
Math	RTS_{class}	48.91%	52.45%	5.46%	5.36%	19.91%	8.63%	92.17%	90.59%	38.66%	16.44%	28.77%	31.73%
	RTS_{method}	50.41%	55.81%	6.27%	6.30%	21.08%	8.99%	93.06%	98.86%	40.16%	16.79%	30.16%	32.44%
	RTS_{stmt}	50.50%	55.89%	6.28%	6.31%	21.11%	9.05%	93.33%	99.63%	40.42%	16.85%	30.35%	32.68%
Time	RTS_{class}	31.25%	29.42%	7.14%	0.00%	7.15%	9.83%	19.62%	47.56%	19.59%	17.90%	7.97%	16.26%
	RTS_{method}	40.12%	35.60%	9.06%	0.00%	9.06%	12.45%	33.15%	98.66%	21.41%	28.27%	9.31%	17.38%
	RTS_{stmt}	40.31%	35.60%	9.06%	0.00%	9.06%	12.45%	33.15%	99.31%	22.60%	29.01%	9.93%	18.01%
Chart	RTS_{class}	58.50%	33.16%	15.68%	5.49%	20.88%	5.85%	85.96%	-	53.39%	40.87%	34.93%	39.33%
	RTS_{method}	62.45%	34.67%	15.76%	5.55%	21.02%	5.88%	89.10%	-	59.10%	43.78%	38.81%	42.02%
	RTS_{stmt}	62.52%	34.67%	15.76%	5.55%	21.02%	5.88%	89.12%	-	59.67%	44.10%	39.22%	42.45%
Closure	RTS_{class}	31.73%	20.27%	3.22%	3.79%	3.90%	0.00%	-	-	-	-	-	-
	RTS_{method}	46.18%	30.51%	5.28%	6.56%	7.38%	0.00%	-	-	-	-	-	-
	RTS_{stmt}	51.80%	32.01%	5.30%	6.65%	7.42%	0.00%	-	-	-	-	-	-
Mockito	RTS_{class}	18.03%	-	21.10%	2.77%	5.49%	0.00%	-	-	-	-	-	-
	RTS_{method}	21.17%	-	30.69%	6.43%	14.82%	0.00%	-	-	-	-	-	-
	RTS_{stmt}	21.30%	-	30.72%	6.43%	19.67%	0.00%	-	-	-	-	-	-
Average	RTS_{class}	36.96%	34.19%	11.75%	4.66%	13.13%	6.52%	66.93%	64.84%	37.08%	24.34%	24.45%	29.36%
	RTS_{method}	42.47%	38.58%	14.23%	5.96%	15.90%	7.09%	71.66%	85.82%	39.71%	27.83%	26.24%	30.75%
	RTS_{stmt}	43.50%	38.90%	14.25%	5.99%	16.73%	7.11%	71.74%	86.30%	40.22%	28.12%	26.55%	31.09%

respectively, the conclusion may further be skewed. Therefore, it is important to adopt the same RTS strategy when comparing the repair efficiency among different APR systems. We also strongly recommend the future APR work to explicitly describe their RTS strategy adopted in experiments, so that their follow-up work can mitigate the threats in RTS configuration.

> *Finding 1: Using inconsistent RTS configurations can skew the conclusion of APR efficiency. Future APR work should explicitly describe their adopted RTS strategies and the efficiency comparison among multiple APR systems should guarantee a consistent RTS configuration.*

4.2 RQ2: Overall Impact of RTS Strategies

Table 3 presents the reduction of the number of test executions achieved by each RTS strategy compared to no RTS. Based on the table, for all the studied APR systems on almost all the subjects, adopting RTS can remarkably reduce the number of test executions in patch validation. For example, on the subject Math, Dynamoth skips 93.33% of test executions if it adopts RTS_{stmt}. Even for RTS at the coarsest level (i.e., RTS_{class}), the average reduction ranges from 4.66% to 66.93% among different APR systems. Note that at some cases there is no difference among RTS strategies, e.g., FixMiner on Closure, because all generated patches fail at the first executed test (i.e., the first originally failed test still fails on these patches). We would further analyze the individual impact on different patches in Section 4.3. In summary, our results suggest that RTS can significantly improve the efficiency of patch validation and future APR work should no longer overlook such an important optimization.

> *Finding 2: For all studied APR systems, adopting RTS can remarkably improve APR efficiency and should be considered in future APR work.*

Overall, there is a common trend that RTS_{stmt} always executes the least tests while RTS_{class} exhibits the most tests. This is not surprising because class-level coverage is the coarsest selection criterion, based on which more tests would inherently be selected. In addition, RTS_{method}/RTS_{stmt}

Table 4. Reduction on $\mathbb{P}_{P2F}$ and $\mathbb{P}_{\checkmark}$

APR	$\mathbb{P}_{P2F}$			$\mathbb{P}_{\checkmark}$		
	RTS_{class}	RTS_{method}	RTS_{stmt}	RTS_{class}	RTS_{method}	RTS_{stmt}
PraPR	73.24%	85.74%	87.95%	87.33%	97.18%	97.88%
SimFix	82.88%	99.11%	99.15%	84.50%	96.77%	98.22%
AVATAR	81.16%	98.95%	99.17%	82.27%	99.36%	99.53%
kPar	66.33%	98.07%	98.58%	83.47%	99.31%	99.80%
TBar	68.34%	98.63%	99.13%	76.07%	95.39%	99.55%
FixMiner	85.02%	96.90%	99.14%	93.28%	99.71%	99.80%
Dynamoth	94.75%	96.71%	97.22%	87.50%	99.19%	99.27%
ACS	-	-	-	98.34%	99.66%	99.95%
Arja	90.09%	95.56%	97.28%	93.58%	99.15%	99.44%
Kali	88.76%	97.97%	99.30%	84.30%	99.00%	99.59%
GenProg	88.85%	96.84%	99.05%	94.28%	99.33%	99.57%
RSRepair	92.83%	97.94%	99.42%	95.79%	99.11%	99.58%

can significantly reduce more test executions (4.36%/4.69% on average) than RTS_{class}. Such a difference is statistically significant according to the Wilcoxon Signed-Rank Test [74] at the significance level of 0.05 on almost all the APR systems (i.e., 11 out of 12). Meanwhile, interestingly, the difference between RTS_{method} and RTS_{stmt} is often subtle. For example, the average difference between RTS_{method} and RTS_{stmt} is only 0.33%. In fact, such observation is consistent with prior work in traditional regression testing [86], which shows that method-level RTS outperforms class-level RTS but is often not significantly worse than statement-level RTS. The reason is that modern system design principles recommend writing simple method bodies for the ease of maintenance, making the majority of method body changes directly affect all tests covering the methods (i.e., RTS_{stmt} is close to RTS_{method}). Hence, method- and statement-level RTS are more recommended for APR.

> *Finding 3: Different RTS strategies exhibit different reduction performance. RTS_{stmt} and RTS_{method} perform similarly, but both of them significantly outperform RTS_{class}.*

4.3 RQ3: Impact on Different Patches

We now investigate the impact of RTS strategies on the patches of different characteristics. Based on the intuition that a patch involving more test executions might be more sensitive to RTS strategies, we categorize patches by their fixing capabilities or fixing code scopes. Sections 4.3.1 and 4.3.2 present the impact on these patch categorizations, respectively.

4.3.1 Impact on Patches of Different Fixing Capabilities. Based on test execution results, we can categorize patches into groups of different fixing capabilities. (1) $\mathbb{P}_{F2F}$: a patch cannot fix all originally failed tests and thus its validation gets aborted by some originally failed test; (2) $\mathbb{P}_{P2F}$: a patch can fix all originally failed tests but fails on some originally passed test, and thus its validation gets aborted by some originally passed test; (3) $\mathbb{P}_{\checkmark}$: a patch that can pass all originally failed and originally passed tests (i.e., plausible patch). Obviously, $\mathbb{P}_{F2F}$ has the weakest fixing capability and its validation halts extremely early, and thus adopting RTS would not make any difference on its number of test executions. Therefore, we present the impact of RTS strategies on the other two patch groups, i.e., $\mathbb{P}_{P2F}$ and $\mathbb{P}_{\checkmark}$ in Table 4. In particular, each cell shows the reduction ratio of test executions compared to no RTS. Note that since ACS generates only $\mathbb{P}_{F2F}$ and $\mathbb{P}_{\checkmark}$, its corresponding cells in $\mathbb{P}_{P2F}$ are empty. As suggested by the table, compared to the overall reduction on all patches (i.e., the reduction in Table 3), the impact of RTS strategies on $\mathbb{P}_{\checkmark}$ and $\mathbb{P}_{P2F}$ is even more remarkable. For example, the reduction ratio ranges from 66.33% to 99.42% on $\mathbb{P}_{P2F}$ and ranges from 76.07% to

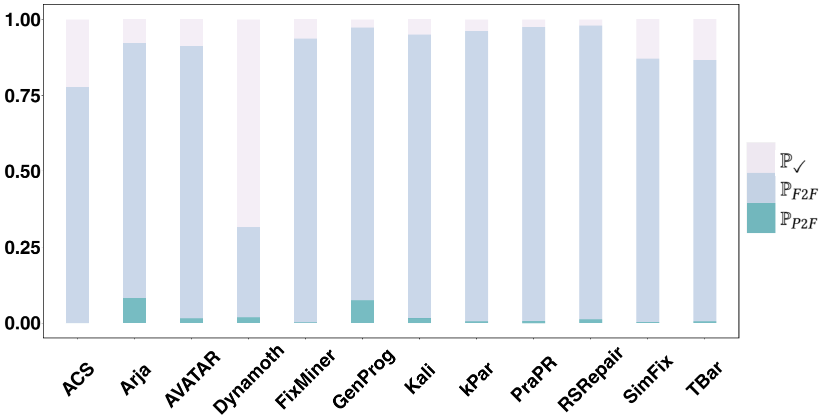

Fig. 1. Ratio of patches of different fixing capabilities.

99.95% on $\mathbb{P}_{\checkmark}$. In addition, compared to all patches, the difference between RTS strategies is also enlarged on $\mathbb{P}_{\checkmark}$ and $\mathbb{P}_{P2F}$. In summary, RTS can achieve larger reductions for patches with stronger fixing capabilities.

> *Finding 4: The impact of RTS strategies is even more prominent on $\mathbb{P}_{\checkmark}$ and $\mathbb{P}_{P2F}$, where an extremely large portion of tests (e.g., up to 99.95%) get reduced.*

Considering the drastic impact of RTS on $\mathbb{P}_{\checkmark}$ and $\mathbb{P}_{P2F}$, we further check whether an APR system is more susceptible to RTS if it exhibits a higher ratio of $\mathbb{P}_{\checkmark}$ or $\mathbb{P}_{P2F}$. Figure 1 presents the ratio of different patch categories.

First, it is noteworthy that almost all the existing APR systems exhibit a very high ratio (e.g., around 90%) of $\mathbb{P}_{F2F}$, indicating that $\mathbb{P}_{P2F}$ and $\mathbb{P}_{\checkmark}$ are sparse in the space of compilable patches. Liu et al. [44] empirically compared the number of un-compilable patches and inplausible patches generated by different APR systems, to measure their costs wasted in patch validation toward generating a valid patch. Their work showed that the state-of-the-art APR techniques can avoid generating un-compilable patches. However in our work, we make the first attempt to investigate patches of different fixing capabilities (e.g., how many tests can pass on each patch). Our results suggest that for most existing APR systems, the majority of their inplausible patches have a low capability of fixing originally failed tests and get immediately terminated in the very beginning stage of validation. We also find that the two constraint-based APR systems (e.g., Dynamoth and ACS) generate a lower ratio of $\mathbb{P}_{F2F}$. The potential reason may be that constraint-based APR systems target at conditional expressions, which are more likely to impact the test execution outcomes, i.e., making the originally failed test pass. In addition, ACS designs ranking strategies on fixing ingredients for condition synthesis, which might bring the plausible patch forward, end up the whole validation process earlier, and thus reduce the ratio of $\mathbb{P}_{F2F}$.

Second, there is no apparent correlation between the ratio of $\mathbb{P}_{P2F}$ (or $\mathbb{P}_{\checkmark}$) and the impact degree of RTS strategies (e.g., with −0.09 Pearson correlation coefficient). For example, although PraPR exhibits a quite low ratio of $\mathbb{P}_{P2F}$ and $\mathbb{P}_{\checkmark}$ (e.g., 0.29% and 0.93%), it reduces a larger portion of test executions than other APR systems. One straightforward reason is that the number of originally failed tests is significantly less than the originally passed tests. For example, each buggy version in

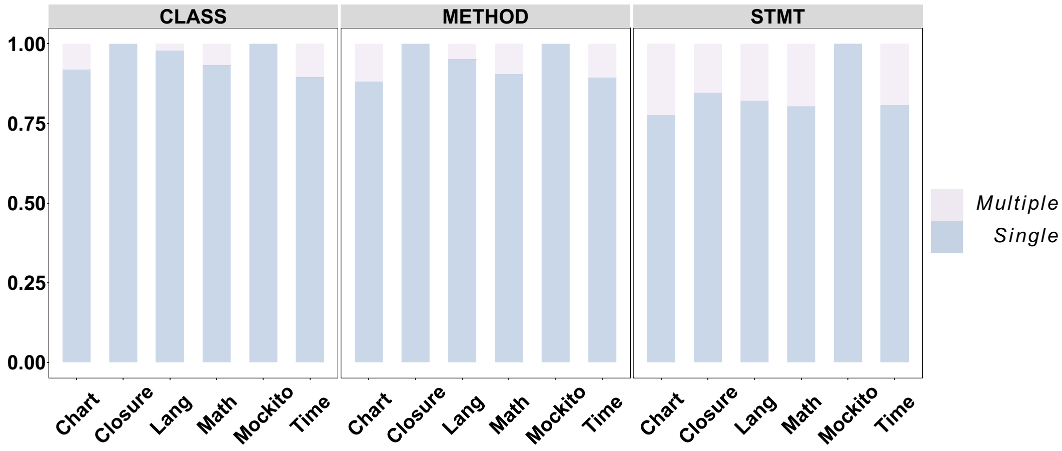

Fig. 2. Ratio of patches of different fixing code scopes.

Closure has 2.63 failed tests and 7,180.89 passed tests on average. The number of skipped originally passed tests is often significantly larger than the number of originally failed tests, and thus the impact of RTS on originally passed tests can still dominate the overall efficiency. It also explains why even with such a low ratio of $\mathbb{P}_{P2F}$ and $\mathbb{P}_{\checkmark}$ we can still observe a remarkable impact of RTS on overall efficiency.

> *Finding 5: The majority of compilable patches generated by existing APR systems immediately fail on the originally failed tests. Even with such a low $\mathbb{P}_{P2F}$ and $\mathbb{P}_{\checkmark}$, the APR efficiency is still sensitive to RTS strategies, since most bugs exhibit substantially more originally passed tests than originally failed tests.*

4.3.2 *Impact on Patches of Different Fixing Scopes.*

Based on the scope of fixed code, all the generated patches can be categorized as single-edit or multiple-edit patches. In particular, according to the granularity of edited code elements, we categorize patches as (1) patches editing single class (denoted as $\mathbb{P}_{SC}$) vs. patches editing multiple classes (denoted as $\mathbb{P}_{MC}$), (2) patches editing single method (denoted as $\mathbb{P}_{SM}$) vs. patches editing multiple methods (denoted as $\mathbb{P}_{MM}$), and (3) patches editing single statement (denoted as $\mathbb{P}_{SS}$) vs. patches editing multiple statements (denoted as $\mathbb{P}_{MS}$).

Figure 2 presents the ratio of patch categories of different fixing code scopes. It shows that the majority of generated patches are single-edit patches, especially when edited code elements are at coarser granularities (i.e., class or method).

Table 5 presents the reduction achieved by RTS at the corresponding granularity on single-edit and multiple-edit patches. For example, the cell in the column "RTS_{class}" and the row "Lang" means that on Lang RTS_{class} can reduce 21.83% and 27.36% of test executions on patches that edit multiple classes ($\mathbb{P}_{MC}$) and on patches that edit single class ($\mathbb{P}_{SC}$), respectively. Based on the table, at most cases, RTS can reduce more test executions on single-edit patches than multiple-edit patches. The reason is that multiple-edit patches often involve more code elements and thus are covered by more tests, resulting in a lower ratio of reduction. Note there are several counter cases, e.g., on Closure the reduction on multiple-statement patches is much larger than on single-statement patches

Table 5. Reduction on Single/Multiple Edit Patches

Subject	RTS_{class}		RTS_{method}		RTS_{stmt}	
	$\mathbb{P}_{MC}$	$\mathbb{P}_{SC}$	$\mathbb{P}_{MM}$	$\mathbb{P}_{SM}$	$\mathbb{P}_{MS}$	$\mathbb{P}_{SS}$
Lang	21.83%	27.36%	21.83%	27.86%	15.25%	25.75%
Math	26.18%	28.82%	17.67%	30.35%	20.45%	29.75%
Time	9.85%	13.26%	20.78%	16.49%	7.53%	14.77%
Chart	32.92%	30.99%	27.95%	35.14%	19.27%	33.64%
Closure	-	11.30%	-	19.48%	55.06%	17.34%
Mockito	-	9.48%	-	14.62%	-	15.62%

MC, multiple classes; MM, multiple methods; MS, multiple statements; SC, single class; SM, single method; SS, single statement.

(55.06% v.s. 17.34%). We find that at these cases a considerable ratio of generated multiple-edit patches have stronger fixing capabilities (i.e., belong to $\mathbb{P}_{P2F}$ or $\mathbb{P}_{\checkmark}$) and thus exhibit a higher reduction.

> *Finding 6: Most patches generated by existing APR systems are single-edit patches. RTS can reduce more test executions on single-edit patches than on multiple-edit patches, since usually fewer tests can cover the modified code elements of single-edit patches. Meanwhile, for some special cases, multiple-edit patches can have high probabilities in producing high-quality patches passing more tests, leaving more room for RTS.*

4.4 RQ4: Impact with the Full Matrix

In RQ1–RQ3, we investigate the impact of RTS with the partial validation matrix ($\mathbb{M}_p$), i.e., the early-exit mechanism is enabled and the validation for each patch would be terminated after any test failure. In RQ4, we study the impact of RTS with the full matrix ($\mathbb{M}_f$), i.e., the validation for each patch would continue even when there are failed tests. A full validation matrix is common in the scenario that execution results of all tests are required, such as unified debugging [48]. In this RQ, we further investigate whether there is any difference of RTS impacts in two scenarios (full validation matrices vs. partial validation matrices).

Table 6 shows the accumulated number of test executions $NT_{num}(\mathbb{M}_f)$ on each APR system. We present the average number of all subjects and buggy versions. It is notable that the full patch validation matrix collection is extremely resource-consuming, e.g., from thousands to millions of tests are executed when there is no RTS. We can observe a prominent reduction when RTS strategies at finer granularities are integrated in APR systems. For example, on average, *for each buggy version*, RTS_{stmt} even reduces millions of test executions for PraPR. In summary, our results show that RTS should definitely be applied in the full-matrix scenario.

> *Finding 7: The full validation matrix is extremely resource consuming, for which RTS has a more remarkable impact by reducing even up to millions of test executions.*

We further compare the reduction achieved by RTS between full and partial matrices in Figure 3. First, the reduction curves of the full matrix are far beyond the partial matrix. For example, with the

Table 6. Number of Test Executions with Full Matrices

APR	RTS_{no}	RTS_{class}	RTS_{method}	RTS_{stmt}
PraPR	20,018,923.45	12,495,932.50	9,958,460.34	9,600,813.31
SimFix	70,588.42	11,408.54	1,483.51	1,039.05
AVATAR	11,065.76	2,821.98	1,219.36	591.52
kPar	9,903.78	2,435.97	1,088.50	509.45
TBar	10,256.57	2,856.96	1,237.31	522.12
FixMiner	6,441.49	740.07	144.00	32.86
Dynamoth	596.65	114.32	17.75	16.86
ACS	93.46	5.12	1.44	0.71
Arja	1,435,025.26	177,282.34	32,882.52	23,861.27
Kali	54,751.70	3,999.24	915.59	755.44
GenProg	1,585,441.03	199,088.14	33,824.91	23,840.29
RSRepair	668,634.75	48,785.24	7,650.52	5,131.89

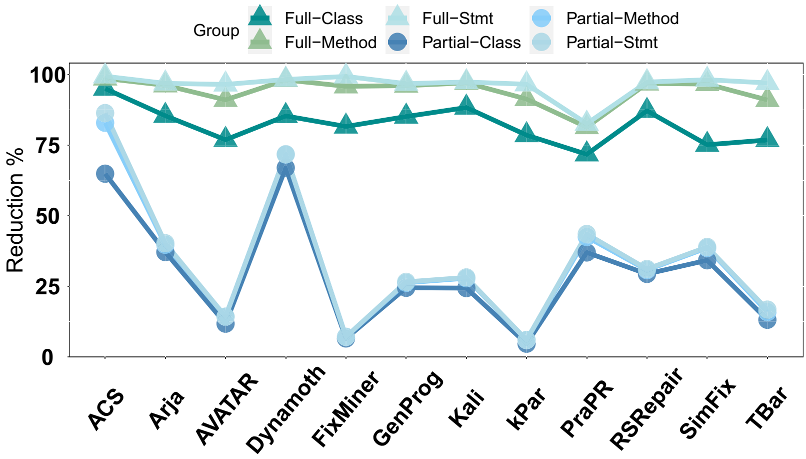

Fig. 3. Reduction with full/partial matrices.

full matrix, all the RTS strategies can help all APR systems save more than 70% of test executions. In particular, RTS_{stmt} and RTS_{method} can save more than 95% of test executions at most cases. Second, the difference between partial and full matrices varies among different APR systems. For example, Dynamoth achieves a slightly larger reduction while AVATAR achieves a much larger reduction after they switch from partial matrices into full matrices. Furthermore, consistent with the finding for partial matrices, method- and statement-level RTS perform similarly, and both substantially outperform class-level RTS. Interestingly, their superiority over class-level RTS is even enlarged with full matrices. In fact, the early-exit mechanism in partial matrices prevents more tests from executing, which mitigates the impact from RTS strategies. Therefore, the reduction achieved in the full-matrix scenario represents the upper bound of the impact.

> *Finding 8: RTS can remarkably reduce more test executions with full matrices than with partial matrices. Method- and statement-level RTS still perform similarly, but their superiority over class-level RTS is further enlarged with full matrices.*

In summary, the impact of RTS in the full matrices differs from the partial matrices in the following two folds. First, the reduction ratio in the full matrices is significantly larger than that in the partial matrices, indicating that RTS is a more necessary efficiency optimization setting for the full-matrix scenario. Second, the gap between the class-level RTS and method-level/statement-level RTS in the full-matrix scenario is substantially larger than that in the partial-matrix scenario, indicating that method-level/statement-level RTS is much more recommended for the full-matrix scenario.

4.5 RQ5: Test Selection + Prioritization

So far we have studied the RTS impact on APR efficiency controlled in a default test execution order (i.e., the lexicographic order). In fact, when the early-exit mechanism is enabled (the default setting for most modern APR systems), the test execution order can also affect the final number of test executions since it decides when the first failure-triggering test would be executed. Therefore, we further study the impact of different test prioritization strategies and their joint impact with RTS on APR efficiency:

- *Baseline prioritization*. Tests are scheduled by their lexicographic order, which is also the default setting in previous RQs (denoted as TP_{base}).
- *Patch-history-based prioritization*. Qi et al. [59] consider the tests that have failed on more validated patches as more likely to fail on the current patch and schedule them to execute earlier. We consider this approach since it is the state-of-the-art test prioritization strategy specifically designed for APR, and we are the first to study its combination with RTS (denoted as TP_{APR}).
- *RTP*. Besides RTS, RTP techniques have also been widely studied in traditional regression testing to reorder tests for early detection of regression bugs [39, 48, 68]. Since each patch can be treated as a new revision in regression testing, we can naturally apply RTP for APR. In particular, we consider two most widely studied approaches, i.e., the *total* and *additional* RTP techniques based on statement coverage [63]. They share the similar intuition that tests covering more statements or more yet-uncovered statements are more likely to detect regression bugs (denoted as RTP_{tot} and RTP_{add}). *Note that we are also the first to directly apply RTP for APR.*

For each patch, RTS is applied to decide the tests for execution and test prioritization is applied to decide their execution order. We still follow the common practice of all recent APR systems that all the originally failed tests are executed before all originally passed tests, because the former are more likely to fail again on patches. Within the originally failed tests or the originally passed tests, we further apply the test prioritization strategies to schedule their execution order.

In Table 7, we present the reductions achieved by the combination of different RTS and prioritization strategies when compared to RTS_{no} with TP_{base}. The highest reduction within each RTS strategy is highlighted. Based on the table, we have the following observations. First, combining RTS with test prioritization can further improve APR efficiency. However, compared to RTS, test prioritization brings much lower reduction ratios. For example, when PraPR adopts no RTS, the best prioritization (i.e., RTP_{add}) can achieve only 15.21% reduction, but the worst RTS alone (i.e., RTS_{class} with TP_{base}) can achieve 36.96% reduction. This indicates that test selection strategies play a more essential role in efficiency optimization. Second, the best prioritization strategy varies when combined with different RTS strategies. For most APR systems, surprisingly, the traditional RTP strategy RTP_{add} reduces the most test executions when combined with RTS_{no}, RTS_{class}, or RTS_{method}. We further perform Wilcoxon Signed-Rank Test at the significance level of 0.05 and find the differences are all statically significant (i.e., p-values < 0.05). To the best of our knowledge, this

Table 7. Reduction of Combining Selection and Prioritization

APR	RTS_{no} (%)			RTS_{class} (%)				RTS_{method} (%)				RTS_{stmt} (%)			
	TP_{APR}	RTP_{tot}	RTP_{add}	TP_{base}	TP_{APR}	RTP_{tot}	RTP_{add}	TP_{base}	TP_{APR}	RTP_{tot}	RTP_{add}	TP_{base}	TP_{APR}	RTP_{tot}	RTP_{add}
PraPR	14.53	5.08	15.21	36.96	41.44	35.44	40.38	42.47	44.55	41.13	44.95	43.50	44.82	42.91	44.35
SimFix	20.51	−33.17	14.21	34.19	36.76	14.99	36.02	38.58	39.25	20.86	38.74	38.90	39.33	37.20	39.03
AVATAR	0.98	1.44	4.31	11.75	11.28	11.48	10.47	14.23	14.24	14.24	14.31	14.25	14.26	14.25	14.30
kPar	−1.75	−0.83	0.66	4.66	3.81	3.93	4.21	5.96	5.98	5.99	4.43	5.99	6.00	6.01	5.90
TBar	−1.08	−1.49	2.79	13.13	11.96	11.66	13.86	15.90	15.90	15.89	15.44	16.73	16.73	16.72	16.14
FixMiner	−1.44	−0.62	0.08	6.52	6.54	6.51	6.07	7.09	7.10	7.10	6.64	7.11	7.11	7.11	8.53
Dynamoth	2.07	1.49	5.44	66.93	67.01	66.95	66.89	71.66	71.71	71.67	71.65	71.74	71.79	71.74	71.77
ACS	0.00	−33.16	-4.12	64.84	64.84	64.42	65.93	85.82	85.82	85.85	85.97	86.30	86.30	86.32	86.32
Arja	20.58	5.33	13.30	37.08	39.33	37.71	39.71	39.71	40.72	40.01	40.92	40.22	40.79	40.48	40.62
Kali	6.80	6.90	10.29	24.34	25.81	25.29	26.16	27.83	28.10	27.87	28.98	28.12	28.17	28.13	28.14
GenProg	14.03	7.85	11.51	24.45	25.79	24.57	25.87	26.24	26.62	26.11	27.14	26.55	26.66	26.49	26.56
RSRepair	14.31	9.00	10.95	29.36	30.47	29.71	31.08	30.75	31.04	30.79	32.02	31.09	31.13	31.06	31.11

is the first study demonstrating that in the APR scenario, the traditional RTP technique even outperforms state-of-the-art APR-specific test prioritization technique at the most cases (e.g., without RTS and with coarse-grained RTS). One potential reason may be that the generated patches share little commonality and it is challenging for TP_{APR} to infer the results of un-executed patches from historical patch executions. In contrast, RTP_{add} executes tests with diverse statement coverage as early as possible and thus can advance buggy patch detection. Furthermore, it is notable that RTP_{add} becomes less effective than TP_{APR} when combined with RTS_{stmt}. Intuitively, the tests selected by coarser-grained coverage criteria (e.g., class-level RTS) tend to exhibit a more diverse distribution of statement coverage than the tests selected by the statement-level RTS. Therefore, when combined with RTS_{stmt}, TP_{APR} has a stronger capability in distinguishing tests based on their historical execution results. Third, we can observe that RTP_{add} outperforms RTP_{tot} in terms of the repair efficiency, which is consistent with the previous findings in traditional regression testing [50, 63].

> *Finding 9: This study demonstrates for the first time that test selection outperforms test prioritization for APR, and their combination could further improve APR efficiency. Also, surprisingly, traditional RTP strategy RTP_{add} outperforms state-of- the-art APR-specific test prioritization TP_{APR} for most cases.*

5 Discussion

Time Costs. Figure 4 presents the reduction of both the time and the number of test executions achieved by different RTS strategies on three representative APR systems (i.e., PraPR, SimFix, and ACS). In particular, we choose PraPR, SimFix, and ACS as they are the latest APR techniques in their belonging categories (i.e., template-based, heuristic-based, and constraint-based APR). Based on the figure, we could have the following observation. In particular, the trends are actually consistent between the time costs and the number of test executions. For example, on ACS, the gap between class-level RTS strategy and other RTS strategies is much larger in terms of both time costs and number of test execution. In fact, such an observation is as expected, as the number of tests in a test suite is often very large and the reduction achieved by RTS is significant. Note that here we already include the RTS overheads into time costs, but it is rather lightweight compared to the patch execution time. For example, for PraPR with the largest project Closure, RTS takes 13 seconds for all patches while the patch execution without RTS takes about 4 hours. Hence, the number of reduced test executions is so large that the diversity in the execution time of different tests would have little effect on the results. In other words, although different tests may have different

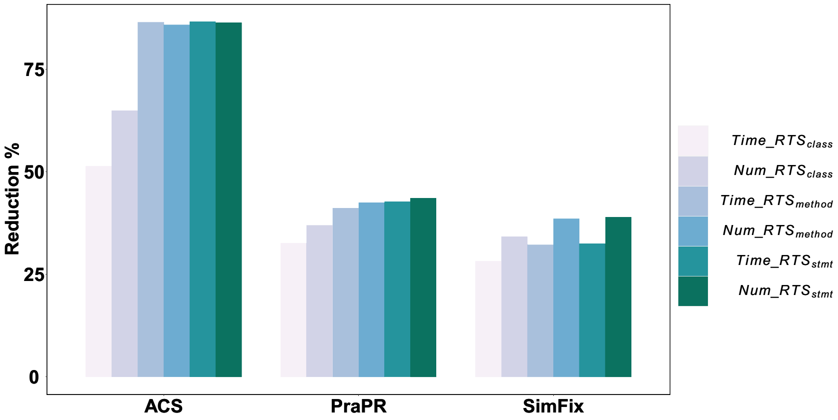

Fig. 4. Reduction in test time vs. number of test executions.

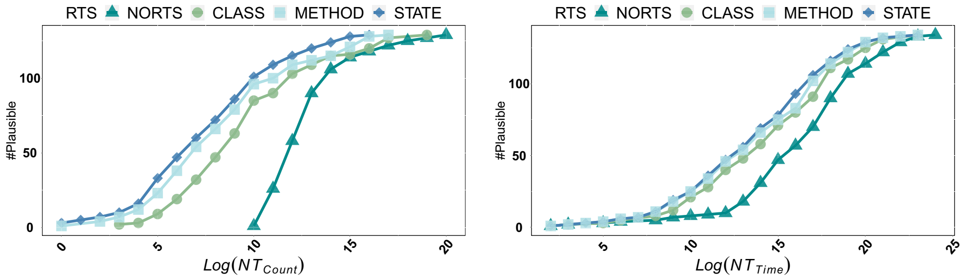

Fig. 5. Plausible patches with different RTS strategies.

execution time, the reduction achieved by RTS strategies is so significant that it exhibits similar trends between the number of test executions and test execution time. In summary, the time costs and number of test executions can be alternative when measuring the efficiency of APR systems. In particular, time costs can demonstrate the efficiency in a more straightforward way while the non-determinism is supposed to be mitigated by multiple executions; the number of executions can demonstrate the efficiency in a more stable way. Our results encourage the community to consider test executions as a status quo metric in the future work.

Repair Effectiveness. We further discuss the impact of different RTS strategies on repair effectiveness. Figure 5 shows the number of plausible patches (the y-axis) found by different number/time of test executions (the x-axis). For space limits, we present PraPR results here, and other results are in our website. The figure indicates a consistent observation with our previous findings: RTS improves APR efficiency, and thus it helps find more plausible patches within the same budgets.

6 Related Work

Since Section 2 presents background about RTS and APR, here we focus on other regression testing techniques and the closely related work for improving APR efficiency.

Regression Testing. Besides RTS, researchers also propose other two categories of regression testing techniques, i.e., RTP [39, 48, 59] and **Test-Suite Reduction (TSR)** [83]. With the common goal of accelerating regression testing, RTP reorders test executions for earlier fault detection, while TSR removes *redundant* tests permanently according to certain testing requirements. Since it is often challenging to identify *redundant* tests, TSR can incur fault detection loss and is not as widely adopted (compared to RTS and RTP). This study excludes TSR because it may produce incorrect patch validation results and reduce the APR accuracy.

APR Efficiency. APR is expensive due to the large number of generated patches and non-trivial costs in patch validation. In addition to many APR approaches that aim to reduce the number of generated patches (mentioned in Section 2.2), researchers propose to reduce the number of test executions for each patch. For example, Qi et al. [59] utilize patch execution history to prioritize tests. Mehne et al. [55] reduce test executions based on statement coverage (i.e., statement-level RTS); similarly, as suggested in Table 1, several existing APR systems also leverage RTS to accelerate patch validation, e.g., the ARJA family [84] adopts statement-level RTS and CapGen [71] adopts class-level RTS. In addition, Mechtaev et al. [54] and Just et al. [27] skip redundant test executions based on program-equivalence, i.e., the test executions of two test-equivalent patches (e.g., exhibiting indistinguishable test results) can be essentially reduced. Lou et al. [8, 9, 49] unify the fault localization and program repair to narrow down the search space of the buggy location, so as to further facilitate the debugging process.

However, the community still lacks a comprehensive understanding on the benefits of RTS for APR, and our work conducts the first study to systematically investigate the impact of representative RTS techniques on a wide range of state-of-the-art APR systems.

Besides reducing the number of test executions, researchers have also looked into speeding up the time for each test/patch execution during APR. For example, JAID [11] and SketchFix [22] transformed the buggy programs into meta-programs or sketches to accelerate patch validation. Ghanbari et al. [17] proposed a bytecode-level APR approach which requires no patch compilation and system reloading. More recently, Chen et al. [12] leveraged on-the-fly patch validation to save patch loading and execution time to speed up all existing source-code-level APR techniques. Such techniques are orthogonal to test execution reduction studied in this work, and they can be combined to further reduce APR cost.

7 Conclusion and Future Work

This work points out an important test execution optimization (RTS) largely neglected and inconsistently configured by existing APR systems. We perform the first extensive study of different RTS techniques for 12 state-of-the-art APR systems on over 2M patches. Our findings include the number of patches widely used for measuring APR efficiency and the inconsistent RTS configurations can both incur skewed conclusions; all studied RTS techniques substantially improve APR efficiency, while method-/statement-level RTS significantly outperform class-level RTS. We also present the RTS impact on different patches and its combination with test prioritization.

In the future, we plan to extend this work with more APR systems and more test selection strategies. In particular, this work currently focuses on traditional APR systems. While given the recent advance in **Large Language Models (LLMs)**, investigating the test execution efficiency in LLM-based APR systems [69, 76, 77] is an essential problem. In addition, this work mainly focuses on the RTS strategies in patch validation, while it remains many open problems on more customized test selection strategies for patch validation, such as leveraging static dependencies or run-time dependencies for test selection. In addition, extending this study to more diverse projects and different programming languages can also be interesting future work.

References

[1] Replication package. 2023. Retrieved from https://github.com/anomynousdata/RTAPR
[2] Apache Camel. 2020a. Retrieved from http://camel.apache.org/
[3] Apache Commons Math. 2020b. Retrieved from https://commons.apache.org/proper/commons-math/
[4] Apache CXF. 2020c. Retrieved from https://cxf.apache.org/
[5] ASM Java bytecode manipulation and analysis framework. 2020. Retrieved from http://asm.ow2.org

[6] JavaAgent. 2020. Retrieved from https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html

[7] Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION '07)*. IEEE, 89–98.

[8] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the effectiveness of unified debugging: An extensive study on 16 program repair systems. In Proceedings of the *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. IEEE, 907–918. DOI: https://doi.org/10.1145/3324884.3416566

[9] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2022. Evaluating and improving unified debugging. *IEEE Trans. Software Eng.* 48, 11 (2022), 4692–4716. DOI: https://doi.org/10.1109/TSE.2021.3125203

[10] Ahmet Çelik, Young-Chul Lee, and Milos Gligoric. 2018. Regression test selection for TizenRT. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE '18)*. 845–850.

[11] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. 637–647.

[12] Lingchao Chen and Lingming Zhang. 2020. Fast and precise on-the-fly patch validation for all. arXiv: 2007.11449.

[13] [13] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Trans Softw. Eng 47* (2019), 1943–1959.

[14] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER '17)*. 349–358.

[15] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test (AST@ICSE '16)*. 85–91.

[16] Sebastian G. Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '22)*. 235–245.

[17] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. 19–30.

[18] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA '15)*. 211–222.

[19] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Softw. Qual. J.* 21 (2013), 421–443.

[20] Giovani Guizzo, Justyna Petke, Federica Sarro, and Mark Harman. 2021. Enhancing genetic improvement of software with regression test selection. In *43rd IEEE/ACM International Conference on Software Engineering (ICSE '21)*. 1323–1333.

[21] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression test selection for java software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '01)*. 312–326.

[22] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 12–23.

[23] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19*) . 255–266.

[24] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '18)*. 298–309.

[25] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain knowledge distilled tree decoder for automated program repair. In *45th IEEE/ACM International Conference on Software Engineering (ICSE '23)*. IEEE, 1251–1263. DOI: https://doi.org/10.1109/ICSE48619.2023.00111

[26] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-aware neural machine translation for automatic program repair. In *43rd IEEE/ACM International Conference on Software Engineering (ICSE '21)*. 1161–1173.

[27] René Just, Michael D. Ernst, and Gordon Fraser. 2014a. Efficient mutation analysis by propagating and partitioning infected execution states. In *International Symposium on Software Testing and Analysis (ISSTA '14)*. Corina S. Pasareanu and Darko Marinov (Eds.), ACM, 315–326. DOI: https://doi.org/10.1145/2610384.2610388

[28] René Just, Darioush Jalali, and Michael D. Ernst. 2014b. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis (ISSTA '14)*. 437–440.

[29] [29] Maria Kechagia, Sergey Mechtaev, Federica Sarro, and Mark Harman. 2021. Evaluating automatic program repair capabilities to repair API misuses. *IEEE Trans. Softw. Eng.* 48 (2021), 2658–2679.

[30] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering (ICSE '13)*. 802–811.

[31] Jindae Kim and Sunghun Kim. 2019. Automatic patch generation with context-based change application. *Empir. Softw. Eng.* 24, 6 (2019), 4071–4106.

[32] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024.

[33] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. 1995. Class firewall, test order, and regression testing of object-oriented programs. *J. Object Oriented Program.* 8, 2 (1995), 51–65.

[34] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. 593–604.

[35] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)* Volume 1:cli. 213–224.

[36] 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 *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. 583–594.

[37] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-based code transformation learning for automated program repair. In *ICSE '20: 42nd International Conference on Software Engineering*. 602–614.

[38] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A novel deep learning-based approach for automated program repair. In *44th IEEE/ACM 44th International Conference on Software Engineering (ICSE '22)*. ACM, 511–523.

[39] Zheng Li, Mark Harman, and Robert M. Hierons. 2007. Search algorithms for regression test case prioritization. *IEEE Trans. Software Eng.* 33, 4 (2007), 225–237.

[40] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019a. You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems. In *12th IEEE Conference on Software Testing, Validation and Verification (ICST '19)*. 102–113.

[41] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019b. AVATAR: Fixing semantic bugs with fix patterns of static analysis violations. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. 456–467.

[42] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019c. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. 31–42.

[43] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. LSRepair: Live search of fix ingredients for automated program repair. In *25th Asia-Pacific Software Engineering Conference (APSEC '18)*. 658–662.

[44] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs. In *ICSE '20: 42nd International Conference on Software Engineering*. 615–627.

[45] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*. 118–129.

[46] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. 727–739.

[47] Fan Long and Martin C. Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. 702–713.

[48] Yiling Lou, Junjie Chen, Lingming Zhang, and Dan Hao. 2019. A survey on regression test-case prioritization. *Adv. Comput.* 113 (2019), 1–46.

[49] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? A unified debugging approach. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 75–87. DOI: https://doi.org/10.1145/3395363.3397351

[50] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How does regression test prioritization perform in real-world software evolution? In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. 535–546.

[51] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 101–114.

[52] Matias Martinez and Martin Monperrus. 2016. ASTOR: A program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16)*. 441–444.

[53] Matias Martinez and Martin Monperrus. 2018. Ultra-large repair search space with automatically mined templates: The Cardumen Mode of Astor. In *Search-Based Software Engineering - 10th International Symposium*, Proceedings (SSBSE '18), Lecture Notes in Computer Science, Vol. 11036:cli. 65–86.

[54] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-equivalence analysis for automatic patch generation. *ACM Trans. Softw. Eng. Methodol.* 27, 4, Article 15 (2018), 37 pages. DOI: https://doi.org/10.1145/3241980

[55] Ben Mehne, Hiroaki Yoshida, Mukul R. Prasad, Koushik Sen, Divya Gopinath, and Sarfraz Khurshid. 2018. Accelerating search-based program repair. In *11th IEEE International Conference on Software Testing, Verification and Validation (ICST '18)*. 227–238.

[56] Atif M. Memon, Zebao Gao, Bao N. Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. 233–242.

[57] Martin Monperrus. 2020. The living review on automated program repair. (2020).

[58] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 241–251.

[59] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2013. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*. 180–189.

[60] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia C. Chesley. 2004. Chianti: A tool for change impact analysis of java programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. 432–448.

[61] Gregg Rothermel and Mary Jean Harrold. 1993. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance (ICSM '93)*. 358–367.

[62] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.* 6, 2 (1997), 173–210.

[63] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *1999 International Conference on Software Maintenance (ICSM '99)*. 179–188.

[64] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. 648–659.

[65] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19*) . 13–24.

[66] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-aware static regression test selection. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), Article 187, 29 pages.

[67] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 532–543.

[68] Song Wang, Jaechang Nam, and Lin Tan. 2017. QTEP: Quality-aware test case prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. 523–534.

[69] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*. Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 172–184. DOI: https://doi.org/10.1145/3611643.3616271

[70] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering*. 356–366.

[71] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 1–11.

[72] Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2020. Boosting automated program repair with bug-inducing commits. In *ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results*. 77–80.

[73] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*. 479–490.

[74] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in Statistics*. Springer, 196–202.

[75] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: Balancing edit expressiveness and search effectiveness in automated program repair. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 354–366. DOI: https://doi.org/10.1145/3468264.3468600

[76] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *45th IEEE/ACM International Conference on Software Engineering (ICSE '23)*. IEEE, 1482–1494. DOI: https://doi.org/10.1109/ICSE48619.2023.00129

[77] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 959–971. DOI: https://doi.org/10.1145/3540250.3549101

[78] Qi Xin and Steven P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. 660–670.

[79] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. 416–426.

[80] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55.

[81] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *44th IEEE/ACM 44th International Conference on Software Engineering (ICSE '22)*. ACM, 1506–1518.

[82] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. In *Proceedings of the 40th International Conference on Software Engineering*. 24.

[83] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verification Reliab.* 22, 2 (2012), 67–120.

[84] Yuan Yuan and Wolfgang Banzhaf. 2020a. ARJA: Automated repair of Java programs via multi-objective genetic programming. *IEEE Trans. Software Eng.* 46, 10 (2020), 1040–1067.

[85] Yuan Yuan and Wolfgang Banzhaf. 2020b. Toward better evolutionary program repair: An integrated approach. *ACM Trans. Softw. Eng. Methodol.* 29, 1 (2020), 5 1–5:53. DOI: https://doi.org/10.1145/3360004

[86] Lingming Zhang. 2018. Hybrid regression test selection. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 199–209.

[87] Hua Zhong, Lingming Zhang, and Sarfraz Khurshid. 2019. TestSage: Regression test selection for large-scale web service testing. In *12th IEEE Conference on Software Testing, Validation and Verification (ICST '19)*. 430–440.

[88] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 341–353.

Received 8 February 2023; revised 15 March 2024; accepted 14 May 2024