On the Effectiveness of Unified Debugging: An Extensive Study on 16 Program Repair Systems

Samuel Benton
The University of Texas at Dallas
Samuel.Benton1@utdallas.edu

Xia Li*
Kennesaw State University
xli37@kennesaw.edu

Yiling Lou*
Peking University
louyiling@pku.edu.cn

Lingming Zhang
University of Illinois at Urbana-Champaign
lingming@illinois.edu

ABSTRACT

Automated debugging techniques, including fault localization and program repair, have been studied for over a decade. However, the only existing connection between fault localization and program repair is that fault localization computes the potential buggy elements for program repair to patch. Recently, a pioneering work, ProFL, explored the idea of *unified debugging* to unify fault localization and program repair in the other direction for the first time to boost both areas. More specifically, ProFL utilizes the patch execution results from one state-of-the-art repair system, PraPR, to help improve state-of-the-art fault localization. In this way, ProFL not only improves fault localization for *manual repair*, but also extends the application scope of *automated repair* to all possible bugs (not only the small ratio of bugs that can be automatically fixed). However, ProFL only considers one APR system (i.e., PraPR), and it is not clear how other existing APR systems based on different designs contribute to unified debugging. In this work, we perform an extensive study of the unified-debugging approach on 16 state-of-the-art program repair systems for the first time. Our experimental results on the widely studied Defects4J benchmark suite reveal various practical guidelines for unified debugging, such as (1) nearly all the studied 16 repair systems can positively contribute to unified debugging despite their varying repairing capabilities, (2) repair systems targeting multi-edit patches can bring extraneous noise into unified debugging, (3) repair systems with more executed/plausible patches tend to perform better for unified debugging, and (4) unified debugging effectiveness does not rely on the availability of correct patches in automated repair. Based on our results, we further propose an advanced unified debugging technique, UniDebug++, which can localize over 20% more bugs within Top-1 positions than state-of-the-art unified debugging technique, ProFL.

*This work was mainly done when they are (visiting) PhD students at UT Dallas.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416566

CCS CONCEPTS

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

KEYWORDS

Unified debugging, Program repair, Fault localization

ACM Reference Format:
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 *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416566

1 INTRODUCTION

With the rapid development of information technology, software systems have been widely adopted in almost all aspects of modern society. However, software bugs (also called software faults in this paper) are inevitable because of the complexity of modern software systems. Software faults can cause software systems to crash or perform unexpected behaviors, both scenarios resulting in disaster, e.g., costing trillions of dollars in financial loss and affecting billions of people [1]. In practice, software debugging is essential for removing bugs from existing faulty software systems. Manual debugging, however, can be extremely challenging, tedious, and costly. Such impediments consume over 50% of the development time/effort [48] and cost the global economy billions of dollars [7].

To date, a huge body of research effort has been dedicated to automated debugging to relieve developer burdens, investigating both fault localization [4, 15, 21, 23, 38, 40, 54, 59, 62, 64] and automated program repair [9, 13, 14, 25, 26, 28–32, 35, 36, 43, 51, 57, 61] techniques. *Fault localization* aims to precisely localize buggy elements within a buggy system based on dynamic and/or static program analysis, and can automatically produce a ranked list of suspicious code elements for developers, reducing their effort for manual bug checking. Classic spectrum-based fault localization (SBFL) techniques [4, 15, 23] mainly analyze the statistical correlation between code coverage and test outcomes to infer potential buggy locations. For example, a code element primarily executed by failed tests are likely to be more suspicious. However, using only coverage information may not be precise enough. Therefore researchers further propose mutation-based fault localization (MBFL) techniques [22, 38, 40, 63] by further considering the impact information between mutated code elements and tests (simulated via mutations). Recently, machine learning techniques have been used

to combine various dimensions of debugging information for more powerful fault localization [6, 21, 22, 60].

While fault localization still requires manual repair, *automated program repair* (APR) aims to directly fix software bugs automatically with minimal human intervention. A typical test-driven APR technique takes a faulty program and its test suite as input and generates program patches with the end goal to find a patch passing all tests. Due to its promising future, various APR techniques have been proposed, including search-based, semantics-driven, and learning-based techniques [14, 20, 35, 59, 61]. For more details, please refer to recent surveys on fault localization [56] and APR [37].

Despite extensive research on automated debugging over the past decades, we still lack practical automated debugging techniques. Current fault localization techniques has limited effectiveness in practice – they either require massive training data that may not always be available [21, 49] or are ineffective for debugging real-world systems [18, 41]. Furthermore, it is rather challenging for APR techniques to fix all possible bugs – even state-of-the-art APR techniques [10, 14, 46] can only fix a small ratio of real bugs (i.e., <20% for Defects4J [16]) automatically.

To enable more practical debugging, the unified debugging approach, ProFL, was recently proposed to unify fault localization and APR to boost both areas [33, 34]. While both fault localization and APR have been studied for over a decade, their only prior connection is that fault localization is leveraged as a supplier for pointing out potentially buggy locations for APR to fix. The unified debugging approach ProFL unifies the two areas in the other direction for the first time, i.e. leveraging large number of patch execution results generated during APR (even when APR fails to fix the bug) to further boost fault localization. The basic intuition is that if a patch passes some originally failing test(s), the patched location is very likely to have some close relationship with the real buggy location (e.g., sharing the same method or even same line), since otherwise the patch cannot mute the bug impact and pass the originally failing test(s). Using the recent PraPR [10] APR system, ProFL is able to substantially boost/outperform state-of-the-art SBFL [3, 15, 23], MBFL [22, 38, 40, 63], and unsupervised/supervised learning based fault localization [21, 49, 64].

In this way, given any buggy project, ProFL not only directly returns the patches when *automated repair* works, but also provides improved fault localization hints for *manual repair* for all other cases. That is, ProFL not only significantly improves fault localization for *manual repair*, but also extends the application scope of *automated repair* to all possible bugs (not only the small portion of bugs that can be automatically fixed).

Despite this promising direction, the ProFL work only considers one APR system (i.e., PraPR), while there are many other available APR systems based on different designs and it is not clear how other APR systems contribute to unified debugging. Therefore, to bridge this gap, we conduct the first extensive study of unified debugging on 16 state-of-the-art APR systems. These 16 systems represent recent public Java APR systems that execute without requiring specialized data or infrastructure. These selected systems utilize constraint-based [9, 36, 58], heuristic-based [14, 35, 61], and template-based [19, 24, 25] repair approaches seen in recent repair literature. Furthermore, we use the Defects4J benchmark suite for our evaluation since it is the most widely used benchmark in recent fault localization and APR work (including the unified debugging work [33]). Our experimental results reveal various practical guidelines for further advancing unified debugging and even software debugging in general. To summarize, this paper makes the following main contributions:

- **Study.** This paper presents the first extensive study of unified debugging using 16 state-of-the-art APR systems.
- **Dataset.** Our detailed experimental data (including patch execution information, experimental script, and result analysis for fault localization, APR, and unified debugging) on the studied Defects4J subjects are publicly available online [2].
- **Guidelines.** Our study reveals various practical guidelines, including: (1) nearly all the studied 16 APR tools can positively contribute to unified debugging despite their varying repairing capabilities, (2) APR tools targeting multi-edit patches bring noise and degrade performance for unified debugging, (3) APR tools with more executed/plausible patches tend to perform better for unified debugging, and (4) unified debugging effectiveness does not exclusively rely on the availability of correct patches from APR.
- **Technique.** Based on our study results, we further propose an advanced unified debugging technique, UniDebug++, which can localize 21% more bugs within Top-1 than state-of-the-art unified debugging technique, ProFL.

2 STUDIED APPROACH

In this section, we first briefly discuss the traditional fault localization and program repair process (Section 2.1) to motivate unified debugging. Then, we present the basic process for the studied unified debugging process (Section 2.2). Lastly, we present a real-world example to further motivate our study in this paper (Section 2.3).

2.1 Fault Localization and Program Repair

Given a buggy program and its failing test suite, test-based fault localization computes each code element's probability to be buggy based on various techniques [4, 5, 21, 40, 65]. For example, the widely studied spectrum-based fault localization (SBFL) [4, 15, 23] will collect the dynamic coverage information for each failing/passing test to compute each code element's suspiciousness value. In this way, developers can choose to directly start *manual repair* with the help of such suspiciousness information.

Alternatively, developers can also choose to directly perform *automated program repair* (APR) [14, 35, 61]. Typical APR techniques leverage fault localization techniques to compute the potential buggy locations for patching, e.g., the Ochiai SBFL technique has been widely used in recent APR work, such as PraPR [10], SimFix [14], and CapGen [55]. After the patch generation and validation, all the *plausible patches* (i.e., the patches that can pass all tests) are returned for manual inspection to find the final *correct patches* (i.e., the patches semantically equivalent to developer patches). In this way, the final correct patches are the only useful outcome from APR; in fact, even plausible but incorrect patches are treated as harmful in traditional APR work [58], since they require time-consuming and tedious manual inspection. However, to date, even state-of-the-art APR can only produce correct patches for a small ratio of real-world bugs, making APR a waste of resources for all the other

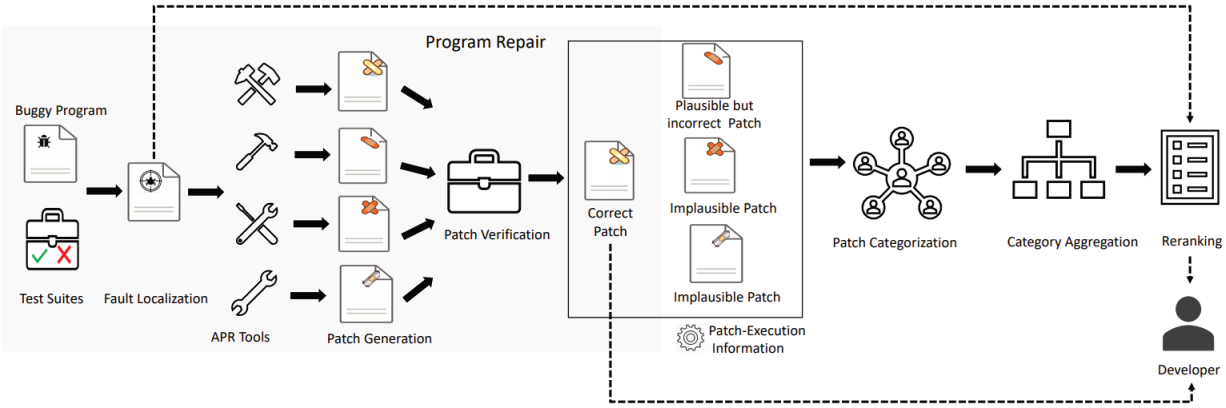

Figure 1: Unified debugging

cases. For example, the current most effective APR work [10, 26, 46] cannot even fix 20% of bugs from the widely studied Defects4J [16].

2.2 Unified Debugging

To further boost both the fault localization and APR areas, unified debugging [33, 34] aims to unify these two areas from the other direction for the first time. The basic insight of unified debugging is that the massive patch execution information from APR (even ones that do not lead to correct patches) can further help substantially improve fault localization to facilitate manual repair. In this way, unified debugging can report correct patches when possible, and more importantly can also return refined fault localization for all cases (even the cases without correct patches). Unified debugging not only extends the application scope of APR to all possible bugs (not only the bugs that can be directly automatically fixed), but also provides more precise fault localization. For example, ProFL [33], the first unified debugging technique based on the recent PraPR APR system, significantly improves/outperforms various state-of-the-art fault localization techniques (e.g., SBFL [4, 15, 23], MBFL [22, 38, 40, 63], and even learning-based techniques [21, 64]).

The basic assumption of unified debugging is that if a patch can pass some originally failing tests, its patch location may be closely related to the actually buggy locations (e.g., sharing the same code element, such as method). Similarly, if a patch fails some originally passing tests, its patch location may be closely related to correct locations since otherwise the passing tests would have been failed before patching [34]. In this way, all the generated patches can be categorized into the following categories according to execution information automatically collected during patch validation for unified debugging: (1) *CleanFix Patches*: patches passing on at least one originally failed test and not failing on any originally passed test, (2) *NoisyFix Patches*: patches passing on at least one originally failed tests but also failing on some originally passed tests, (3) *NoneFix Patches*: patches not impacting the outcome for any originally failed or passed test, (4) *NegFix Patches*: patches not passing any originally failed test but failing on some originally passed tests. Note that all such patch validation information can be directly obtained from the studied test-based APR tools. Unified debugging simply leverages such existing information to classify each patch into the aforementioned categories.

Table 1: Example of Math-32

Suspicious Method	SBFL	PraPR	Kali-A	TBar	UniDebug+
`PolyhedronsSet.<init>`	1.0	NoneFix	Unmodified	Unmodified	NoneFix
`PolygonsSet.compute...`	1.0	NoneFix	CleanFix	NoneFix	CleanFix
`PolygonsSet.followLoop`	1.0	NoneFix	NoneFix	Unmodified	NoneFix
`AVLTree.getNotSmaller`	1.0	NoneFix	NoneFix	NoneFix	NoneFix

The overall approach of unified debugging is presented in Figure 1. Given any buggy program and its test suite, unified debugging first applies off-the-shelf APR systems to generate and execute various possible patches. Then, while existing APR work only returns the correct patches to the developers, unified debugging further utilizes the execution information for all patches and categorizes them into relevant categories discussed in the previous paragraph. Then, for each code element (e.g., method), unified debugging then adopts the best category from its corresponding patches according to this predefined order (i.e., CleanFix > NoisyFix > NoneFix > NegFix) [34]. Finally, all the elements are re-ranked first according to their patch categories, e.g., all elements with the CleanFix category are ranked higher than all elements with the NoisyFix category; after that, the elements within the same category are then further re-ranked in the descending order by their initial suspiciousness scores computed by any existing fault localization technique (i.e., Ochiai [4] by default). In this way, the developers will obtain largely refined fault localization for all possible bugs (even including the case where no correct or plausible patch is found).

2.3 Motivating Example

While the existing unified debugging technique ProFL has demonstrated promising results, in this section, we use Math-32 from Defects4J (V1.0.0) [16], a widely used real-world Java bug benchmark, to motivate our study. Math-32 denotes the 32nd buggy version of Apache Commons Math project. The bug is located in method `computeGeometricalProperties` of Class `PolygonsSet`.

Table 1 shows 4 example suspicious methods including the actual buggy method shown in gray. Please note that we disregard the arguments since the class and method names can sufficiently distinguish them.

In the table, Column "SBFL" indicates the suspiciousness score of each method according to the state-of-the-art SBFL technique Ochiai [4] with aggregation strategy [49], which aggregates the maximum suspiciousness values from statements to methods and

```
protected void computeGeometricalProperties() {
    ...
    if (v.length == 0) {
        final BSPTree<Euclidean2D> tree = getTree(false);
-       if ((Boolean) tree.getAttribute()) {
+       if (false) {
            // the instance covers the whole space
            setSize(Double.POSITIVE_INFINITY);
            setBarycenter(Vector2D.NaN);
    ...
}
```

```
protected void computeGeometricalProperties() {
    ...
    if (v.length == 0) {
        final BSPTree<Euclidean2D> tree = getTree(false);
-       if ((Boolean) tree.getAttribute()) {
+       if (tree.getCut() == null && (Boolean) tree.getAttribute()) {
            // the instance covers the whole space
            setSize(Double.POSITIVE_INFINITY);
            setBarycenter(Vector2D.NaN);
    ...
}
```

Figure 2: Generated patch of Kali-A and developer patch for Math-32

has been demonstrated to substantially outperform raw method-level SBFL. Columns "PraPR", "Kali-A", and "TBar" represent the unified debugging approaches using the patch-execution information from APR systems PraPR, Kali-A, and TBar, respectively. The patch category information for each method is included in the table. Unmodified, hereby referred to as Non-Modify, represents a new patch category implying that these code elements are never patched by an APR tool. Lastly, Column "UniDebug+" presents the technique simply using all patches from the prior three APR systems. From the motivating example, we have the following interesting findings:

First, unified debugging using other APR systems can have promising fault localization results even when the PraPR system used by ProFL cannot help improve the performance. For example, Kali-A can directly rank the buggy method at the 1st location, while both SBFL and ProFL with PraPR rank the bug at the 4th location. Figure 2 represents the patch generated by Kali-A (left side) and the correct patch provided by developers (right side). From the patches, we found that Kali-A generates a patch by changing the buggy conditional statement into `if (false)` which is useless for fixing the real bug; however, this patch does help pinpoint the actual bug location, demonstrating the generality of unified debugging for all possible APR systems. This finding motivates us to perform an extensive study to investigate the effectiveness of different APR systems for unified debugging. Second, different APR systems have different unified debugging performances and combining them may potentially result in even more powerful unified debugging. Shown in the last column of Table 1, simply combining all patches generated by different APR systems can also localize the bug within Top-1.

3 STUDY DESIGN

3.1 Research Questions

In this study, we aim to investigate the following research questions:

- **RQ1:** How does unified debugging perform with all studied APR systems?
- **RQ2:** How do unmodified code elements during APR impact unified debugging?
- **RQ3:** How does unified debugging correlate with program repair effectiveness?
- **RQ4:** How do we further advance state-of-the-art unified debugging with all studied APR systems?

Tool Category	Tools
Constraint-based	ACS, Cardumen, Dynamoth
Heuristic-based	Arja, GenProg-A, jGenProg, jKali, jMutRepair, Kali-A, RSRepair-A, Simfix
Template-based	AVATAR, FixMiner, kPar, PraPR, TBar

Table 2: Repair systems studied

Project	Name	# Bugs	# Tests	LOC
Chart	JFreeChart	26	2,205	96K
Lang	Apache Lang	65	2,245	22K
Math	Apache Math	106	3,602	85K
Time	Joda-Time	27	4,130	28K
	Total	**224**	**12,362**	**231K**

Table 3: Studied bugs from Defects4J v1.0.0

3.2 Experimental Setup

For this study, we considered all the 16 program repair systems accessible from a recent study [27]. Furthermore, we also considered the recent PraPR repair system [10] which the initial unified debugging work is based on. Table 2 shows the breakdown of all the APR systems studied, including: heuristic-based - Arja [61], GenProg-A [61], jGenProg [35], jKali [35], jMutRepair [35], Kali-A [61], RSRepair-A [61], and Simfix [14]; constraint-based - ACS [58], Cardumen [36], and Dynamoth [9]; and template-based - AVATAR [25], FixMiner [19], kPar [24], TBar [26], and PraPR [10]. We manually modified all studied APR systems to collect the detailed patch execution information required by unified debugging, and ensured that our modified version did not impact the tool functionality. [1] Each system used original time settings suggested by the original papers.

Most of the studied APR systems have been implemented to target version 1.0.0 (and older) of the widely used Defects4J benchmark [16], which includes 357 real-world bugs from five real-world software systems. Meanwhile, in our evaluation process, we found that many of the studied tools (e.g., CapGen, ACS, Arja, GenProg-A, etc.) do not support or cannot successfully execute Closure from Defects4J 1.0.0 (according to their original publications). Therefore, for fair comparison, all of our experiments are performed on the remaining four subjects from Defects4J 1.0.0; the Chart, Time, Lang, and Math projects. Detailed statistics are shown in Table 3.

Each tool was executed using each 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. Thus,

[1] Note that we fail to get the our modified version of Nopol functional due to the specific design of Nopol. Nevertheless, we believe that excluding one specific tool does not impact the general findings and contributions of this study (especially we also have many other constraint-based APR systems studied).

in our experiments, we ultimately utilized two JDK versions, JDK 1.8.0.242 (hereby referred to as JDK 1.8) and JDK 1.7.0.80 (hereby referred to as JDK 1.7). Systems Simfix and Dynamoth executed using JDK 1.8 exclusively. Systems Cardumen, jGenProg, jKali, and jGenProg executed using JDK 1.8 and validated system test suites with JDK 1.7. All other systems executed using JDK 1.7 exclusively.

All our experiments were conducted within the following environment: 36 3.0GHz Intel Xeon Platinum Processors, 60GBs of memory, and Ubuntu 18.04.4 LTS operating system.

3.3 Implementation Details

3.3.1 ProFL Configuration. Although unified debugging can be used to refine any existing fault localization technique, by default, the original unified debugging work, ProFL, utilizes APR to refine state-of-the-art SBFL technique, Ochiai [3] with aggregation strategy [49]. Actually, the original ProFL work demonstrates that unified debugging has consistent performance for refining different state-of-the-art fault localization techniques. Therefore, in this paper, we also focus on using Ochiai (with aggregation) to investigate the impact of different repair systems. Furthermore, following the original ProFL work, this study also focuses on method-level fault localization (i.e., localizing potential buggy methods), as researchers have demonstrated that class-level fault localization can be too coarse-grained [18] while statement-level fault localization can be too fine-grained and miss necessary contextual information [41].

3.3.2 Non-Modify Category. For the original ProFL work, the used PraPR repair system [10] is extremely fast due to the bytecode-level manipulation and can generate patches for almost all the possible *suspicious* methods, i.e., methods executed by failed tests (since methods not executed by failed tests should not be responsible for the current test failures). However, for all other APR tools, there may exist many suspicious methods without any patch, since it is expensive for APR tools to generate patches for every method. Therefore, besides the four categories of methods mentioned in Section 2.2, we create a new category, Non-Modify, to represent the methods that do not receive any patch for a specific APR system. It is unclear how this new category compares with the other four categories studied in the original unified debugging work. Therefore, we explore the impact of the ranking this new Non-Modify category within the existing four ProFL categories in Section 4.2. Note that as the default setting, we put Non-Modify alongside the NegFix category since the plurality of *all patches* fall into the NegFix category (thus the majority of Non-Modify methods may also fall into the NegFix category if they had been generated with patches).

3.3.3 Repair Tool Integration with ProFL. The original ProFL tool has been implemented as a publicly available Maven plugin. We obtained the original ProFL source code from the authors and analyzed the interface between ProFL and its underlying APR system. Then, we modified all the 16 studied APR systems to produce detailed patch execution information consistent with the original ProFL interface (e.g., regarding the patch location, failing and passing tests for each patch). In this way, we can safely replace the original PraPR system with any other studied APR systems for our study. Please note that we also augment the original ProFL code to handle the new Non-Modify method category.

3.4 Evaluation Metrics

Following prior work [21, 49, 64], we measure the number of bugs localized within Top-1, Top-3, and Top-5 positions as the primary metrics for this study. The reason is that researchers have observed that most developers will abort automated debugging tools if they cannot return the actual buggy elements within the Top-5 positions [18]. Specifically, given a set of methods which tie for the same rank, each method is assigned the **worst** rank of the tied methods, following prior work [21, 22, 33]. Furthermore, we also present the mean first rank (MFR) and mean average rank (MAR) results widely used in prior fault localization [21, 22] and unified debugging [33] work. More specifically, for precise localization of all buggy elements of each bug, we compute the average ranking of all the buggy elements for each bug; MAR is simply the mean of the average ranking of all bugs. Similarly, for a bug with multiple buggy elements,the localization of the first buggy element is critical since the rest buggy elements may be directly localized after that; therefore, we use MFR to compute the mean of the first buggy element's rank for each bug.

4 RESULT ANALYSIS

4.1 RQ1 - Performance of Unified Debugging with Different APR systems

In this research question, we first investigate the effectiveness of unified debugging on all the 16 studied APR systems. Figure 3 shows the fault localization results on all the studied subjects (i.e., Lang, Chart, Time, and Math from the Defects4J benchmark) in terms of the Top-1, Top-3, Top-5, MFR, and MAR metrics. The upper sub-figure represents the Top-N results and bottom sub-figure indicates the MFR/MAR results. Each bar in both sub-figures represents different APR systems. Note that we use the default treatment for the Non-Modify category which inserts such methods alongside the NegFix category, i.e., CleanFix > NoisyFix > NoneFix > NegFix = Non-Modify (discussed in Section 3.3.2). Also note that the 16 repair systems in this figure are ordered chronologically with respect to the date for each publication, following the existing APR study [27]. We also include the result of state-of-the-art SBFL (i.e., Ochiai with aggregation) and the first unified debugging technique (i.e., ProFL with PraPR) in the last for comparison (Note that ProFL has been demonstrated to outperform/improve all state-of-the-art fault localization [33, 34]). From the figure, we have the following observations. First, unified debugging with most APR systems performs better than state-of-the-art SBFL! For example, in terms of Top-1, 15 out of 16 tools can help improve SBFL and only Arja fails to meet the initial SBFL results. That said, Arja can still localize 73 faults within Top-1 which is fairly close to the SBFL result. This finding indicates the broad applicability of the unified debugging approach. Second, even though existing APR study [27] has observed that more recent APR systems can fix more bugs than earlier systems, there is no obvious trend showing that unified debugging with more recent (i.e., chronologically later) APR systems can help localize more bugs than earlier ones. This finding demonstrates that APR systems' capability to produce correct patches is not highly correlated to the unified debugging effectiveness in fault localization. Lastly, different results of the 16 APR systems indicate that

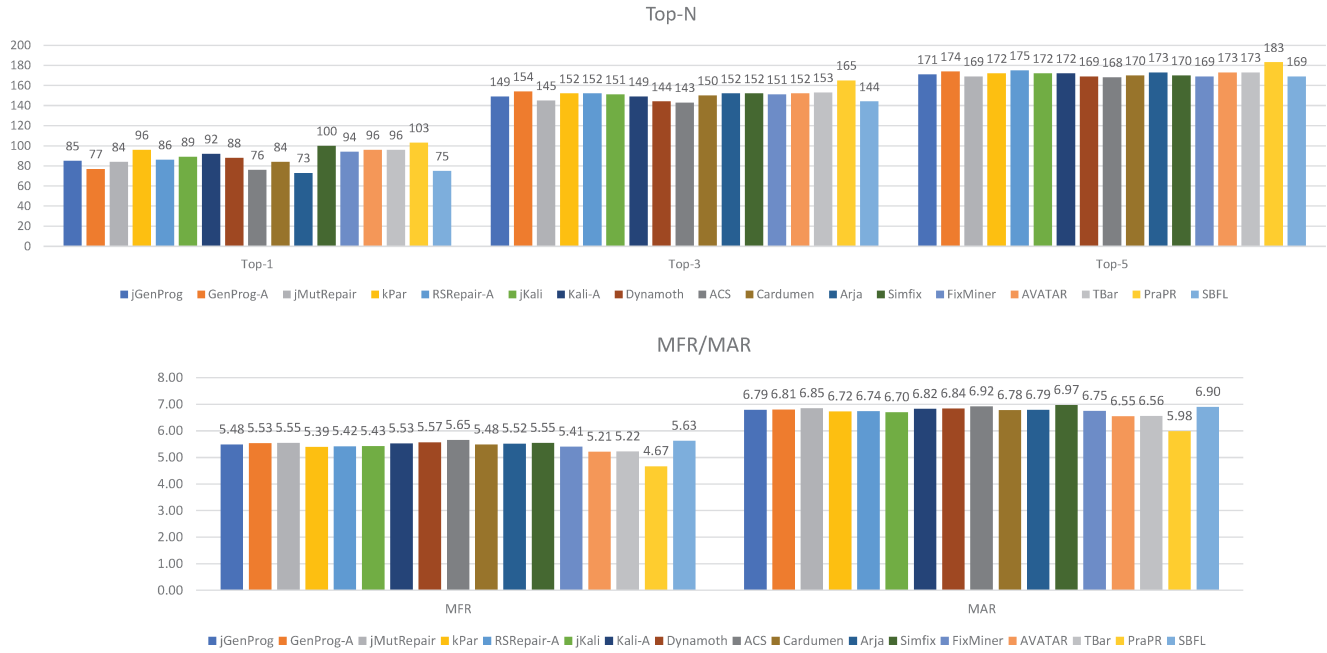

Figure 3: Unified debugging results with all studied APR systems

each APR system has its own advantages and disadvantages for unified debugging. The potential reason is that some tools have exclusive abilities to repair various classes of bugs by leveraging different algorithms to generate patches, incurring various levels of effectiveness for fault localization. This finding further motivates us to combine multiple APR systems to advance state-of-the-art unified debugging (studied in Section 4.4).

> **Finding 1:** Despite their varying repair capabilities, almost all the studied 16 APR systems individually boost state-of-the-art SBFL and contribute to unified debugging.

4.1.1 Qualitative Analysis. Now we perform a detailed qualitative analysis to investigate the different performance of different APR systems. Table 4 shows a subset of the unified debugging results with different APR systems from Math-77. Note that we also include the results for UniDebug+, which simply uses all patches from different APR systems. Column "EID" describes the ID for each method. Column "Category" represents the category computed for each method based on each tool. Column "Rank" describes how each method ranks in the final results. Math-77 fails on two tests, `testBasicFunctions` within class `ArrayRealVectorTest` and `testBasicFunctions` within class `SparseRealVectorTest`, both from the `org.apache.commons.math.linear` package. The buggy methods for Math-77 involve modifications of methods `e4` and `e5`, according to developer patch in Figure 4. According to traditional SBFL, all five methods tie and are ranked 5th (according to the *worst* ranking). We next discuss the performance of three example APR systems for unified debugging:

TBar is able to generate a CleanFix patch by exclusively modifying method `e4`, shown in Figure 5. This CleanFix patch successfully passes one of the originally failed tests, and passes every other test. Even though this patch is not a correct patch, it does help to boost the rank of one buggy method to Top-1. The reason is that the patch shares the same location with the bug and thus is able to mute the bug impact via modifying the return value. This further demonstrates the effectiveness of unified debugging.

RSRepair-A generates 43 NegFix and 355 NoneFix patches across 14 unique methods for this bug. All five methods in Table 4 are NoneFix based on RSRepair-A. From this categorization, `e1` - `e5` are ranked the same as the SBFL results. Note that, in this case, although RSRepair-A was not able to improve SBFL, it will not deteriorate the fault localization results when combining with the more effective TBar. The reason is that when putting all patches together, methods with higher patch categories will still be ranked higher.

Arja is a very interesting case. It actually produces many incorrect but CleanFix patches for Math-77, including for all five suspicious methods shown in Table 4. We were surprised by the fact that Arja can produce so many CleanFix since they are usually hard to generate. Digging into various such patches, we found the reason to be that Arja specifically targets multi-edit patches (i.e., each patch modifies multiple program locations). For example, one such CleanFix patch is shown in Figure 6. In this way, as long as one/part of the multiple edits within a multi-edit patch can make some failing tests to pass, the patch can potentially be CleanFix, making all modified methods of this patch to be highly ranked. Furthermore, such noise incurred by multi-edit APR systems can also be harmful when combining different APR systems for unified debugging. For example, shown in the last column of Table 4, UniDebug+ also cannot distinguish the five suspicious methods. Therefore, we exclude all such multi-edit APR tools when combining different APR systems for unified debugging (Section 4.4) to remove unnecessary noise.

EID	Suspicious Method	SBFL		TBar		RSRepair-A		Arja		UniDebug+	
		Susp.	Rank	Category	Rank	Category	Rank	Category	Rank	Category	Rank
e1	AbstractRealVector:getL1Norm()D	0.707	5	Non-Modify	5	NoneFix	5	CleanFix	5	CleanFix	5
e2	AbstractRealVector:getNorm()D	0.707	5	Non-Modify	5	NoneFix	5	CleanFix	5	CleanFix	5
e3	ArrayRealVector:getL1Norm()D	0.707	5	Non-Modify	5	NoneFix	5	CleanFix	5	CleanFix	5
e4	ArrayRealVector:getLInfNorm()D	0.707	5	**CleanFix**	**1**	NoneFix	5	CleanFix	5	CleanFix	5
e5	OpenMapRealVector:getLInfNorm()D	0.707	5	Non-Modify	5	NoneFix	5	CleanFix	5	CleanFix	5

Table 4: Unified debugging with different APR systems for Math-77

```
org.apache.commons.math.linear.ArrayRealVector.java
public double getLInfNorm() {
    double max = 0;
    for (double a : data) {
-       max += Math.max(max, Math.abs(a));
+       max = Math.max(max, Math.abs(a));
    }
    return max;
}
```

```
org.apache.commons.math.linear.OpenMapRealVector.java
- public double getLInfNorm() {
-     double max = 0;
-     Iterator iter = entries.iterator();
-     while (iter.hasNext()) {
-         iter.advance();
-         max += iter.value();
-     }
-         return max;
- }
```

Figure 4: Correct developer patch for Math-77

```
org.apache.commons.math.linear.ArrayRealVector.java
public double getLInfNorm() {
    double max = 0;
    for (double a : data) {
        max += Math.max(max, Math.abs(a));
    }
-        return max;
+        return getDimension();
}
```

Figure 5: TBar's incorrect CleanFix patch for Math-77

> **Finding 2:** APR tools specifically targeting multi-edit patches can bring noise into unified debugging, as each multi-edit patch involves multiple modifications and many modifications are not helpful in muting the bug impacts even the patch can pass some originally failing test(s).

4.1.2 Quantitative Analysis. Since the capability to produce correct patches is not highly correlated with unified debugging effectiveness, we further performs detailed quantitative analysis to explore what factors of APR systems are highly correlated to the effectiveness of unified debugging. Figure 7 represents the correlation analysis between different factors of APR systems and representative fault localization metrics (i.e., Top-1 and MFR). Note that we also excluded APR tools targeting multi-edit patches. In this figure, "TotalPatch" represents the number of all executed compilable patches generated from each APR tool, "MethodByTotal" represents the number of unique methods modified by all executed patches, "PlausiblePatch" represents the number of plausible patches generated by each tool, and "MethodsByPlausible" represents the number of unique methods covered by the plausible patches. Within each sub-figure, each data point represents one studied APR system, and we perform Pearson Correlation Coefficient analysis [42] at significance level of 0.05. From this figure, we can observe that APR systems tend to perform significantly better for unified debugging when executing more patches for more methods and/or producing more plausible patches for more methods. The finding is statistically significant for all the sub-figures at the significance level of 0.05. Although the finding is surprisingly uniform, this makes intuitive sense, since APR systems patching more code elements tend to accumulate more information for debugging. This finding suggests future APR systems to explore more patches for more powerful unified debugging, and also calls for research for faster patch execution (otherwise APR systems cannot afford massive patch executions).

> **Finding 3:** APR systems executing more patches across code elements and/or producing more plausible patches tend to perform better for unified debugging, calling for future research on fast & exhaustive patch exploration .

4.2 RQ2 - Impacts of Non-Modify Code Elements on Unified Debugging

As discussed in Section 3.3.2, we add one new patch category, Non-Modify, which represents suspicious methods that some APR tools do not afford to modify. The first research question has demonstrated the effectiveness of default unified debugging setting on 16 APR systems by treating Non-Modify equivalently as the fourth patch category NegFix. In this research question, we further evaluate the unified debugging effectiveness when casting the Non-Modify category into each of the four different ProFL categories. Figure 8 represents the representative Top-1 and MFR results of all the 16 APR systems with such four settings represented with lines in different colors. From the figure, we can observe that for most systems, casting Non-Modify code elements into the last NegFix category performs better than casting them into other three categories in terms of both Top-1 and MFR. For example, Simfix can localize 100 bugs within Top-1 when casting Non-Modify into NegFix category, and can only localize 95/93/90 bugs within Top-1 when casting Non-Modify into NoneFix/NoisyFix/CleanFix category. Also, in terms of MFR, Kali-A can achieve 5.53 with NegFix, which is also significantly better than Kali-A with other three categories (6.76/6.25/6.45). The reason is that NegFix patches are more prevalent for most code elements (including Non-Modify ones), while other patch categories can be harder to generate.

> **Finding 4:** Non-Modify code elements (i.e., elements with no patches) can be treated in the same way as elements with only NegFix patches (i.e., the patches that cannot fix any failing test but can cause originally passing tests to fail) for precise unified debugging.

```
// ArrayRealVector.java
public double getLInfNorm() {
    double max = 0;
    for (double a : data) {
        max+=Math.max(max,Math.abs(a));
    }
-   return max;
+   return data.length;
}
```

(a) Modifications for ArrayRealVector.java

```
// OpenMapRealVector.java
public double getLInfNorm() {
    double max = 0;
    Iterator iter=entries.iterator();
    while (iter.hasNext()) {
        iter.advance();
        max += iter.value();
    }
-       return max;
+       return virtualSize;
}
```

(b) Modifications for OpenMapRealVector.java

```
// AbstractRealVector.java
public double getL1Norm() {
    double norm = 0;
    Iterator<Entry> it = sparseIterator();
    Entry e;
+   while (it.hasNext()&&(e=it.next())!=null){
+       norm += Math.abs(e.getValue());
+   }
    while (it.hasNext()&&(e=it.next())!=null){
        norm += Math.abs(e.getValue());
    }
    return norm;
}
```

(c) Modifications for AbstractRealVector.java

Figure 6: An incorrect Arja CleanFix patch (with three modified methods) for Math-77

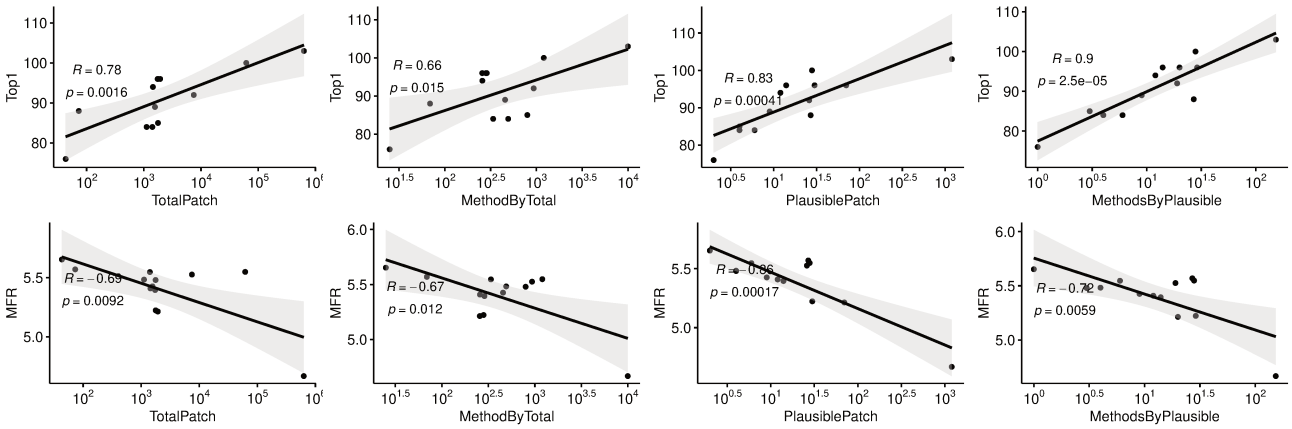

Figure 7: Correlation analysis

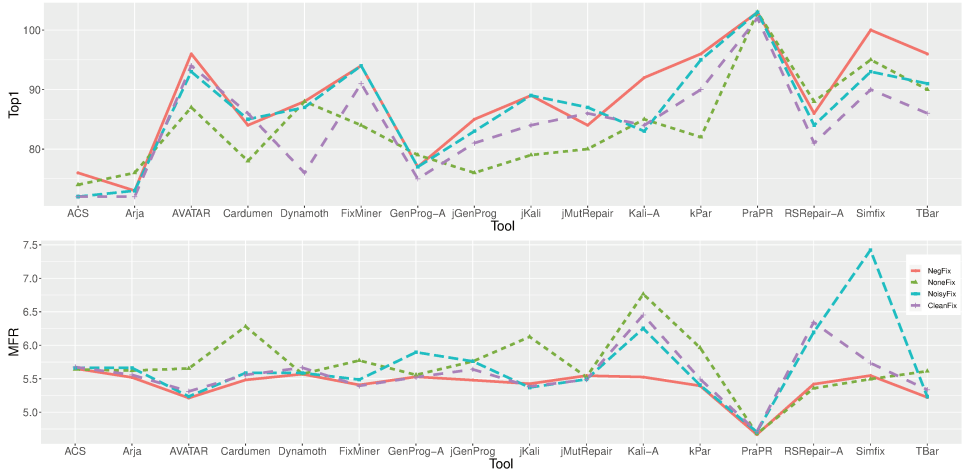

Figure 8: Impact of casting Non-Modify code elements into different categories

4.3 RQ3 - How Does Unified Debugging Correlate with APR Effectiveness?

APR systems all aim to produce correct patches for as many bugs as possible. However, this is a rather challenging goal, and even state-of-the-art APR tools can only fix less than 20% of the studied bugs [10]. Therefore, in this research question, we empirically study whether unified debugging is also limited by the APR effectiveness (i.e., in producing correct patches). The three sub-figures in Figure 9 show the representative Top-1 metric for SBFL and unified debugging using each APR system on (1) buggy versions where the corresponding APR system has correct patches, (2) buggy versions with incorrect but plausible patches, and (3) buggy versions without even plausible patches, respectively. Please note that we omit the APR systems that did not present detailed correct patch IDs in their original publications in this figure. From the figures, we can observe that different APR systems perform differently in all three different bug sets, and almost all APR systems can contribute to unified debugging to outperform SBFL. More importantly, we

Figure 9: Unified debugging on buggy versions with (1) correct, (2) incorrect but plausible, and (3) implausible patches

Table 5: Effectiveness of UniDebug+ and UniDebug++

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
SBFL	75	144	169	5.63	6.90
ProFL	103	165	183	4.67	5.98
UniDebug+$_{all}$	95	160	177	4.84	6.16
UniDebug++$_{all}$	**119**	**168**	**180**	**4.48**	**5.86**
UniDebug+	110	168	182	4.49	5.88
UniDebug++	**125**	**172**	**184**	**4.27**	**5.71**

observe that almost all APR systems consistently outperform SBFL in all the three bug sets. One potential reason is that as long as a patch can pass some originally failing test(s), its patch location may be closely related to the actual buggy location since otherwise it cannot mute the bug impact to pass failing tests. In this way, patches do not need be correct or even plausible to contribute to unified debugging. Furthermore, even the patches that only make originally passing tests turn to fail can help eliminate the potentially correct/benign locations to also boost unified debugging. This further demonstrates the general applicability and promising future for unified debugging.

> **Finding 5:** Unified debugging effectiveness does not rely on the availability of correct or even plausible patches from APR. Similar as when conducting manual program repair, APR patch execution results from even incorrect/implausible patches can still reveal actual buggy locations (when they pass some failing test(s)) or eliminate correct locations (even when they only fail on originally passing tests).

4.4 RQ4 - More Advanced Unified Debugging

To combine the strengths of different APR systems, one naive way is to simply combine the patches of different APR systems for unified debugging, i.e., the UniDebug+ technique that we have talked about.

Table 6: Example for UniDebug++

	SBFL	Tool1	Tool2	Tool3	UniDebug+	UniDebug++
e1	0.8	CleanFix	CleanFix	CleanFix	CleanFix	CleanFix(3)
e2	0.8	CleanFix	NegFix	CleanFix	CleanFix	CleanFix(2)
e3	0.8	CleanFix	NoneFix	NoneFix	CleanFix	CleanFix(1)

In this way, the final category information for a code element can be determined by the best category information of all patches of the code element from the combined APR systems. In this section, we further propose a more advanced technique, UniDebug++, which further distinguishes code elements with the same suspiciousness values in the same category. More specifically, after assigning the patch group category (for patches generated by all combined APR systems) to a code element in the category aggregation step (shown in Figure 1), we further count the total number of APR systems that generate patches in the same category as this code element.

The intuition is that if more APR systems can assign the best category information to a code element, this element should have higher priority in the ranked list compared to its tied peers. Table 6 shows an simple example to illustrate UniDebug++. In this example, elements `e1`, `e2` and `e3` have the same SBFL suspiciousness value 0.8 and are all in the CleanFix category according to UniDebug+, therefore they cannot be distinguished when using UniDebug+. In contrast, UniDebug++ further considers the number pf APR systems producing CleanFix patches for each element. For example, `e1` has CleanFix patches when using all three APR systems and should be ranked higher than other elements. In this way, we can leverage more precise APR information for more powerful unified debugging.

Table 5 shows the results of original SBFL, ProFL, UniDebug+ and UniDebug++ in terms of Top-1, Top-3, Top-5, MFR and MAR. Note that as discussed in Section 4.1.1, APR systems specifically targeting multi-edit patches can introduce extra noise for unified debugging and have been excluded for UniDebug+ and UniDebug++. Meanwhile, we also include, as references, their variants which consider all studied APR systems, denoted as UniDebug+$_{all}$ and UniDebug++$_{all}$ in the table. From the results, we have the following observations. First, UniDebug+$_{all}$ and UniDebug++$_{all}$ perform worse than UniDebug+ and UniDebug++, respectively. In fact, UniDebug+$_{all}$ even performs worse than ProFL which only uses the PraPR APR system. This finding further confirms our earlier qualitative analysis and *Finding 2* that APR systems targeting multi-edit patches are not suitable for unified debugging. Second, both UniDebug+ and UniDebug++ can significantly outperform SBFL and ProFL in all metrics. For example, UniDebug+ can localize 110 bugs within Top-1, i.e., 35/7 more than SBFL/ProFL. Third, UniDebug++ can achieve the best result (even comparing against our own UniDebug+), localizing 125 faults within Top-1, i.e., 50/22 more than state-of-the-art SBFL/ProFL.

Shown in Section 4.1, APR systems with more plausible patches tend to perform better in unified debugging. Therefore, we further study the impact of having different subsets of APR systems for UniDebug+ and UniDebug++. To that end, we rank all APR systems in descending order of the number of *additional* bugs that each APR system can come up with plausible patches. In this way, we can observe the effectiveness trend of UniDebug+ and UniDebug++ with more and more APR systems. Figure 10 presents Top-1

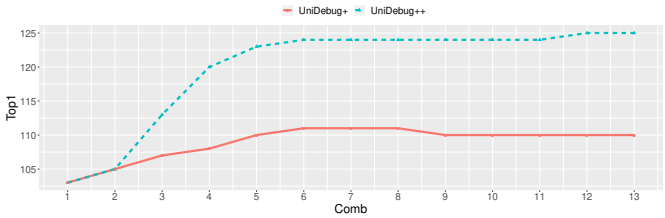

Figure 10: UniDebug+ and UniDebug++ with increasing number of APR systems

results when including more and more APR systems under UniDebug+ and UniDebug++. From the figure, we can observe several interesting findings. First, both UniDebug+ and UniDebug++ overall have increasing effectiveness when including more and more APR systems. Second, we also observe that both techniques tend to saturate when new systems cannot provide much additional plausible patches. This actually indicates that a small subset of the studied APR systems (e.g., 6 of them) can be combined to achieve same effectiveness as the whole set. Third, UniDebug++ has much better effectiveness compared to UniDebug+ which simply puts all patches from different APR systems together. The reason is that most APR systems are helpful for fault localization, and the code elements ranked high by multiple APR systems are indeed more likely to be buggy.

Please note that running multiple APR systems for UniDebug+ or UniDebug++ can be costly. Although the efficiency issue is out of scope for this paper, our above finding demonstrates that a small subset (e.g., 6) of them can already be sufficient for effective unified debugging. Also, Chen et al. [8] have recently proposed a general on-the-fly patch-validation framework to substantially speed up existing APR systems. Furthermore, such APR systems can be run in parallel to further increase the potential for improved unified debugging with minimal delay; this not only 1) improves the probability for the buggy project to be directly automatically fixed via multiple APR systems, but also 2) further boosts fault localization for manual repair even when none of the APR systems can directly fix the bug.

> **Finding 6:** Our new unified debugging technique UniDebug++ considering common behaviors between multiple APR systems can localize 125 bugs within Top-1, which significantly improves the state-of-the-art ProFL by over 20%.

4.5 Threats to Validity

4.5.1 Internal Validity. All of our study results are directly dependent on the correctness of our implementation of all the studied techniques. Faulty implementations in any aspect will yield misleading / inaccurate results. To mitigate this threat, we reuse the implementation of the SBFL and ProFL techniques obtained from the ProFL authors. We also obtained the source code for all the studied APR systems from the authors to investigate their impact on unified debugging. Furthermore, we execute both APR systems with our modifications (to record detailed patch execution information required by unified debugging) and the original APR systems to ensure that they produce the same results (i.e., our modification does not change the APR behavior).

4.5.2 Construct Validity. This threat mainly lies in the dependent variables or metrics used in this study. To reduce such threats, we adopted the most widely used metrics in recent fault localization [21, 22, 49] and unified debugging [33, 34] studies.

4.5.3 External Validity. To evaluate on real-world bugs, we choose the Defects4J v1.0.0 dataset with hundreds of real-world bugs, which is the most widely used benchmark suite for recent APR and fault localization work. However, the study results may still not generalize to all possible systems in the wild, and we plan to further enlarge our benchmark selection in the near future.

Also, since the study aims to investigate the impact of APR systems for unified debugging, different APR systems may yield totally different results. Therefore, we simply studied all the state-of-the-art APR systems that (1) have publicly available source code and (2) are applicable to the used Defects4J benchmark [27].

5 RELATED WORK

As the studied unified debugging approach unifies traditional fault localization and automated program repair (APR) to boost both areas, in this section, we talk about the related work in both areas.

5.1 Fault Localization

The basic idea of fault localization is to automatically produce a ranking list of code elements (e.g., program methods or statements) based on the descending order of their suspiciousness values (i.e., the probability of being buggy) to help developers in manual debugging or serve as the supplier for APR. Various fault localization techniques have been proposed over the past decades. Spectrum-based fault localization (SBFL) [39, 47], one of the most classic fault localization approaches, has been intensively studied due to its effectiveness and scalability. Its basic insight is that code elements primarily executed by failed tests are more suspicious than elements primarily executed by passed tests.

To date, various formulae (e.g., based on statistical analysis or other heuristics) have been proposed to compute code element suspiciousness, such as Tarantula [15], Ochiai [3], SBI [23], and so on. One main limitation for such traditional SBFL is that faulty code elements may be coincidentally executed by passed tests and elements executed by failed tests do not always have real impacts on the program failure. To bridge the gap between coverage and impact information, mutation-based fault localization (MBFL) [22, 38, 40, 63] has been proposed to transform program source code based on mutation testing [12] to check the impact of each code element on test outcomes. The basic idea of MBFL is that if one mutant incurs different failure outputs of failed tests before and after mutation, the corresponding code element of this mutant may have a high impact on program failures, and thus may be the buggy. MUSE [38] and Metallaxis [40] are two widely studied MBFL techniques targeting traditional application scenarios, while FIFL [63] is a MBFL technique specifically targeting evolving software systems. Compared with MBFL, unified debugging utilizes program repair information that aims to fix software bugs to *pass* more tests rather than mutation testing that was originally proposed to create new

artificial bugs to *fail* more tests; furthermore, unified debugging has also been shown to substantially outperform state-of-the-art MBFL [33, 34]. In the literature, researchers have also proposed various other fault localization techniques, including techniques based on program slicing [5, 44], development history [17], and information retrieval [65], as well as techniques for combining various dimensions of information via machine learning [21, 49, 60].

5.2 Automated Program Repair

Automated program repair (APR) aims to directly fix program bugs without human intervention. Given a buggy project, APR techniques utilize various strategies to automatically generate potential patches and then validate those patches to check their correctness, e.g., based on regression tests [10], static analysis [50], or formal specifications [52]. To date, test-driven APR has been extensively studied due to the wide adoption of testing in practice. A typical test-driven APR technique first applies off-the-shelf fault localization techniques (e.g., Ochiai [3] has been widely used for APR [10, 14, 55]) to pinpoint potential buggy locations for patching. Then, any patches that can pass all the originally failing and passing tests are called *plausible patches*, while plausible patches semantically equivalent to corresponding developer patches are called *correct patches* (which are the final outcome for APR). Depending on how the patches are generated, APR [10, 14, 31, 58] can be categorized into the following categories [11, 27]: (1) *heuristic-based APR*, which investigates possible code modifications for patching by iterating a search space, e.g., GenProg [53] uses genetic programming algorithm to search donor code from existing code for generating patches; (2) *constraint-based APR*, which typically transforms the APR problem into a satisfiability problem by constructing a repair constraint that the patches should satisfy, e.g., Nopol [59] leverages an SMT solver to solve the condition synthesis problem; (3) *template-based APR*, which performs APR via predefined fixing patterns, e.g., FixMiner [19] automatically mines bug-fix patterns from existing code repositories; (4) *learning-based APR*, which uses machine learning techniques to learn correct code locations/snippets from a training code corpus, e.g., Prophet [31] and ELIXIR [45].

Recently, ProFL [33] initializes the idea of unified debugging to investigate the effectiveness of APR for fault localization. The experimental results show that ProFL is able to boost/outperform state-of-the-art SBFL [3, 15, 23], MBFL [22, 38, 40, 63], and unsupervised/supervised learning based fault localization [21, 49, 64] using the recent PraPR APR system [10], and also extends the application scope of APR to all possible bugs. However, it is not clear how other state-of-the-art APR techniques contribute to unified debugging and how to further advance unified debugging, while this paper moves one step forward to that end, particularly with regards to assessing the impact of other APR tools in unified debugging.

6 CONCLUSION

In this paper, we have performed an extensive study of the impacts of different automated program repair systems on the recently proposed unified debugging approach [33, 34]. Our study results on the popular Defects4J benchmark suite have revealed various practical guidelines for further advancing unified debugging, including: (1) nearly all the studied 16 repair systems can contribute to unified debugging despite their varying repairing capabilities, (2) repair systems targeting multi-edit patches can bring noise and degrade performance for unified debugging, (3) repair systems with more executed/plausible patches tend to perform better for unified debugging, (4) unified debugging effectiveness does not exclusively rely on the availability of correct patches from automated repair. Based on our findings, we further proposed an advanced unified debugging technique, UniDebug++, which can localize over 20% more bugs within Top-1 than state-of-the-art unified debugging technique, ProFL [33, 34]. In the near future, we will work on *tentative program repair*, a new direction enabled by unified debugging to allow fault localization and program repair to boost each other for more powerful debugging, e.g., patch execution results from an initial set of repair systems can enable precise fault localization for applying more advanced repair systems for cost-effective repair.

ACKNOWLEDGEMENTS

This work was partially supported by National Science Foundation under Grant Nos. CCF-1763906 and CCF-1942430, and Alibaba.

REFERENCES

[1] 2020. Tricentis reports. https://www.tricentis.com/resources/software-fail-watch-5th-edition/
[2] 2020. Unified Debugging Website. https://github.com/ProdigyXable/UnifiedDebuggingStudy
[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46.
[4] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.
[5] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. IEEE, 143–151.
[6] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 177–188.
[7] CO Boulder. 2013. University of Cambridge Study: Failure to Adopt Reverse Debugging Costs Global Economy $41 Billion Annually. https://www.roguewave.com/company/news/2013/university-of-cambridge-reverse-debugging-study. Accessed: Jan. 8, 2019.
[8] Lingchao Chen and Lingming Zhang. 2020. Fast and Precise On-the-fly Patch Validation for All. *arXiv preprint arXiv:2007.11449* (2020).
[9] 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 '16)*. Association for Computing Machinery, 85–91.
[10] 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*. 19–30.
[11] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.
[12] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE TSE* 37, 5 (2011), 649–678.
[13] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 255–266.
[14] 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*. 298–309.
[15] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.
[16] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[17] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. 2007. Predicting faults from cached history. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 489–498.

[18] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 165–176.

[19] 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. *Empirical Software Engineering* (2020), 1–45.

[20] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[21] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 169–180.

[22] Xia Li and Lingming Zhang. 2017. Transforming Programs and Tests in Tandem for Fault Localization. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 92 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133916

[23] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. *ACM Sigplan Notices* 40, 6 (2005), 15–26.

[24] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 102–113.

[25] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *SANER*. 456–467.

[26] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.

[27] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé François D Assise Bissyande, 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 *42nd ACM/IEEE International Conference on Software Engineering (ICSE)*.

[28] 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*. 727–739.

[29] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 166–178.

[30] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 702–713.

[31] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 298–312.

[32] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 43–54.

[33] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Can Automated Program Repair Refine Fault Localization? *arXiv preprint arXiv:1910.01270* (2019).

[34] 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 *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 75–87.

[35] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 441–444.

[36] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In *SSBSE*. 65–86.

[37] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.

[38] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162.

[39] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 1–32.

[40] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[41] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.

[42] K Pearson. 1895. Notes on Regression and Inheritance in the Case of Two Parents Proceedings of the Royal Society of London, 58, 240-242.

[43] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 24–36.

[44] Manos Renieres and Steven P Reiss. 2003. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 30–39.

[45] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 648–659.

[46] Seemanta Saha et al. 2019. Harnessing evolution for multi-hunk program repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 13–24.

[47] Raul Santelices, James A Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight fault-localization using multiple coverage types. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 56–66.

[48] Undo Software. 2016. Increasing software development productivity with reversible debugging. https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf. Accessed: Jan. 21, 2019.

[49] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 273–283.

[50] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*. 151–162.

[51] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far are We?. In *the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)*.

[52] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*. 61–72.

[53] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 364–374.

[54] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2020. Historical Spectrum based Fault Localization. *IEEE Transactions on Software Engineering (TSE)* (2020).

[55] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 1–11.

[56] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[57] Mingyuan Wu, Lingming Zhang, Cong Liu, Shin Hwei Tan, and Yuqun Zhang. 2019. Automating cuda synchronization via program transformation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 748–759.

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

[59] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[60] Jifeng Xuan and Martin Monperrus. 2014. Learning to combine multiple ranking metrics for fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 191–200.

[61] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering* (2018).

[62] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 23–32.

[63] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting Mechanical Faults to Localize Developer Faults for Evolving Software. In *OOPSLA*. 765–784.

[64] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using PageRank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 261–272.

[65] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 14–24.