Fast and Precise On-the-fly Patch Validation for All

Lingchao Chen
The University of Texas at Dallas
lingchao.chen@utdallas.edu

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

ABSTRACT

Generate-and-validate (G&V) automated program repair (APR) techniques have been extensively studied during the past decade. Meanwhile, such techniques can be extremely time-consuming due to manipulation of the program code to fabricate a large number of patches and also repeated executions of tests on patches to identify potential fixes. PraPR, a recent G&V APR technique, reduces these costs by modifying program code directly at the level of compiled bytecode, and further performing *on-the-fly patching* by allowing multiple patches to be tested within the same JVM session. However, PraPR is limited due to its pattern-based, bytecode-level nature and it is basically unsound/imprecise as it assumes that patch executions do not change global JVM state and affect later patch executions on the same JVM session. Inspired by the PraPR work, we propose a unified patch validation framework, named UniAPR, which aims to speed up the patch validation for both bytecode and source-code APR via on-the-fly patching; furthermore, UniAPR addresses the imprecise patch validation issue by resetting the JVM global state via runtime bytecode transformation. We have implemented UniAPR as a fully automated Maven Plugin. We have also performed the first study of on-the-fly patch validation for state-of-the-art source-code-level APR. Our experiments show the first empirical evidence that vanilla on-the-fly patch validation can be imprecise/unsound; in contrast, our UniAPR framework can speed up state-of-the-art APR by over an order of magnitude without incurring any imprecision in patch validation, enabling all existing APR techniques to explore a larger search space to fix more bugs in the near future. Furthermore, UniAPR directly enables hybrid source and bytecode APR to fix substantially more bugs than all state-of-the-art APR techniques (under the same time limit) in the near future.

KEYWORDS

Program repair, Program transformation, Runtime optimization, JVM bytecode manipulation

1 INTRODUCTION

Software bugs are inevitable in modern software systems, costing trillions of dollars in financial loss and affecting billions of people [5]. Meanwhile, software debugging can be extremely challenging and costly, consuming over half of the software development time and resources [58]. Therefore, a large body of research efforts have been dedicated to automated debugging techniques [13, 43, 63]. Among the existing debugging techniques, automated program repair [15] (APR) techniques hold the promise of reducing debugging effort by suggesting likely patches for buggy programs with minimal human intervention, and have been extensively studied in the recent decade. Please refer to the recent surveys on APR for more details [13, 43].

Generate-and-validate (G&V) APR refers to a practical category of APR techniques that attempt to fix the bugs by first generating a pool of patches and then validating the patches via certain rules and/or checks [13]. A patch is said to be *plausible* if it passes all the checks. Ideally, we would apply formal verification [48] techniques to guarantee correctness of generated patches. However, in practice, formal specifications are often unavailable for real-world projects, thus making formal verification infeasible. In contrast, testing is the prevalent, economic methodology of getting more confidence about the quality of software [2]. Therefore, the vast majority of recent G&V APR techniques leverage developer tests as the criteria for checking correctness of the generated patches [13], i.e., *test-based* G&V APR.

Two main costs are associated with such test-based G&V APR techniques: (1) the cost of manipulating the program code to fabricate/generate a patch based on certain transformation rules; (2) repeated executions of all the developer tests to identify plausible patches for the bugs under fixing. Since the search space for APR is infinite and it is impossible to triage the elements of this search space due to theoretical limits, test-based G&V APR techniques usually lack clear guidance and act almost in a brute-force fashion: they usually generate a huge pool of patches to be validated and the larger the program the larger the set of patches to be generated and validated. This suggests that the speed of patch generation and validation plays a key role in scalability of the APR techniques, which is one of the most important challenges in designing practical APR techniques [9]. Therefore, apart from introducing new, more effective, transformation rules, some APR techniques have been proposed to mitigate the aforementioned costs. For example, JAID [6] uses mutation schema to fabricate meta-programs that bundle multiple patches in a single source file, while SketchFix [17] uses sketches [28] to achieve a similar effect. However, such techniques mainly aim to speed up the patch generation time, while patch validation time has been shown to be dominant during APR [42]. Most recently, PraPR [14] aims to reduce both patch generation and validation time – it reduces the cost of patch generation by modifying program code directly at the level of compiled JVM bytecode, and reduces the cost of patch validation by avoiding expensive process creation/initialization via reusing Java Virtual Machine (JVM) sessions across patches.

We have empirical evidence that shows the optimizations offered by state-of-the-art APR tools make patch generation and validation much faster than before. However, compared to other techniques, the speed-up due to PraPR is huge (e.g. it is 90+X faster than SketchFix). A careful analysis of the architecture of PraPR reveals that this remarkable speed-up is not just because of the way it generates patches, but is also because the tool is using the HotSwap technique [8] to validate all the generated patches on-the-fly on the same JVM instead of creating a separate JVM process for each patch. This turns out to be a dominant factor. Process creation overhead in systems like JVM is especially pronounced as virtually all the optimization tasks, as well as Just-In-Time (JIT) compilation [1], are done at runtime. Therefore, compared to natively executed programs, JVM-based programs are expected to take relatively longer time to warm up. Furthermore, during APR, a similar set of used bytecode files are loaded, linked, and initialized again and again for each patch. This suggests repeated executions of patches in separate processes (which is the dominant approach in mainstream APR techniques) for JVM-based based languages waste a significant amount of time that could be otherwise spent on applying more sophisticated patch generation rules or exploring more of the search space.

PraPR, although is effective, suffers from two major problems: (1) it is not flexible due to its bytecode-level nature; and (2) it is unsafe as it might report unsound/imprecise patch validation results. The first problem is best illustrated by inserting a factor in an arithmetic expression. For example, mutating `a*b+(a-b)` to `a*b+(a*c-b)` turns out to be a non-trivial program analysis task at the bytecode level. This is because Java compiler might reorder the bytecode instructions based on the priority of arithmetic operators or avoid repeated memory accesses when the variables `a` and `b` are declared `final` by doing simple optimization of loading the variables once and duplicating their values on the JVM stack. Either of these optimizations would make the such a program transformation hard to implement efficiently as locating the right place for inserting the instructions corresponding to the second multiplication is hard. Apparently this task would be trivial if we had modified the program at the level of source code by manipulating Abstract Syntax Trees (ASTs) of the programs. Also, it has been widely recognized as notoriously challenging to perform large-scale changes at the bytecode level, making the set of bugs fixable by PraPR rather limited. The second problem is that the global JVM state may be *polluted* by earlier patch executions, making later patch execution results unreliable. For example, some patches may modify some static fields, which are used by some later patches sharing the same JVM. Note that although the original PraPR work does not have such imprecision issue due to the specific limited types of patches supported (by bytecode APR), we do find instances of such imprecision in our study (detailed shown in Section 5.1.2).

Motivated by the strengths and weaknesses of PraPR, in this paper, we propose a unified test-based patch validation framework, named UniAPR, that reduces the cost of patch validation by avoiding unnecessary restarts of JVM for all existing bytecode or sourcode-level APR techniques. UniAPR achieves this by using a single JVM for patch validation, as much as possible, and depends on JVM's dynamic class redefinition feature (a.k.a. the HotSwap mechanism and Java Agent technology) to only reload the patched bytecode classes on-the-fly for each patch. Furthermore, it also addresses the imprecision problem of PraPR by isolating patch executions via resetting JVM states after each patch execution via runtime bytecode transformation. In this way, UniAPR not only substantially speeds up all state-of-the-art APR techniques at the source-code level (enabling them to explore larger search space to fix more bugs in the near future), but also provides a natural framework to enable *hybrid APR* to combine the strengths of various APR techniques at both the source-code and bytecode levels.

UniAPR has been implemented as a fully automated Maven [12] plugin, to which almost all existing state-of-the-art Java APR tools can be attached in the form of patch generation *add-ons*. We have constructed add-ons for representative APR tools from different APR families. Specifically, we have constructed add-ons for CapGen [62], SimFix [19], and ACS [65] that are modern representatives of template-/pattern-based [10, 26, 27], heuristic-based [3], and constraint-based [47, 66] techniques. With these techniques, we have conducted the first extensive study of on-the-fly validation of patches generated at the source code level. Our experiments show that UniAPR can speed up state-of-the-art APR systems (i.e., CapGen, SimFix, and ACS) by over an order of magnitude without incurring any imprecision in patch validation, enabling all existing APR techniques to explore a larger search space to fix more bugs in the near future.

We envision a future wherein all existing APR tools (like SimFix [19], CapGen [62], and ACS [65]) and major APR frameworks (like ASTOR [39] and Repairnator [45]) are leveraging this framework for patch validation. In this way, researchers will need only to focus on devising more effective algorithms for better exploring the patch search space, rather than spending time on developing their own components for patch validation, as we can have a unified, generic, and much faster framework for all. Furthermore, our UniAPR directly enables hybrid source and bytecode APR to fix substantially more bugs than all state-of-the-art APR techniques (under the same time limit) in the near future.

In summary, this paper makes the following contributions:

- **Framework.** We introduce the first unified on-the-fly patch validation framework, UniAPR, to speed up APR techniques for JVM-based languages at both the source and bytecode levels.
- **Technique.** We show the first empirical evidence that on-the-fly patch validation can be imprecise/unsound, and introduce a new technique to reset the JVM state right after each patch execution to address such issue.
- **Implementation.** We have implemented on-the-fly patch validation based on the JVM HotSwap mechanism and Java Agent technology [8], and implemented the JVM-reset technique based on the ASM bytecode manipulation framework [49]; the overall UniAPR tool has been implemented as a practical Maven plugin, and can accept different APR techniques as patch generation add-ons to reduce their patch validation cost.
- **Study.** We conduct a large-scale study of the effectiveness of UniAPR on its interaction with state-of-the-art APR tools from three different APR families, demonstrating that UniAPR can speed up state-of-the-art APR by over an order of magnitude (with precise validation results), and can enable hybrid APR to directly combine the strengths of different APR tools.

The rest of this paper is organized as follows. We introduce the necessary background information in Section 2. In Section 3, we introduce the details of the proposed UniAPR technique. Next, we present our experimental setup and result analysis in Section 4 and Section 5. Finally we discuss related work in Section 6 before we conclude the paper in Section 7.

2 BACKGROUND

In this section, we set the scene by introducing some background for better understanding this work. More specifically, we first talk about the current status of automated program repair (Section 2.1); then, we talk about Java Agent and HotSwap, on which our UniAPR work is built on (Section 2.2).

2.1 Automatic Program Repair

Automatic program repair (APR) aims to suggest likely patches for buggy programs to reduce the manual effort during debugging. Based on the actions taken for fixing a bug, state-of-the-art APR techniques can be divided into: (1) techniques that monitor the execution of a system to find deviations from certain specifications, and then heal the system by modifying its runtime state in case of any abnormal behavior [35, 54]; (2) generate-and-validate (G&V) techniques that attempt to fix the bug by first generating a pool of patches and validating the patches via certain rules and/or checks [14, 19, 27, 39, 47, 62]. Generated patches that can pass all the tests/checks are called *plausible* patches. However, not all plausible patches are the patches that the developers want. Therefore, these plausible patches are further manually checked by the developers to find the final *correct* patches (i.e., the patches semantically equivalent to developer patches). Among these, G&V techniques, especially those based on tests, have gained popularity as testing is the dominant way for detecting bugs in practice, while very few real-world systems are based on rigorous and up-to-date formal specifications.

In recent years, a large number of APR-related research papers have been published in different software engineering and programming languages conferences and journals [13, 43]. These papers either introduce new techniques for generating high quality patches or study different aspects of already introduced techniques. Recently, Ghanbari et al. [14] showed for the first time that the sheer speed of patch generation and validation gives an otherwise simplistic template-/pattern-based technique like PraPR a discrete advantage, in that it allows the tool to explore more of the entire search space in an affordable amount of time. The explored search space is more likely to contain plausible and hence correct patches.

The speed of patch generation and validation in PraPR comes from on-the-fly patch generation which is possible due to two features: (1) bytecode-level patch generation; (2) on-the-fly patch validation based on dynamic class redefinition. PraPR generates patches by directly modifying programs at the level of compiled JVM bytecode [31]. This allows the tool to bypass expensive tasks of parsing and modifying ASTs, as well as type checking and compilation (which itself is preceded by several undoubtedly expensive accesses to secondary memory). Furthermore, the tool also avoids starting a new JVM session for validating each and every one of the patches. Instead, it creates a single JVM process and leverages the Java Agent technology and HotSwap mechanism offered by JVM to reload only the patched bytecode files for each patch without restarting the JVM. In this way, PraPR not only can avoid reloading (also including linking and initializing) all used classes for each patch (i.e., only the patched bytecode file(s) needs to be reloaded for each patch), but also can avoid the unnecessary JVM warm-up time (i.e., the accumulated JVM profiling information cross patches enables more and more code to be JIT-optimized and the already JIT-optimized code can also be shared across patches). In this paper, we further generalize the PraPR on-the-fly patching to all existing APR systems (whereas prior work only applied it for bytecode APR [14]), and also address the unsoundness/imprecision issues for such optimization.

2.2 Java Agent and HotSwap

A Java Agent [8] is a compiled Java program (in the form of a JAR file) that runs alongside of the JVM in order to intercept applications running on the JVM and modify their bytecode. Java Agent utilizes the instrumentation API [8] provided by Java Development Kit (JDK) to modify existing bytecode that is loaded in the JVM. In general, developers can both (1) *statically* load a Java Agent using `-javaagent` parameter at JVM startup, and (2) *dynamically* load a Java Agent into an existing running JVM using the Java Attach API. For example, to load it statically, the manifest of the JAR file containing Java Agent must contain a field `Premain-Class` to specify the name of the class defining `premain` method. Such a class is usually referred to as an `Agent class`. Agent class is loaded before any class in the application class is loaded and the `premain` method is called before the main method of the application class is invoked. The method `premain` usually has the following signature:

```
public static void premain(String agentArgs,
              Instrumentation inst)
```

The second parameter is an object of type `Instrumentation` created by the JVM that allows the Java Agent to analyze or modify the classes loaded by the JVM (or those that are already loaded) before executing them. Specifically, the method `redefineClasses` of `Instrumentation`, given a *class definition* (which is essentially a class name paired with its "new" bytecode content), even enables dynamically updating the definition of the specified class, i.e., directly replacing certain bytecode file(s) with the new one(s) during JVM runtime. This is typically denoted as the JVM HotSwap mechanism. It is worth mentioning that almost all modern implementations of JVM (especially, so-called HotSpot JVMs) have these features implemented in them.

By obtaining `Instrumentation` object, we have a powerful tool using which we can implement a HotSwap Agent. As the name suggests, HotSwap Agent is a Java Agent and is intended to be executed alongside the patch validation process to dynamically reload patched bytecode file(s) for each patch. In order to test a generated patch during APR, we can pass the patched bytecode file(s) of the patch to the agent, which *swaps* it with the original bytecode file(s) of the corresponding class(es). Then, we can continue to run tests which results in executing the patched class(es), i.e., validating the corresponding patch. Note that subsequent requests to HotSwap Agent for later patch executions on the same JVM are always preceded by replacing previously patched class(es) with its original version. In this way, we can validate all patches (no matter generated by source-code or bytecode APR) on-the-fly sharing the same JVM for much faster patch validation.

3 APPROACH

In this section, we first talk about the overall approach for our UniAPR system (Section 3.1). Then, we will talk about our detailed

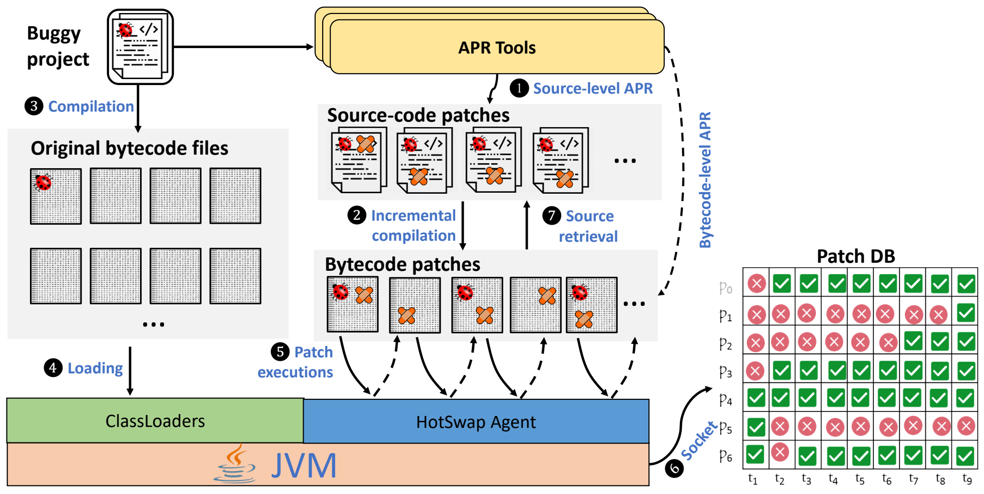


Figure 1: An architectural overview of UniAPR and its workflow

design for *fast* patch validation via on-the-fly patching (Section 3.2) as well as *precise* patch validation via JVM reset (Section 3.3).

3.1 Overview

Figure 1 depicts an the overall flow of our UniAPR framework. According to the figure, given a buggy project, UniAPR first leverages any of the existing APR tools (integrated as UniAPR add-ons) to generate source-code level patches (marked with ❶). Then, UniAPR performs incremental compilation to compile the patched source file(s) by each patch into bytecode file(s) (marked with ❷). Note that, UniAPR is a unified framework and can also directly take the bytecode patches generated by the PraPR (and future) bytecode APR technique (marked with the dashed line directly connecting APR tools into bytecode patches). In this way, UniAPR has a pool of bytecode patches for patch validation.

During the actual patch validation, UniAPR first compiles the entire buggy project into bytecode files (i.e., `.class` files), and then loads all the bytecode files into the JVM through JVM class loaders (marked with ❸ and ❹ in the figure). Note that these two steps are exactly the same as executing the original tests for the buggy project. Since all the bytecode files for the original project are loaded within the JVM, for validating each patch, UniAPR only reloads the patched bytecode file(s) by that particular patch via the Java Agent technology and HotSpot mechanism, marked with ❺ (as the other unpatched bytecode files are already within the JVM). Then, the test driver can be triggered to execute the tests to validate against the patch without restarting a new JVM. After all tests are done for this patch execution, UniAPR will replace the patched bytecode file(s) with the original one(s) to revert to the original version. Furthermore, UniAPR also resets the global JVM states to prepare a clean JVM environment for the next patch execution (marked with the short dashed lines). The same process is repeated for each patch. Finally, the patch validation results will be stored into the patch execution database via socket connections (marked with ❻). Note that for any plausible patch that can pass all the tests, UniAPR will directly retrieve the original source-level patch for manual inspection (marked with ❼) in case the patch was generated by source-level APR.

We have already constructed add-ons for three different APR tool representing three different families of APR techniques. These add-ons include CapGen [62] (representing pattern/template-based APR techniques), SimFix [19] (representing heuristic-based techniques), and ACS [65] (representing constraint-based techniques). Users of UniAPR can easily build a new patch generation add-on by implementing the interface `PatchGenerationPlugin` provided by the framework. For already implemented APR tools, this can be easily done by changing their source code so that the tools abandon validation of patches after generating and compiling them.

3.2 Fast Patch Validation via On-the-fly Patching

Algorithm 1 is a simplified description of the steps that *vanilla* UniAPR (without JVM-reset) takes in order to validate candidates patches on-the-fly. The algorithm takes as inputs the original buggy program $\mathcal{P}$, its test suite $\mathcal{T}$, and the set of candidate patches $\mathbb{P}$ generated by any APR technique. The output is a map, $\mathcal{R}$, that maps each patch into its corresponding execution result. The overall UniAPR algorithm is rather simple. UniAPR first initializes all patch execution results as unknown (Line 2). Then, UniAPR gets into the loop body and obtains the set of patches still with unknown execution results (Line 4). If there is no such patches, the algorithm simply returns since all the patches have been validated. Otherwise, it means this is the first iteration or the earlier JVM process gets terminated abnormally (e.g., due to timeout or JVM crash). In either case, UniAPR will create a new JVM process (Line 7) and start to evaluate the remaining patches in this new JVM (Line 8).

We next talk about the detailed `validate` function, which takes the remaining patches, the original test suite, and a new JVM as input. For each remaining patch $\mathcal{P}'$, the function first obtains the patched class name(s) $C_{patched}$ and patched bytecode file(s) $\mathcal{F}_{patched}$ within $\mathcal{P}'$ (Lines 11 and 12). Then, the function leverages our HotSwap Agent to replace the bytecode file(s) under the same class name(s) as $C_{patched}$ with the patched bytecode file(s)

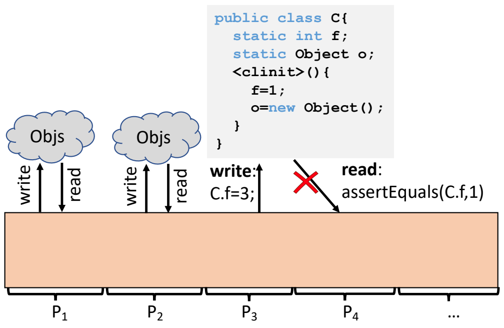
Algorithm 1: Vanilla on-the-fly patch validation in UniAPR

```
   Input: Original buggy program $\mathcal{P}$, test suite $\mathcal{T}$, and set of candidate patches $\mathbb{P}$
   Output: Validation status $\mathcal{R} : \mathbb{P} \rightarrow$ {PLAUSIBLE, NON − PLAUSIBLE, ERROR}
1  begin
2      $\mathcal{R} \leftarrow \mathbb{P} \times$ {UNKNOWN} ; // initialize the result function
3      while True do
4          $\mathbb{P}_{left} \leftarrow \{\mathcal{P}' \mid \mathcal{P}' \in \mathbb{P} \wedge \mathcal{R}(\mathcal{P}') =$ UNKNOWN$\}$// get all the left
             patches not yet validated
5          if $\mathbb{P}_{left} = \emptyset$ then
6              return $\mathcal{R}$ // return if there is no left patches
7          $\mathcal{JVM} \leftarrow$ createJVMProcess()// create a new JVM
8          validate($\mathbb{P}_{left}$, $\mathcal{T}$, $\mathcal{JVM}$)) // validate the left patches on the
             new JVM
9  function validate($\mathbb{P}_{left}$, $\mathcal{T}$, $\mathcal{JVM}$):
10     for $\mathcal{P}'$ in $\mathbb{P}_{left}$ do
11         $\mathcal{C}_{patched} \leftarrow$ patchedClassNames($\mathcal{P}'$)
12         $\mathcal{F}_{patched} \leftarrow$ patchedBytecodeFiles($\mathcal{P}'$)
13         $\mathcal{F}_{orig} \leftarrow$ HotSwapAgent.swap($\mathcal{JVM}$, $\mathcal{C}_{\text{patched}}$, $\mathcal{F}_{patched}$) // Swap
             in the patched bytecode files
14         for $t$ in $\mathcal{T}$ do
15             try:
16                 if run($\mathcal{JVM}$, $t$) = FAILING then
17                     $status \leftarrow$ NON − PLAUSIBLE
18                 else
19                     $status \leftarrow$ PLAUSIBLE
20             catch TimeOutException, MemoryError:
21                 $status \leftarrow$ ERROR
22             $\mathcal{R} \leftarrow \mathcal{R} \cup \{\mathcal{P}' \rightarrow status\}$
23             if $status$ = NON-PLAUSIBLE then
24                 break // continue with the next patch when current one
                     is falsified
25             if $status$ = ERROR then
26                 return // restart a new JVM when this current one timed
                     out or crashed
27         HotSwapAgent.swap($\mathcal{JVM}$, $\mathcal{C}_{patched}$, $\mathcal{F}_{orig}$)// Swap back the
             original bytecode files
```

$\mathcal{F}_{patched}$; it also stores the replaced bytecode file(s) as $\mathcal{F}_{orig}$ to recover it later (Line 13). Note that our implementation will explicitly load the corresponding class(es) to patch (e.g., via `Class.forName()`) if they are not yet available before swapping. In this way, the function can now execute the tests within this JVM to validate the current patch since the patched bytecode file(s) has already been loaded (Lines 14-26). If the execution for a test finishes normally, its status will be marked as `Plausible` or `Non-Plausible` (Lines 16-19); otherwise, the status will be marked as `Error`, e.g., due to timeout or JVM crash (Lines 20-21). Then, $\mathcal{P}'$'s status will be updated in $\mathcal{R}$ (Line 22). If the current status is `Non-Plausible`, the function will abort the remaining test executions for the current patch since it has been falsified, and move on to the next patch (Line 24); if the current status is `Error`, the function will return to the main algorithm (Line 26), which will restart the JVM. When the validation for the current patch finishes without the `Error` status, the function will also recover the patched bytecode file(s) into the original one(s) to facilitate the next patch validation (Line 27).

3.3 Precise Patch Validation via JVM Reset

3.3.1 Limitations for vanilla on-the-fly patch validation. The vanilla on-the-fly patch validation presented in Section 3.2 works for most patches of most buggy projects. The basic process can be illustrated via Figure 2. In the figure, each patch (e.g., from $\mathcal{P}_1$ to $\mathcal{P}_4$) gets executed sequentially on the same JVM. It would be okay if every patch accesses and modifies the objects created by itself, e.g., $\mathcal{P}_1$ and $\mathcal{P}_2$ will not affect each other and the vanilla on-the-fly patch validation results for $\mathcal{P}_1$ and $\mathcal{P}_2$ will be the same as the ground-truth patch validation results. However, it will be problematic if one patch writes to some global space (e.g., static fields) and later on some other patch(es) reads from that global space. In this way, earlier patch executions will affect later patch executions, and we call such global space *pollution sites*. To illustrate, in Figure 2, $\mathcal{P}_3$ write to some static field `C.f`, which is later on accessed by $\mathcal{P}_4$. Due to the existence of such pollution site, the execution results for $\mathcal{P}_4$ will no longer be precise, e.g., its assertion will now fail since `C.f` is no longer 1, although it may be a correct patch.



```
public class C{
  static int f;
  static Object o;
  <clinit>(){
    f=1;
    o=new Object();
  }
}
```

Figure 2: Imprecision under vanilla on-the-fly patch validation

```
// org.joda.time.TestYearMonthDay_Constructors.java
public class TestYearMonthDay_Constructors extends TestCase {
    private static final DateTimeZone PARIS = DateTimeZone.forID("Europe/Paris");

    private static final DateTimeZone LONDON = DateTimeZone.forID("Europe/London
           ");
    private static final Chronology GREGORIAN_PARIS =
           GregorianChronology.getInstance(PARIS);
    ...
```

Figure 3: Static field dependency

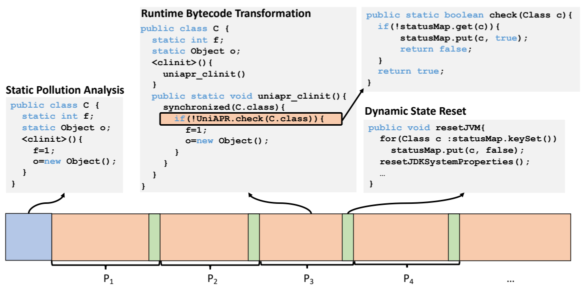
3.3.2 Technical challenges. We observe that accesses to static class fields are the main reason leading to imprecise on-the-fly patch validation. Ideally, we only need to reset the values for the static fields that may serve as pollution sites right after each patch execution. In this way, we can always have a clean JVM state to perform patch execution without restarting the JVM for each patch. However, it turns out to be a rather challenging task:

First, we cannot simply reset the static fields that can serve as pollution sites. The reason is that some static fields are `final` and cannot be reset directly. Furthermore, static fields may also be data-dependent on each other; thus, we have to carefully maintain their original ordering, since otherwise the program semantics may be changed. For example, shown in Figure 3, `final` field `GREGORIAN_PARIS` is data-dependent on another `final` field, `PARIS` under the same class within project Joda-Time [20] from the widely studied Defects4J dataset [22]. The easiest way to keep such ordering and reset `final` fields is to simply re-invoke the original class initializer for the enclosing class. However, according to the JVM specification, only JVM can invoke such static class initializers.

Second, simply invoking the class initializers for all classes with pollution sites may not work. A naive way to reset the pollution sites is to simply trace the classes with pollution sites executed

```
// org.joda.time.TestDateTime_Basics.java
public class TestDateTime_Basics extends TestCase {
    private static final ISOChronology ISO_UTC = ISOChronology.getInstanceUTC();
    ...
// org.joda.time.chrono.ISOChronology.java
public final class ISOChronology extends AssembledChronology {
    private static final ISOChronology[] cFastCache;
    static {
        cFastCache = new ISOChronology[FAST_CACHE_SIZE];
        INSTANCE_UTC = new ISOChronology(GregorianChronology.getInstanceUTC());
        cCache.put(DateTimeZone.UTC, INSTANCE_UTC);
    }
    ...
```

Figure 4: Static initializer dependency



```
public class C {
  static int f;
  static Object o;
  <clinit>(){
    f=1;
    o=new Object();
  }
}
```

```
public class C {
  static int f;
  static Object o;
  <clinit>(){
    uniapr_clinit()
  }
  public static void uniapr_clinit(){
    synchronized(C.class){
      if(!UniAPR.check(C.class)){
        f=1;
        o=new Object();
      }
    }
  }
}
```

```
public static boolean check(Class c){
  if(!statusMap.get(c)){
      statusMap.put(c, true);
      return false;
  }
  return true;
}
```

```
public void resetJVM{
  for(Class c :statusMap.keySet())
    statusMap.put(c, false);
  resetJDKSystemProperties();
  …
}
```

Figure 5: On-the-fly patch validation via JVM reset

during each patch execution; then, we can simply force JVM to invoke the class initializers for all those classes after each patch execution. However, it can bring side effects in practice because the class initializers may also depend on each other. For example, shown in Figure 4, within Joda-Time, the static initializer of class `TestDateTime_Basics` depends on the static initializer of `ISOChronology`. If `TestDateTime_Basics` is reinitialized earlier than `ISOChronology`, then field `ISO_UTC` will no longer be matched with the newest `ISOChronology` state. Therefore, we have to reinitialize all such classes following their original ordering if they had been executed on a new JVM.

Third, based on the above analysis, we basically have two choices to implement such system: (1) customizing the underlying JVM implementation, and (2) simulating the JVM customizations at the application level. Although it would be easier to directly customize the underlying JVM implementation, the system implementation will not be applicable for other stock JVM implementations. That said, we are only left with way of simulating the JVM customizations at the application level.

3.3.3 JVM reset via bytecode transformation. We now present our detailed approach for resetting JVM at the the application level. Inspired by prior work on speeding up traditional regression testing [4], we perform runtime bytecode transformation to simulate JVM class initializations. The overall approach is illustrated in Figure 5. We next present the detailed three phases as follows.

Static Pollution Analysis. Before all the patch executions, our approach performs lightweight static analysis to identify all the pollution sites within the bytecode files of all classes for the project under repair, including all the application code and 3rd-party library code. Note that we do not have to analyze the JDK library code since JDK usually provides public APIs to reset the pollution sites within the JDK, e.g., `System.setProperties(null)` can be used to reset any prior system properties and `System.setSecurityManager(null)` can be leveraged to reset prior security manager. The analysis basically returns all classes with non-`final` `static` fields or `final` `static` fields with non-primitive types (their actual object states in the heap can be changed although their actual references cannot be changed), since the states for all such static fields can be changed across patches. Shown in Figure 5, the blue block denotes our static analysis, and class `C` is identified since it has static fields `f` and `o` that can be mutated during patch execution.

C1	T is a class and an instance of T is created
C2	T is a class and a static method declared by T is invoked.
C3	A static field declared by T is assigned
C4	A static field declared by T is used and the field is not a constant variable
C5	T is a top level class, and an assert statement lexically nested within T is executed

Table 1: Class initialization conditions

Runtime Bytecode Transformation. According to Java Language Specification (JSL) [50], static class initializers get invoked when any of the five conditions shown in Table 1 gets satisfied. Therefore, the ideal way to reinitialize the classes with pollution sites is to simply follow the JSL design. To this end, we perform runtime bytecode transformation to add class initializations right before any instance that falls in to the five conditions shown in Table 1. Note that our actual implementation also handles the non-conventional Reflection-based accesses to all potential pollution sites within classes. Since JVM does not allow class initialization at the application level, following prior work [4] on speeding up normal test executions during regression testing, we rename the original class initializers (i.e., `<clinit>()`) to be invoked into another customizable name (say `uniapr_clinit()`). Meanwhile, we still keep the original `<clinit>()` initializers since JVM needs that for the initial invocation; however, now `<clinit>()` initializers do not need to have any content except an invocation to the new `uniapr_clinit()`. Note that we also remove potential `final` modifiers for pollution sites during bytecode transformation to enable reinitializations of `final` non-primitive static fields. Since this is done at the bytecode level after compilation, the original compiler will still ensure that such `final` fields cannot be changed during the actual compilation phase.

Now, we will be able to reinitialize classes via invoking the corresponding `uniapr_clinit()` methods. However, JVM only initializes the same class once within the same JVM, while now `uniapr_clinit()` will be executed for each instance satisfying the five conditions in Table 1. Therefore, we need to add the dynamic check to ensure that each class only get (re)initialized once for each patch execution. Shown in Figure 5, the pink blocks denote the different patch executions. During each patch execution, the classes with pollution sites will be transformed at runtime, e.g., class `C` will be transformed into the code block connected with the $\mathcal{P}_3$ patch execution. Note that the pink code block denotes the dynamic check to ensure that `C` is only initialized once for each patch[1]. The pseudo code for the dynamic check is shown in the top-left of the figure. We can see that the check maintains a concurrent `HashMap` for the classes with pollution sites and their status (`true` means the corresponding class has been reinitialized). The entire

[1]Note that this pseudo code is just for illustration, and our actual implementation manipulates arrays for faster and safe tracking/check.

Sub.	Name	#Bugs	#Tests	LoC
Chart	JFreeChart	26	2,205	96K
Time	Joda-Time	27	4,130	28K
Lang	Apache commons-lang	65	2,245	22K
Math	Apache commons-math	106	3,602	85K
Closure	Google Closure compiler	133	7,927	90K
Total		357	20,109	321K

Table 2: Defects4J V1.0.0 statistics

initialization is also synchronized based on the `class` object to handle concurrent accesses to class initializers; in fact, JVM also leverages a similar mechanism to avoid class reinitializations due to concurrency (despite implementing that at a different level). In this way, when the first request for initializing class `C` arrives, all the other requests will be blocked. If the class has not been initialized, then only the current access will get the return value of `false` to reinitialize `C`, while all other other requests will get the return value of `true` and skip the static class initialization. Furthermore, the static class initializers get invoked following the same order as if they were invoked in a new JVM.

Dynamic State Reset. After each patch execution, our approach will reset the state for the classes within the status `HashMap`. In this way, during the next patch execution, all the used classes within the `HashMap` will be reinitialized (following the check in Figure 5). Note that besides the application and 3rd-party classes, the JDK classes themselves may also have pollution sites. Luckily, JDK provides such common APIs to reset such pollution sites without the actual bytecode transformation. In this way, our implementation also invokes such APIs to reset potential JDK pollution sites. Please also note that our system provides a public interface for the users to customize the reset content for different projects under repair. For example, some projects may require preparing specific external resources for each patch execution, which can be easily added to our public interface. In Figure 5, the green strips denote the dynamic state reset, and the example reset code connected to $\mathcal{P}_3$ simply resets the status for all classes within the status map as `false` and also resets potential JDK pollution sites within classes.

4 EXPERIMENTAL SETUP

4.1 Research Questions

To thoroughly evaluate our UniAPR framework, in this study, we aim to investigate the following research questions:

- RQ1: How does vanilla on-the-fly patch validation perform for automated program repair?
- RQ2: How does on-the-fly patch validation with jvm-reset perform for automated program repair?

For both RQs, we study both the *effectiveness* of UniAPR in reducing the patch validation cost, and the *precision* of UniAPR in producing precise patch validation results.

4.2 Benchmarks

The benchmark suites are important for evaluating APR techniques. We choose the Defects4J (V1.0.0) benchmark suite [22], since it contains hundreds of real-world bugs from real-world systems, and has become the most widely studied dataset for program repair or even software debugging in general [14, 19, 29, 62]. Table 2 presents the statistics for the Defects4J dataset. Column "Sub." presents the project IDs within Defects4J, while Column "Name" presents the actual project names. Column "#Bugs" presents the number of bugs collected from real-world software development for each project, while Columns "#Tests" and "LoC" present the number of tests (i.e., JUnit test methods) and the lines of code for the `HEAD` buggy version of each project.

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

Table 3: Available Java APR tools for Defects4J

```
<plugin>
  <groupId>anonymized</groupId>
  <artifactId>uniapr-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
</plugin>
```

Figure 6: UniAPR POM configuration

4.3 Studied Repair Tools

Being a well-developed field, APR offers us a cornucopia of choices to select from. According to a recent study [33], there are 31 APR tools targeting Java programs considering two popular sources of information to identify Java APR tools: the community-led program-repair.org website and the living review of APR by Monperrus [44]. 17 of those Java APR tools are found to be publicly available and applicable to the widely used Defects4J benchmark suite (without additional manually collected information, e.g., potential bug locations) as of July 2019. Note that all such tools are source-level APR, since the only bytecode-level APR tool PraPR was only available after July 2019. Table 3 presents all such existing Java-based APR tools, which can be categorized into three main categories according to prior work [33]: heuristic-based [19, 27, 34], constraint-based [11, 66], and template-based [32, 62] repair techniques. In this work, we aims to speed up all existing source-level APR techniques via on-the-fly patch validation. Therefore, we select one representative APR tool from each of the three categories for our evaluation to demonstrate the general applicability of our UniAPR framework. All the three considered APR tools, i.e., ACS [65], SimFix [19], and CapGen [62] are highlighted in bold font in the table. For each of the selected tools, we evaluate them on all the bugs that have been reported as fixed (with correct patches) by their original papers to evaluate: (1) UniAPR effectiveness, i.e., how much speedup UniAPR can achieve on those tools, and (2) UniAPR precision, i.e., whether the previously fixed bugs are still fixed when running with UniAPR.

4.4 Implementation

UniAPR has been implemented as a fully automated Maven Plugin, on which one can easily integrate any patch generation add-ons. The current implementation involves over 10K lines of Java code. As a Maven plugin, the users simply need to add the necessary plugin information into the POM file. The plugin information can be as simple as shown in Figure 6. In this way, once the users fire command `mvn [anonymized-groupId]:prf-plugin:validate`, the plugin will automatically obtain all the necessary information for patch validation. It will automatically obtain the test code, source code, and 3-rd party libraries from the underlying POM file for the actual test execution. Furthermore, it will automatically load all the patches from the default `patches-pool` directory (note that the patch directory name and patch can be configured through POM as well) created by the APR add-ons for patch validation. The APR add-ons are constructed by modifying the behavior of the studied APR tools (either through direct modification of the source code or via inheritance and/or decoration when the source code is not available) to not to perform patch validation after generating and/or compiling the patches. UniAPR assumes the patch directory generated by the APR add-ons to include all available patches represented by their patched bytecode files. Note that, each patch may involve more than one patched bytecode file, e.g., some APR tools (such as SimFix [19]) can fix bugs with multiple edits.

During patch validation, our system marks the status of all the patches to `UNKNOWN`. It forks a JVM and passes all the information about the test suites and the subject programs to the child process. The process runs tests on each patch and reporting their status. Note that following the common practice in APR and the original setting of the studied APR tools [19, 62, 65], the process always runs the originally failing tests earlier than the originally passing tests for each patch. The reason is that the originally failing tests have a high probability to fail again and can falsify non-plausible patches faster. The execution results can be either of the following: `PLAUSIBLE` if the patch passes all the tests, `NON-PLAUSIBLE` if the the patch fails to pass a test, `TIME_OUT` if the patch times out on some test, `MEMORY_ERROR` if the patch runs out of heap space, and `UNKNOWN_ERROR` if testing the patch makes the child JVM to crash. We use TCP Socket Connections as a means of communication between processes. UniAPR repeats this process of forking and receiving report results until all the patches are executed. It is worth noting that it is very well possible to fork two or more processes to take maximum advantage of today's powerful computers' potentials. However, for a fair comparison with existing work, we always ensure that only one JVM is running patch validation at any given time stamp.

4.5 Experimental Setup

For each of the studied APR tools, we perform the following experiments on all the bugs that have been reported as fixed in their original papers. First, we execute the original APR tools to trace their original patch validation time and detailed repair results (e.g., the number of patches executed and plausible patches produced). Next, we modify the studied tools and make them conform to UniAPR add-on interfaces, i.e., dumping all the generated patches into the patch directory format required by UniAPR. Then, we launch our UniAPR to validate all the patches generated by each of the studied APR tools, and trace the new patch validation time and detailed repair results.

To evaluate our UniAPR system, we include the following metrics: (1) the speedup compared with the original patch validation time, measuring the effectiveness of UniAPR, and (2) the repair results compared with the original patch validation, measuring the precision of our patch validation (i.e., checking whether UniAPR fails to fix any bugs that can be fixed via traditional patch validation).

All our experimentation is done on a Dell workstation with Intel Xeon CPU E5-2697 v4@2.30GHz and 98GB RAM, running Ubuntu 16.04.4 LTS and Oracle Java 64-Bit Server version 1.7.0_80.

5 RESULT ANALYSIS

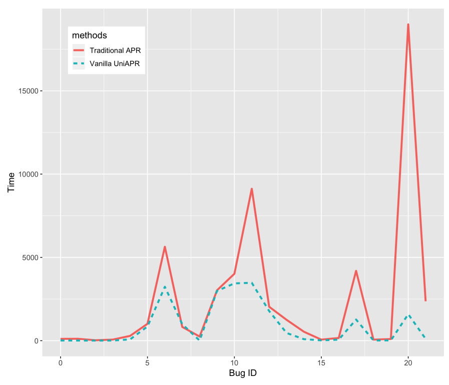
In this section we present the detailed result analysis for the research questions outlined in Section 4.1.

5.1 RQ1: Results for Vanilla On-the-fly Patch Validation

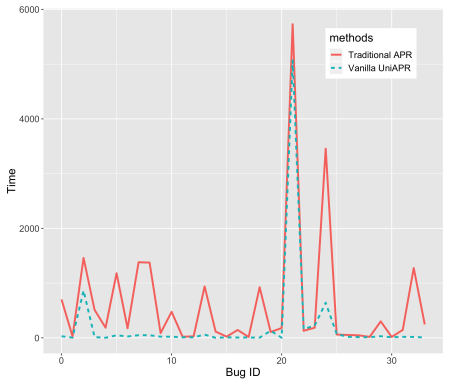
5.1.1 Effectiveness. For answering this RQ, we executed vanilla UniAPR (without JVM-reset) that is configured to use the add-on corresponding to each studied APR tool. Please note that we measure the original patch validation time by each original APR tool, and compare that against the patch validation time using vanilla UniAPR. The only exception is for CapGen: we observed that CapGen performs much faster even than vanilla UniAPR for some cases; digging into the decompiled CapGen code (the CapGen source code is not available), we realized that the CapGen tool excluded some (expensive) tests for certain bugs (confirmed by the authors). Therefore, to enable a fair comparison, for CapGen, we build a variant for our UniAPR framework that simply restarts a new JVM for each patch. The main experimental results are presented in Figure 7. In each sub-figure, the horizontal axis presents all the bugs that have been reported to be fixed by each studied tool, while the vertical axis presents the time cost (s); the solid and dashed lines then present the time cost for traditional patch validation and our vanilla UniAPR, respectively. From the figure, we can have the following observations:

First, for all the studied APR tools, vanilla UniAPR can substantially speed up the existing patch validation component for all state-of-the-art APR tools. For example, when running CapGen on Math-80, the traditional patch validation costs 18,991s while on-the-fly patch validation via vanilla UniAPR takes only 1,590s to produce the same patch validation results, i.e., 11.9X speedup; when running SimFix on Closure-62, the traditional patch validation costs 1,381s, while on-the-fly patch validation via vanilla UniAPR takes only 48s to produce the same patch validation results, i.e., 28.8X speedup; when running ACS on Math-82, the traditional patch validation costs 127s, while on-the-fly patch validation via vanilla UniAPR takes less than 9s to produce the same patch validation results, i.e., 14.3X speedup. To the best of our knowledge, this is the first study demonstrating that on-the-fly patch validation can also substantially speed up the existing state-of-the-art source-level APR techniques.

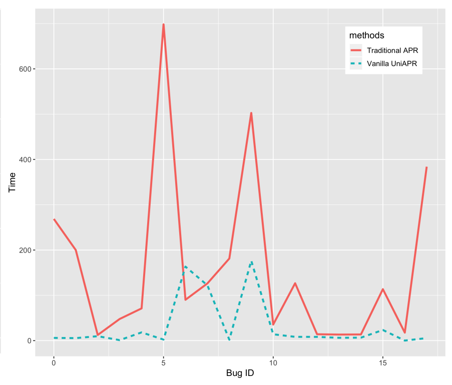
Second, while we observe clear speedups for the vast majority of the bugs, the achieved speedups vary a lot for all the studied APR tools on all the studied bugs. The reason is that the speedups are impacted by many different factors, such as the number of patches executed, the number of bytecode files loaded for each patch execution, the individual test execution time, and so on. For example, we observe that UniAPR even slows down the patch validation for ACS slightly on one bug (i.e., for 1min). Looking into the specific bug (i.e., Math-3), we find that ACS only produces one patch for



(a) CapGen



(b) SimFix



(c) ACS

Figure 7: Speedup achieved by vanilla UniAPR

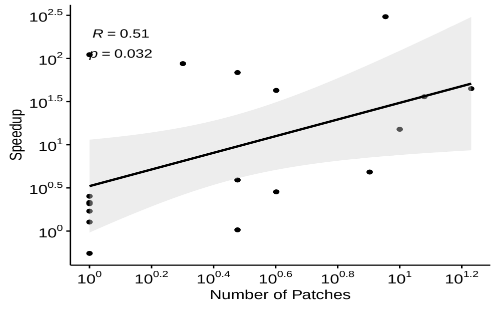


Figure 8: Correlation between patch number and speedup

Tool	# All	# Mismatch	Ratio (%)
CapGen	22	3	13.64%
SimFix	34	1	2.94%
ACS	18	0	0.00%
All	74	4	5.41%

Table 4: Inconsistent fixing results

that bug, and there is no JVM sharing optimization opportunity for UniAPR on-the-fly patch validation. To further confirm our finding, we perform the Pearson Correlation Coefficient analysis [53] (at the significance level of 0.05) between the number of patches for each studied bug and its corresponding speedup for ACS. Shown in Figure 8, the horizontal axis denotes the number of patches, while the vertical axis denotes the speedup achieved; each data point represents one studied bug for ACS. From this figure, we can observe that UniAPR tends to achieve significantly larger speedups for bugs with more patches (at the significance level of 0.05), demonstrating that UniAPR tends to achieve larger speedups for larger systems with more patches.

Finding-1: This study demonstrates for the first time that vanilla on-the-fly patch validation can also substantially speed up the existing state-of-the-art source-level APR techniques; furthermore, UniAPR tends to perform better for systems/bugs with more patches, demonstrating the scalability of on-the-fly patch validation.

5.1.2 Precision. We further study the number of bugs that vanilla UniAPR does not produce the same repair results as the traditional patch validation (that restarts a new JVM for each patch). Table 4 presents the summarized results for all the studied APR tools on all their fixable bugs. In this table, Column "Tool" presents the studied APR tools, Column "# All" presents the number of all studied fixable bugs for each APR tool, Column "# Mismatch" presents the number of bugs that vanilla UniAPR has inconsistent fixing results with the original APR tool, and Column "Ratio (%)" presents the ratio of bugs with inconsistent results. From this table, we can observe that vanilla UniAPR produces imprecise results for 5.41% of the studied cases overall. To our knowledge, this is the first empirical study demonstrating that on-the-fly patch validation may produce imprecise/unsound results compared to traditional patch validation. Another interesting finding is that 3 out of the 4 cases with inconsistent patching results occur on the CapGen APR tool. One potential reason is that CapGen is a pattern-based APR system and may generate far more patches than SimFix and ACS. For example, CapGen on average generates over 1,400 patches for each studied bug, while SimFix only generates around 150 on average. In this way, CapGen has way more patches that may affect the correct patch execution than the other studied APR tools. Note that SimFix has only around 150 patches on average since we only studied its fixed bugs, if we had considered the unfixed bugs as well, SimFix will produce many more patches, exposing more imprecise/unsound patch validation issues as well potentially leading to larger UniAPR speedups.

Finding-2: This study presents the first empirical evidence that vanilla on-the-fly patch validation does incur the imprecise/unsound patch validation issue, e.g., failing to fix 5.41% of the studied cases.

5.2 RQ2: Results for On-the-fly Patch Validation via JVM-Reset

5.2.1 Effectiveness. We now present the experimental results for our UniAPR with JVM-reset. We observe that UniAPR with JVM reset has negligible overhead compared with the vanilla UniAPR on all the studied bugs for all the studied APR systems. Figure 9

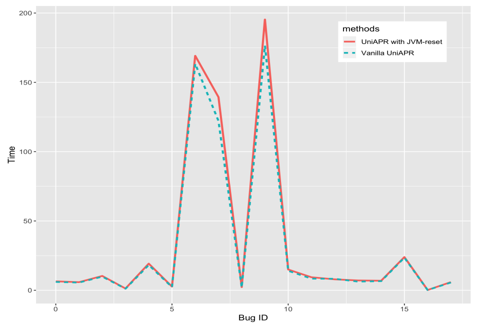


Figure 9: JVM-reset overhead on UniAPR

presents the time cost comparison among the two UniAPR variants on ACS (note that we omit the results for the other two APR tools since they have even lower average overhead). In the figure, the horizontal axis presents all the bugs studied by ACS while the vertical axis presents the time cost; the solid and dashed lines present the time cost for UniAPR with JVM-reset and our vanilla UniAPR. Shown in the figure, JVM reset has incurred negligible overhead among all the studied bugs for ACS on UniAPR, e.g., on average 8.33% overhead. The reason is that class reinitializations only need to be performed at certain sites for only the classes with pollution sites. Also, we have various optimizations to speed up JVM reset. For example, although our basic JVM-reset approach in Figure 5 performs runtime checks on a concurrent `HashMap`, our actual implementation uses arrays for faster class status tracking/check. Furthermore, we observe that the overhead does not change much regardless of the bugs studied, indicating that our UniAPR with JVM-reset has stable overhead across different systems/bugs.

Finding-3: UniAPR with JVM-reset only incurs negligible overhead (e.g., less than 10% for all studied tools) compared to the vanilla UniAPR, demonstrating the scalability of UniAPR with JVM-reset.

5.2.2 Precision. According to our experimental results, UniAPR with JVM-rest is able to produce exactly the same APR results as the traditional patch validation, i.e., UniAPR with JVM-reset successfully fixed all the bugs that vanilla UniAPR failed to fix. We now discuss all the four bugs that UniAPR can fix while vanilla UniAPR without JVM reset cannot fix in details:

```
// org.apache.commons.lang3.StringEscapeUtilsTest.java
    public void testUnescapeHtml4() {
        for (int i = 0; i < HTML_ESCAPES.length; ++i) {
            String message = HTML_ESCAPES[i][0];
            String expected = HTML_ESCAPES[i][2];
            String original = HTML_ESCAPES[i][1];
            // assertion failure: ampersand expected:<bread &[] butter> but was:<
                bread &[amp;] butter>
            assertEquals(message, expected, StringEscapeUtils.unescapeHtml4(
                original));
    ...
```

Figure 10: Test failed for a plausible patch without JVM-reset on Lang-6

Figure 10 presents the test that fails on the only plausible (also correct) patch of Lang-6 (using CapGen) when running UniAPR without JVM-reset. Given the expected resulting string ``‘‘bread &[] butter’’``, the actual returned one is ``‘‘bread &[amp;] butter’’``. Digging into the code, we realize that class `StringEscapeUtils` has a static field named `UNESCAPE_HTML4`, which is responsible for performing the `unescapeHtml4` invocation. However, during earlier patch executions, the actual object state of that field is changed, making the `unescapeHtml4` method invocation return problematic result with vanilla UniAPR. In contrast, when running UniAPR with JVM-reset, field `UNESCAPE_HTML4` will be recreated before each patch execution (if accessed) and will have a clean object state for performing the `unescapeHtml4` method invocation.

```
// org.apache.commons.math3.EventStateTest.java
    public void testIssue695() {
        FirstOrderDifferentialEquations equation = new
                FirstOrderDifferentialEquations() {
        ...
        double tEnd = integrator.integrate(equation, 0.0, y, target, y);
        ...

    private static class ResettingEvent implements EventHandler {
        private static double lastTriggerTime = Double.NEGATIVE_INFINITY;
        public double g(double t, double[] y) {
            // assertion error
            Assert.assertTrue(t >= lastTriggerTime);
            return t - tEvent;
        }
        ...
```

Figure 11: Test failed for a plausible patch without JVM-reset on Math-30 and Math-41

Figure 11 shows another test that fails on the only plausible (and correct) patch of Math-30 when running vanilla UniAPR (without JVM-reset) with CapGen patches as well as the only plausible (and correct) patch of Math-41 when running vanilla UniAPR with SimFix patches. Looking into the code, we find that the invocation of `integrate()` in the test will finally call the method `g()` in class `ResettingEvent` (in the bottom). The static field `lastTriggerTime` of class `ResettingEvent` should be `Double.NEGATIVE_INFINITY` in Java, which means the assertion should not fail. Unfortunately, the earlier patch executions pollute the state and change the value of the field. Thus, the test failed when running with vanilla UniAPR on the two plausible patches. In contrast, UniAPR with JVM-reset is able to successfully recover the field value.

There are four plausible CapGen patches on Math-5 (one is correct) when running with the traditional patch validation. With

```
// org.apache.commons.math3.genetics.UniformCrossoverTest.java
    public class UniformCrossoverTest {
        private static final int LEN = 10000;
        private static final List<Integer> p1 = new ArrayList<Integer>(LEN);
        private static final List<Integer> p2 = new ArrayList<Integer>(LEN);
        public void testCrossover() {
            performCrossover(0.5);
            ...

        private void performCrossover(double ratio) {
            ...
            // assertion failure: expected:<0.5> but was:<5.5095>
            Assert.assertEquals(1.0 - ratio, Double.valueOf((double) from1 / LEN),
                    0.1);
            ...
```

Figure 12: Test1 failed for a plausible patch without JVM-reset on Math-5

```
// org.apache.commons.math3.complex.ComplexTest.java
    public class ComplexTest {
        private double inf = Double.POSITIVE_INFINITY;
        ...
        public void testMultiplyNaNInf() {
            Complex z = new Complex(1,1);
            Complex w = z.multiply(infOne);
            // assertion failure: expected:<-Infinity> but was:<Infinity>
            Assert.assertEquals(w.getReal(), inf, 0);
            ...
```

Figure 13: Test2 failed for a plausible patch without JVM-reset on Math-5

vanilla UniAPR, all the plausible patches failed on some tests. Figure 12 shows the test that fails on three plausible patches (including the correct one) on Math-5. The expected value of the assertion should be 0.5, but the actual value turned to 5.5095 due to the change of variable `from1`. After inspecting the code, we found the value of `from1` is decided by two static fields `p1` and `p2` in class `UniformCrossoverTest`. The other earlier patch executions pollute the field values, leading to this test failure when running with vanilla UniAPR. Figure 13 presents another test that fails on one plausible patch on Math-5. The expected value from invocation `w.getReal()` should be `Infinity`, which should be the same as field `inf` defined in class `ComplexTest`; however, the actual result from the method invocation is `-Infinity`. The root cause of this test failure is similar to the previous ones, the static fields `NaN` and `INF` in class `Complex` are responsible for the result of method invocation `getReal()`. In this way, `getReal()` returns a problematic result because the earlier patch executions changed the corresponding field values. In contrast, using UniAPR with JVM-reset, all the four plausible patches are successfully produced.

Finding-4: UniAPR with JVM-reset is able to successfully fix all the studied bugs, i.e., mitigating the imprecise/unsound patch validation issue by vanilla on-the-fly patch validation.

5.3 Discussion

Having single JVM session for validating more than one patch has the immediate benefit of skipping the costly process creation, validations done by the JVM, and Just-in-Time compilation. As per our experiments, this offers up to several hundred times speed up in patch validation. On the other hand, this approach might have the following limitations:

First, the execution of the patches might interfere with each other, i.e. the execution of some tests in one patch might have side-effects affecting the execution of other tests on another patch. UniAPR mitigates these side-effects by resetting static fields to their default values and resetting JDK properties. Although our experimental results demonstrate that such JVM reset is able to fix all bugs fixed by the traditional patch validation, resetting in-memory state of JVM might not be enough as the side-effects could propagate via operating system or the network. Our current implementation provides a public interface for the users to resolve such issue between patch executions (note that no subject systems in our evaluation require such manual configuration). In the near future, we will study more subject programs to fully investigate the impact of such side effects and design new solutions to address them fully automatically.

Second, HotSwap-based patch validation does not support patches that involve changing the layout of the class, e.g. adding/removing fields and/or methods to/from a class. It also does not support patches that occur inside non-static inner classes, anonymous classes, and lambda abstraction. Luckily, the existing APR techniques mainly target patches within ordinary method bodies, and our UniAPR framework is able to reproduce all correct patches for all the three studied state-of-the-art techniques. Another thing that worths discussion is that HotSwap originally does not support changes in static initializers; interestingly, our JVM-reset approach naturally helped UniAPR to overcome this limitation, since the new initializers can now be reinvoked based on our bytecode transformation to reinitialize the classes. In the near future, we will further look into other more advanced dynamic class redefinition techniques for implementing our on-the-fly patch validation, such as JRebel [60] and DCEVM [59].

6 RELATED WORK

In this section, we first talk about the related work in the automated program repair (APR) area, including techniques for reducing APR cost (Section 6.1). Next, we further talk about the cost reduction techniques in the mutation testing area (Section 6.2), which shares similar cost issues with APR.

6.1 Automated Program Repair

APR has been the subject of intense research in the last decade [13, 15, 43]. APR techniques can be classified into two broad families: (1) techniques that monitor the execution of a system to find deviations from certain specifications, and then heal the system by modifying its runtime state in case of any abnormal behaviors [35, 54], and (2) so-called generate-and-validate (G&V) techniques that attempt to fix the bug by first generating a pool of patches (at the code representation level) and validating the patches via certain rules and/or checks [14, 18, 19, 27, 36, 39, 47, 62]. G&V techniques have been widely studied in recent years, since it can substantially reduce developer efforts in both automated [27] and manual [37] bug fixing for improving software productivity. To date, researchers have designed various G&V APR techniques based on heuristics [19, 27, 34], constraint solving [11, 41, 47, 66], and pre-defined templates [14, 23, 32]. *Heuristic-based APR techniques* investigate various strategies to explore a (potentially infinite) search space of syntactic program changes. GenProg [27], the pioneering work for APR, leverages genetic programming to compose and mutate single-change patches into more advanced ones that can fix more complex bugs. Later on, RSRepair [55] demonstrates that using random search rather than genetic programming can help GenProg to mitigate the search space explosion problem. Recently, SimFix [19] and ssFix [64] leverage code search information (e.g., from the current project under test or even other projects) to help further reduce the potential search space. *Constraint-based APR techniques* generally leverage advanced constraint-solving or synthesis techniques to fix certain types of bugs. For example, ACS [65], Nopol [66], and Cardumen [40] aim to fix problematic conditional expressions. Nopol [66] transforms the APR problem into a satisfiability problem and leverages off-the-shelf SMT solvers to find the fixing ingredients. Cardumen [40] fixes bugs by synthesizing potential correct expression candidates with its mined templates from the current program under repair. ACS [65] leverages various dimensions of information (including local contexts, API document, and code mining information) to directly synthesize the correct conditional expressions. *Template-based APR techniques* are also often denoted as pattern-based APR techniques. The basic idea is to fix program bugs via a set of predefined rules/templates/patterns. The PAR [23] work produces patches based on a list of fixing patterns manually summarized from a large number of human-written patches. Later on, researchers have also proposed various techniques to automatically mine potential fixing patterns from historical bug fixes, such as the CapGen [62] and FixMiner [25] work. More recently, the TBar [32] work presents an empirical study of prior fixing patterns used in the literature.

Despite this spectacular progress in designing new APR techniques, very few of the works have attempted to reduce the time cost for APR, especially the patch validation time which dominates repair process. For example, JAID [6] uses patch schema to fabricate meta-programs that bundle several patches in a single source file, while SketchFix [17] uses sketches [28] to achieve a similar effect. Although they can potentially help with patch generation and compilation, they still require validating each patch in a separte JVM, and have been shown to be rather costly during patch validation [14]. More recently, PraPR [14] uses direct bytecode-level mutation and HotSwap technique to generate and validate patches on-the-fly, thereby bypassing expensive operations such as AST manipulation and compilation on patch generation side and process creation and JVM warm-up on patch validation side. This makes PraPR at least an order of magnitude faster than state-of-the-art APR (including JAID and SketchFix). However, PraPR is limited to only the bugs that can be fixed via bytecode manipulation, and can also return imprecise patch validation results due to the potential JVM pollution.

Motivated by PraPR work, in this paper, we introduce UniAPR, the first unified APR framework for all source-code and bytecode level APR. Compared with PraPR, UniAPR can be applied to any of the existing source-level APR technique and is not limited by the PraPR bug-fixing patterns at the bytecode level. Furthermore, although UniAPR also uses the JVM HotSwap technique to validate patches without unnecessary restart of JVM sessions, different from PraPR, UniAPR resets JVM internal state between patches so as to contain side effects of patch executions for fast&precise patch validation. Lastly, unlike PraPR, this framework is generic and can use any existing APR technique as a *patch generation add-on*; in this way, it provides a natural platform for combining different APR techniques to take maximum advantage of their strengths.

6.2 Cost Reduction for Mutation Testing

Mutation testing [2], a traditional testing methodology studied for decades, aim to generate artificial bugs to simulate real bugs for test quality evaluation. In recent years, mutation testing has also been applied to various other areas, including regression testing [38, 57], test generation [52, 68], bug localization [29, 30, 46, 51, 69], and even APR [10, 14]. In fact, mutation testing shares similarities with APR in that both involve making small changes to program and repeatedly running tests. The repeated executions of tests on a large number of mutated program versions also dominate the end-to-end time of mutation testing. Therefore, a body of research is devoted to reduce such mutation execution cost [7, 24, 61, 67]. Weak mutation testing [16], one of the earliest techniques that attempts to reduce mutation testing cost, executes each mutated program partially (only to the mutated program locations) and checks the program internal state change to reduce the mutation testing time. However, it may produce rather imprecise mutation testing results, since the internal state changes may not always propagate to the end. Split-stream [24] attempts to reduce mutation testing cost by reusing the state before the first mutation point. In a recent work [61], Wang et al. introduce the concept of Equivalence Modulo States to further elaborate on split-stream technique by reusing the shared states, as much as possible, even after the mutation point. In all these optimization approaches, creating a new process is done as a last resort to avoid the overhead of process creation and to use the shared state as much as possible. However, such techniques cannot handle programs with external resource accesses (e.g., file/network accesses) well [61]. Process creation overhead in systems like JVM is especially pronounced as virtually all the optimization tasks, as well as Just-in-Time compilation, are done at runtime. Therefore, compared to natively executed programs, JVM-based programs are expected to take relatively longer time to warm up. Therefore, state-of-the-art mutation engine for JVM-based systems, PIT [7], attempts to use a single process to perform mutation testing instead of creating a separate process for testing each mutation. PIT has been demonstrated to achieve a significant speedup over similar mutation engines (such as MAJOR [21] and Javalanche [56]), and represent state of the art. In fact, the PraPR APR tool is built on top the PIT mutation engine. Compared with our UniAPR system, PIT only supports very simple change/mutation patterns at the bytecode level and also cannot handle the JVM state pollution problem.

7 CONCLUSION

Automated program repair (APR) has been extensively studied in the last decade, and various APR systems/tools have been proposed. However, state-of-the-art APR tools still suffer from the efficiency problem largely due to the expensive patch validation process. In this work, we have proposed a unified on-the-fly patch validation framework for all JVM-based APR systems. Compared with the existing on-the-fly patch validation work [14] which only works for bytecode APR, this work generalizes on-the-fly patch validation to all existing state-of-the-art APR systems, even including systems at the source code level. This work also shows the first empirical evidence that on-the-fly patch validation can incur imprecise/unsound patch validation results, and further introduces a new technique for resetting JVM state for precise patch validation. The experimental results show that this work can speed up state-of-the-art representative APR tools, including CapGen, SimFix, and ACS, by over an order of magnitude without incurring any imprecision.

REFERENCES

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2006. *Compilers, principles, techniques.*

[2] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing.* Cambridge University Press.

[3] Earl T. Barr, Yuriy Brun, Premkumar T. Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *FSE (FSE'14)*. 306–317.

[4] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering.* 550–561.

[5] CO Boulder. 2013. University of Cambridge Study: Failure to Adopt Reverse Debugging Costs Global Economy $41 Billion Annually. https://bit.ly/2T4fWnq. Accessed: May, 2019.

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

[7] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis.* 449–452.

[8] Oracle Corporation. 2020. Java Instrumentation API. https://bit.ly/3czmzFV Accessed: May, 2020.

[9] Xuan Bach D Le. 2018. *Overfitting in Automated Program Repair: Challenges and Solutions.* Ph.D. Dissertation. Singapore Management University.

[10] V. Debroy and W. E. Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *ICST*. 65–74.

[11] 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.* 85–91.

[12] Apache Software Foundation. 2020. Apache Maven. http://maven.apache.org/ Accessed: May, 2020.

[13] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.

[14] 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.

[15] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[16] William E. Howden. 1982. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering* 4 (1982), 371–379.

[17] 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.* 12–23.

[18] 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).* 255–266.

[19] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *ISSTA*. ACM, 298–309.

[20] JodaOrg. 2020. Joda Time. https://github.com/JodaOrg/joda-time Accessed: May, 2020.

[21] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* 433–436.

[22] 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.

[23] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE*. IEEE Press, 802–811.

[24] Kim N King and A Jefferson Offutt. 1991. A fortran language system for mutation-based software testing. *Software: Practice and Experience* 21, 7 (1991), 685–718.

[25] 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.

[26] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *SANER*, Vol. 1. IEEE, 213–224.

[27] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE TSE* 38, 1 (2012), 54–72.

[28] A Solar Lezama. 2008. *Program synthesis by sketching.* Ph.D. Dissertation. PhD thesis, EECS Department, University of California, Berkeley.

[29] 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.

[30] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.

[31] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2017. *The Java Virtual Machine Specification, Java SE 9 Edition* (1st ed.). Addison-Wesley Professional.

[32] 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.

[33] 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).*

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

[35] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. In *PLDI (PLDI'14).* 227–238.

[36] 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.

[37] 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*. to appear.

[38] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 46–57.

[39] 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 2016).* 441–444.

[40] Matias Martinez and Martin Monperrus. 2018. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *International Symposium on Search Based Software Engineering.* Springer, 65–86.

[41] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering.* 691–701.

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

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

[44] Martin Monperrus. 2018. *The Living Review on Automated Program Repair.* Technical Report. Technical Report hal-01956501. HAL/archives-ouvertes. fr, HAL/archives âĂę.

[45] Martin Monperrus, Simon Urli, Thomas Durieux, Matias Martinez, Benoit Baudry, and Lionel Seinturier. 2019. Repairnator Patches Programs Automatically. *Ubiquity* 2019, July, Article Article 2 (July 2019), 12 pages. https://doi.org/10.1145/3349589

[46] 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.

[47] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE).* IEEE, 772–781.

[48] Flemming Nielson and Hanne Riis Nielson. 2019. *Formal Methods.* Springer.

[49] Objectweb. 2020. ASM Bytecode Manipulation Framework. https://asm.ow2.io/ Accessed: May, 2020.

[50] Oracle. 2020. The Java Language Specification. https://docs.oracle.com/javase/specs/ Accessed: May, 2020.

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

[52] Mike Papadakis, Nicos Malevris, and Maria Kallia. 2010. Towards automating the generation of mutation tests. In *Proceedings of the 5th Workshop on Automation of Software Test.* 111–118.

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

[54] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. 2009. Automatically patching errors in deployed software. In *SOSP (SOSP'09).* 87–102.

[55] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering.* 254–265.

[56] David Schuler and Andreas Zeller. 2009. Javalanche: efficient mutation testing for Java. In *Proceedings of the 7th joint meeting of the European software engineering*

conference and the ACM SIGSOFT symposium on The foundations of software engineering. 297–298.

[57] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 246–256.

[58] Undo Software. 2016. Increasing software development productivity with reversible debugging. https://bit.ly/3c8ccbn. Accessed: May, 2019.

[59] DCEVM Team. 2020. Dynamic Code Evolution VM for Java. http://dcevm.github.io/ Accessed: May, 2020.

[60] JRebel Team. 2020. JRebel. https://bit.ly/3fQox6Y Accessed: May, 2020.

[61] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster Mutation Analysis via Equivalence modulo States. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. 295âĂŞ306.

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

[63] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE TSE* 42, 8 (Aug. 2016), 707–740.

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

[65] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *ICSE (ICSE'17)*. IEEE Press, 416–426.

[66] 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 TSE* 43, 1 (2017), 34–55.

[67] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2018. Predictive mutation testing. *IEEE Transactions on Software Engineering* 45, 9 (2018), 898–918.

[68] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. 2010. Test generation via dynamic symbolic execution for mutation testing. In *2010 IEEE International Conference on Software Maintenance*. IEEE, 1–10.

[69] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. *ACM SIGPLAN Notices* 48, 10 (2013), 765–784.