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

Lingchao Chen
Department of Computer Science
The University of Texas at Dallas
lxc170330@utdallas.edu

Yicheng Ouyang
Department of Computer Science
The University of Texas at Dallas
yicheng.ouyang@utdallas.edu

Lingming Zhang
Department of Computer Science
University of Illinois at Urbana-Champaign
lingming@illinois.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 the manipulation of program code to fabricate a large number of patches and also the repeated test executions on patches to identify potential fixes. PraPR, a recent G&V APR technique, reduces such costs by modifying program code directly at the level of compiled JVM bytecode with *on-the-fly patch validation*, which directly allows multiple bytecode patches to be tested within the same JVM process. However, PraPR is limited due to its unique bytecode-repair design, and 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 process. In this paper, we propose a unified patch validation framework, named UniAPR, to perform the first empirical study of on-the-fly patch validation for state-of-the-art source-code-level APR techniques widely studied in the literature; furthermore, UniAPR addresses the imprecise patch validation issue by resetting the JVM global state via runtime bytecode transformation. We have implemented UniAPR as a publicly available fully automated Maven Plugin. Our study demonstrates for the first time that on-the-fly patch validation can often speed up state-of-the-art source-code-level APR by over an order of magnitude, enabling all existing APR techniques to explore a larger search space to fix more bugs in the near future. Furthermore, our study shows the first empirical evidence that vanilla on-the-fly patch validation can be imprecise/unsound, while UniAPR with JVM reset is able to mitigate such issues with negligible overhead.**

I. 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 [44]. Therefore, a large body of research efforts have been dedicated to automated debugging techniques [49], [36], [12]. Among the existing debugging techniques, Automated Program Repair [14] (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 [36], [12].

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 [12]. A patch is said to be *plausible* if it passes all the checks. Ideally, we would apply formal verification [40] 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 [1]. Therefore, the vast majority of recent G&V APR techniques leverage developer tests as the criteria for checking correctness of the generated patches [12], 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 program code to fabricate/generate patches 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 often act in a rather 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 [8]. Therefore, apart from introducing new and/or 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 [15] uses sketches [23] 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 [35]. Most recently, PraPR [13] aims to reduce both patch generation and validation time by modifying program code directly at the bytecode level with *on-the-fly patch validation*, which directly allows multiple bytecode-level patches to be tested within the same JVM process. However, bytecode-level APR is not flexible (e.g., large-scope changes can be extremely hard to implement at the bytecode level) and fails to fix many bugs that can be fixed at the source-code level [13]; furthermore, PraPR requires decompilation (which may be imprecise or even fail) to decompile the bytecode-level patches for manual inspection. In fact, all other popular general-purpose G&V APR techniques fix at the source code level.

In this paper, we propose a unified test-based patch validation framework, named UniAPR, to empirically study the impact of *on-the-fly* patch validation for state-of-the-art source-code-level APR techniques. While existing source-code-level APR usually restarts a new JVM process for each patch, our on-the-fly patch validation aims to use a single JVM process for patch validation, as much as possible, and leverages JVM's dynamic class redefinition feature (a.k.a. the HotSwap mechanism and Java Agent technology [7]) to only reload the patched bytecode classes on-the-fly for each patch. In this way, UniAPR not only avoids reloading (also including linking and initializing) all used classes for each patch (i.e., only reloading the *patched* bytecode files), but also can avoid the unnecessary JVM warm-up time (e.g., the accumulated JVM profiling information across patches enables more and more code to be JIT-optimized and the already JIT-optimized code can also be shared across patches).

UniAPR has been implemented as a fully automated Maven [11] plugin (available at [47]), 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 [48], SimFix [17], and ACS [50] that are modern representatives of template-/pattern-based [9], [21], heuristic-based [2], [22], and constraint-based [51], [39] techniques. Our empirical study shows for the first time that on-the-fly patch validation can often speed up state-of-the-art APR systems by over an order of magnitude, enabling all existing APR techniques to explore a larger search space to fix more bugs in the near future.

Furthermore, our study (Section V-A2) shows the first empirical evidence that when sharing JVM across multiple patches, 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. Therefore, we further propose the first solution to address such imprecision problem by isolating patch executions via resetting JVM states after each patch execution using runtime bytecode transformation. Our experimental results show that our UniAPR with JVM reset is able to the avoid imprecision/unsoundness of vanilla on-the-fly patch validation with negligible overhead.

We envision a future wherein all existing APR tools (like SimFix [17], CapGen [48], and ACS [50]) and major APR frameworks (like ASTOR [33] and Repairnator [38]) are leveraging this framework for patch validation. In this way, researchers will only need 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. In summary, this paper makes the following contributions:

- **Framework.** We introduce the first unified on-the-fly patch validation framework, UniAPR, to empirically study the impact of on-the-fly patch validation for state-of-the-art source-code-level APR techniques.
- **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 [7], and implemented the JVM-reset technique based on the ASM bytecode manipulation framework [41]; the overall UniAPR tool has been implemented as a practical Maven plugin [47], and can accept different APR techniques as patch generation add-ons.
- **Empirical Study.** We conduct a large-scale study of the effectiveness of UniAPR on its interaction with state-of-the-art APR systems from three different APR families, demonstrating that UniAPR can often speed up state-of-the-art APR by over an order of magnitude (without validation imprecision/unsoundness). Furthermore, the study results also indicate that UniAPR can serve as a unified platform to naturally support hybrid APR to directly combine the strengths of different APR tools.

II. Background and Related Work

In this section, we first discuss the current status of automated program repair (Section II-A); then, we introduce Java Agent and HotSwap, on which UniAPR is built (Section II-B).

A. Automatic Program Repair

Automatic program repair (APR) aims to suggest likely patches for buggy programs to reduce the manual effort during debugging. The widely studied generate-and-validate (G&V) techniques attempt to fix bugs by first generating a pool of patches and then validating the patches via certain rules and/or checks [22], [39], [48], [13], [17], [33], [30], [29]. 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). G&V APR techniques [22], [39], [48], [13], [17], [33], [31], [16] have been extensively studied in recent years, since it can substantially reduce developer efforts in bug fixing. According to a recent work [27], researchers have designed various APR techniques based on heuristics [28], [22], [17], constraint solving [51], [10], [39], [34], and pre-defined templates [20], [13], [26]. Besides *automated* bug fixing, researchers have also proposed Unified Debugging [32], [4] to leverage various off-the-shelf APR techniques to help with *manual* bug fixing. In this way, the application scope of APR techniques has been extended to all possible bugs, not only the bugs that can be automatically fixed.

Meanwhile, despite the spectacular progress in designing and applying new APR techniques, very few techniques 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

[15] uses sketches [23] 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 [13]. More recently, PraPR [13] uses direct bytecode-level mutation and HotSwap to generate and validate patches on-the-fly, thereby bypassing expensive operations such as AST manipulation/compilation on the patch generation side as well as process creation and JVM warm-up on the patch validation side. This makes PraPR substantially 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 potential JVM pollution.

B. Java Agent and HotSwap

A Java Agent [7] 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 [7] 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. The 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 `premain` method 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 `redefineClasses` method 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 result 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/bytecode APR) on-the-fly sharing the same JVM for much faster patch validation.

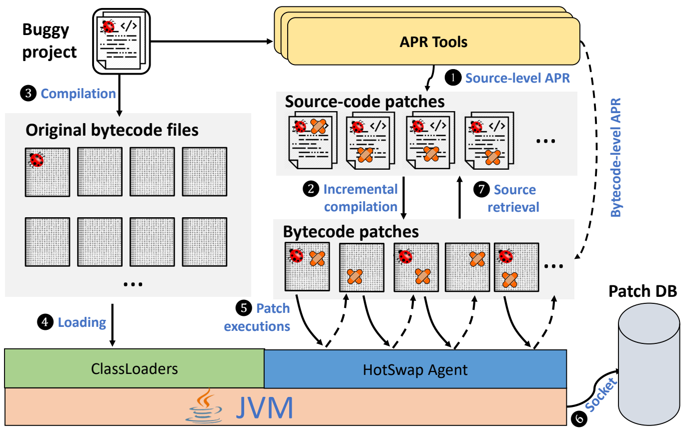


Fig. 1: UniAPR workflow

III. Approach

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 [13] (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. Also note that besides constructed before patch validation, the patch pool can also be continuously generated during the patch-validation process[1]; in either way, UniAPR's reduction on patch-validation time is also not affected.

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, when 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

[1]If patches are continuously generated, the patch-validation component needs to obtain the live stream of patch information from the running patch-generation component (e.g., via lightweight socket connections).

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 tools representing three different families of APR techniques. These add-ons include CapGen [48] (representing pattern/template-based APR techniques), SimFix [17] (representing heuristic-based techniques), and ACS [50] (representing constraint-based techniques). Of course, users of UniAPR can also easily build new patch generation add-ons for other APR tools. For existing APR tools, this can be easily done by modifying their source code so that the tools abandon validation of patches after generating/compiling them.

Next, we will talk about our detailed design for *fast* patch validation via on-the-fly patching (Section III-A) as well as *precise* patch validation via JVM reset (Section III-B).

A. *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 candidate 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[2]. 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) to evaluate the remaining patches (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) $\mathcal{C}_{patched}$ 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 $\mathcal{C}_{patched}$ with the patched bytecode file(s) $\mathcal{F}_{patched}$; 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).

[2]Note that here we assume that $\mathbb{P}$ is available before patch validation for the ease of presentation, but our overall approach is general and can also easily handle the case where $\mathbb{P}$ is continuously constructed during patch validation.

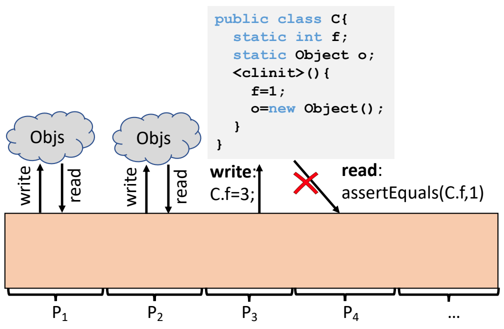
Algorithm 1: Vanilla on-the-fly patch validation

```
Input: Original buggy program $\mathcal{P}$, test suite $\mathcal{T}$, and set of candidate patches $\mathbb{P}$
Output: Validation status
        $\mathcal{R} : \mathbb{P} \rightarrow \{\texttt{PLAUSIBLE}, \texttt{NON-PLAUSIBLE}, \texttt{ERROR}\}$
1  begin
2      $\mathcal{R} \leftarrow \mathbb{P} \times \{\texttt{UNKNOWN}\}$ ; // initialize result function
3      while True do
4          $\mathbb{P}_{left} \leftarrow \{\mathcal{P}' \mid \mathcal{P}' \in \mathbb{P} \wedge \mathcal{R}(\mathcal{P}') = \texttt{UNKNOWN}\}$// get all
             the left patches not yet validated
5          if $\mathbb{P}_{left} = \emptyset$ then
6              return $\mathcal{R}$ // return if 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($\mathcal{P}'$)
           // Swap in the patched bytecode files
13         $\mathcal{F}_{orig} \leftarrow$ HotSwapAgent.swap($\mathcal{JVM}, \mathcal{C}_{\text{patched}}, \mathcal{F}_{patched}$)
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
           // Swap back the original bytecode files
27         HotSwapAgent.swap($\mathcal{JVM}, \mathcal{C}_{patched}, \mathcal{F}_{orig}$)
```

B. *Precise Patch Validation via JVM Reset*

1) Limitations for vanilla on-the-fly patch validation: The vanilla on-the-fly patch validation presented in Section III-A works for most patches of most buggy projects. The basic process can be illustrated via Figure 2. In the figure, each



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

Fig. 2: Imprecision under vanilla UniAPR

```
// 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);
  ...
```

Fig. 3: Static field dependency

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.

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 rather challenging:

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 [18] from the widely studied Defects4J dataset [19]. The easiest way to keep such ordering and reset `final` fields is to simply re-invoke the original class initializer. However, according to the JVM specification, only

```
// 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);
  }
  ...
```

Fig. 4: Static initializer dependency

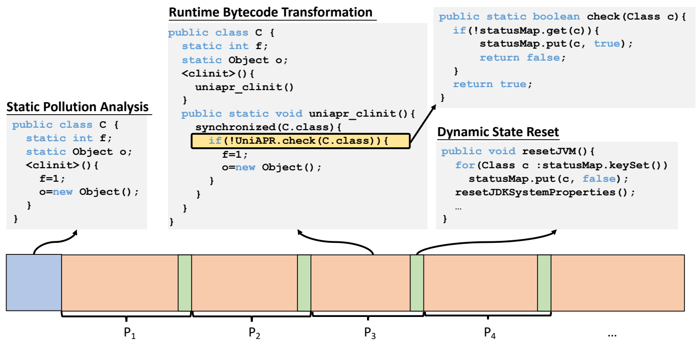
JVM can invoke such static class initializers.

Second, simply invoking the class initializers for all classes with pollution sites may not work. For example, a naive way to reset the pollution sites is to simply trace the classes with pollution sites executed during each patch execution; then, we can simply force JVM to invoke all their class initializers after each patch execution. However, it can bring side effects 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 as if they had been executed on a new JVM.

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. Therefore, we choose to simulate the JVM customizations at the application level.

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 [3], we perform runtime bytecode transformation to simulate JVM class initializations for patch execution isolation for the first time. 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



```
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();
  …
}
```

Fig. 5: On-the-fly patch validation via JVM reset

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.

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 in `T` is executed

TABLE I: Class initialization conditions

Runtime Bytecode Transformation. According to Java Language Specification (JSL) [42], static class initializers get invoked when any of the five conditions shown in Table I 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 I. Note that our implementation also handles the non-conventional Reflection-based accesses to such potential pollution sites. Since JVM does not allow class initialization at the application level, following prior work on speeding up traditional regression testing [3], 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 I. Therefore, we need to add the dynamic check to ensure that each class only gets (re)initialized once for each patch execution. Shown in Figure 5, the orange blocks denote different patch executions. During each patch execution, the classes with pollution sites will be transformed at runtime. For example, class `C` will be transformed into the code block connected with the $\mathcal{P}_3$ patch execution in Figure 5; the yellow line in the transformed code denotes the dynamic check to ensure that `C` is only initialized once for each patch. The pseudo code for the dynamic check is shown in the top-right of the figure: the check maintains a `ConcurrentHashMap` for the classes with pollution sites and their status (`true` means the corresponding class has been reinitialized). The entire initialization is also synchronized based on the corresponding `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). (Note that this simplified mechanism is just for illustration purpose; our actual implementation manipulates arrays with optimizations for faster and safe tracking/check.) 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 `true` value 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 `ConcurrentHashMap`. In this way, during the next patch execution, all the used classes within the `ConcurrentHashMap` 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 after $\mathcal{P}_3$'s execution simply resets the status for all classes within the status map as `false` and also resets potential JDK pollution sites within classes.

IV. Experimental Setup

A. *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?

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 II: Defects4J V1.0.0 statistics

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.

B. Benchmarks

We choose the Defects4J (V1.0.0) benchmark suite [19], since it contains hundreds of real-world bugs from real-world systems, and has become the most widely studied dataset for program repair [13], [10], [6], [17], [48] or even software debugging in general [24], [25], [4]. Table II 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.

C. Studied Repair Tools

Being a well-developed field, APR offers us a cornucopia of choices to select from. According to a recent study [27], 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 [37]. 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 III presents all such existing Java-based APR tools, which can be categorized into three main categories according to prior work [27]: heuristic-based [22], [17], [28], constraint-based [51], [10], and template-based [26], [48] 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 [50], SimFix [17], and CapGen [48] 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, and (2) UniAPR precision, i.e., whether the patch validation results are consistent with and without UniAPR.

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 III: Available Java APR tools for Defects4J [27]

D. Implementation

UniAPR has been implemented as a publicly available fully automated Maven plugin [47], 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. In this way, once the users fire command: `mvn org.uniapr:uniapr-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 current UniAPR version assumes the patch directory generated by the APR add-ons to include all available patches represented by their patched bytecode files, i.e., the patch pool is constructed before patch validation. Note that, each patch may involve more than one patched bytecode file, e.g., some APR tools (such as SimFix [17]) can fix bugs with multiple edits.

During patch validation, UniAPR 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 reports their status. We use TCP Socket Connections to communicate 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 easy for UniAPR to fork two or more processes to take maximum advantage of today's powerful machines. However, for a fair comparison with existing work, we always ensure that only one JVM is running patch validation at any given time stamp.

E. 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). Note that the only exception is for CapGen: digging into the decompiled CapGen code (CapGen source code is not available), we observed that CapGen excluded some (expensive) tests for certain bugs via unsafe test selection. Such unsafe test selection is inconsistent with the original paper [48], and can be dangerous (i.e., it may fail to falsify incorrect patches). Therefore, to enable a fair and realistic study, for CapGen, we build a variant for vanilla UniAPR that simply restarts a new JVM for each patch (same as CapGen) to simulate the original CapGen performance. Note that if we had presented the performance comparison between UniAPR and the original CapGen using the same reduced tests, the UniAPR speedup can be even larger because UniAPR mainly reduces the JVM-restart overhead — similar reduction on JVM overhead would yield larger overall speedup given shorter test-execution time (as the overall patch-validation time includes JVM overhead and test-execution time). For example, the average speedup achieved by UniAPR with JVM-reset on Chart bugs is 15.7X compared with the original CapGen (on the same set of reduced tests) and 8.4X compared with our simulated CapGen.

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 on all the available tests, and trace the new patch validation time and results. Note that we repeat this step for both variants of UniAPR (i.e., vanilla UniAPR and UniAPR with JVM reset) to evaluate their respective performance.

To evaluate our UniAPR variants, 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.

V. Result Analysis

A. RQ1: Vanilla On-the-fly Patch Validation

1) Effectiveness: For answering this RQ, we evaluated vanilla UniAPR (i.e., without JVM-reset) that is configured to use the add-on corresponding to each studied APR tool. The main experimental results are presented in Figure 6. 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 present the time cost for traditional patch validation and our vanilla UniAPR, respectively.

From the figure, we can observe that the vanilla UniAPR can substantially speed up the existing patch validation component for all state-of-the-art APR tools with almost no slowdowns. For example, when running ACS on Math-25, the traditional patch validation costs 698s, while on-the-fly patch validation via vanilla UniAPR takes only 2.3s to produce the same patch validation results, i.e., 304.89X speedup; when running SimFix on Lang-60, the traditional patch validation costs 924s, while vanilla UniAPR takes only 4s to produce the same patch validation results, i.e., 229.96X speedup; when running CapGen on Math-80, the traditional patch validation costs 18,991s, while vanilla UniAPR takes only 1582s to produce the same patch validation results, i.e., 12.00X speedup. Note that we have further marked various peak speedups in the figure to help better understand the effectiveness of UniAPR. To our knowledge, *this is the first study demonstrating that on-the-fly patch validation can also substantially speed up state-of-the-art source-level APR.*

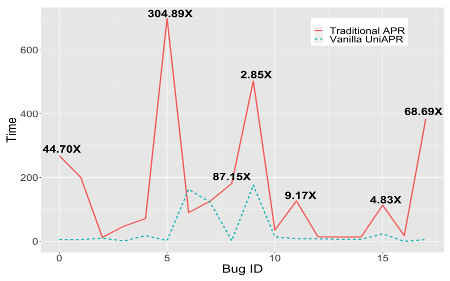
Tool	# All	# Mismatch	Ratio (%)
CapGen	22	3	13.64%
SimFix	34	1	2.94%
ACS	18	0	0.00%
All	74	4	5.41%

TABLE IV: Inconsistent fixing results

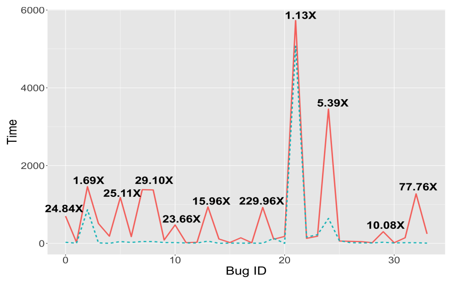
2) Precision: We further study the number of bugs that vanilla UniAPR does not produce the same repair results as the traditional patch validation (that restarts a new JVM for each patch). Table IV presents the summarized results for all the studied APR tools on all their fixable bugs. In this table, 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 all Defects4J bugs studied by the original SimFix paper (including the bugs that cannot be fixed by SimFix), SimFix will produce many more patches, exposing more imprecise/unsound patch validation issues as well as leading to much larger UniAPR speedups.

B. RQ2: On-the-fly Patch Validation via JVM-Reset

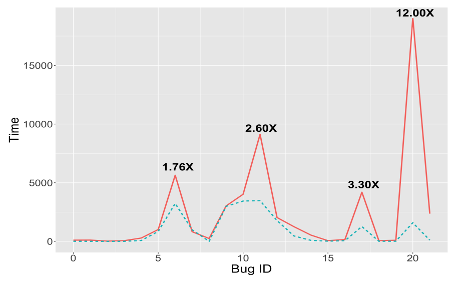
1) Effectiveness: We now present the experimental results for our UniAPR with JVM-reset. The main experimental results are presented in Figure 7. In each sub-figure, the horizontal axis presents all the bugs that have been reported to be fixed by each studied tool, while the vertical axis presents the time cost (s); the solid and dashed lines present the time cost for traditional patch validation and UniAPR with JVM reset, respectively. From the figure, we can observe that for all the studied APR tools, UniAPR with JVM reset can also substantially speed up the existing patch validation component with almost no performance degradation. For example, when running ACS on Math-25, the traditional patch validation costs 698s, while on-the-fly patch validation via UniAPR



(a) ACS

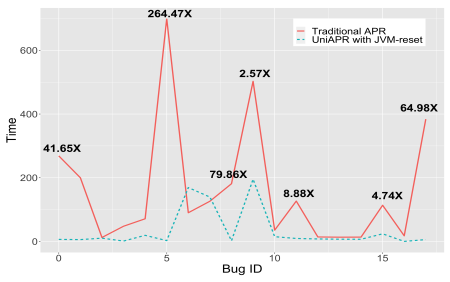


(b) SimFix

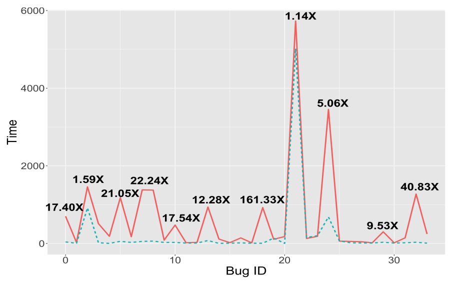


(c) CapGen

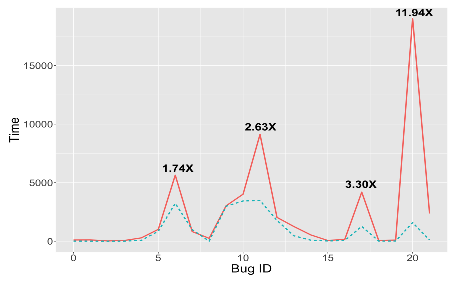
Fig. 6: Speedup achieved by vanilla UniAPR



(a) ACS

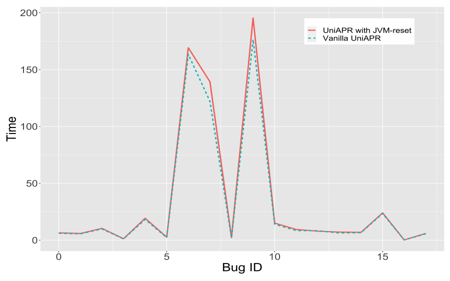


(b) SimFix

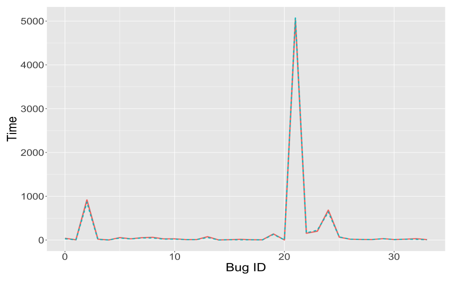


(c) CapGen

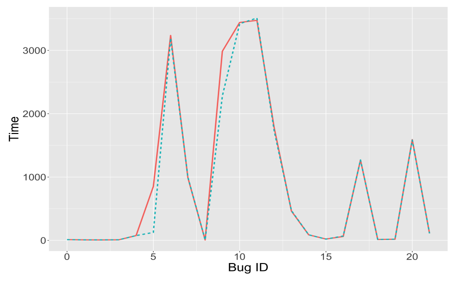
Fig. 7: Speedup achieved by UniAPR with JVM reset



(a) ACS



(b) SimFix



(c) CapGen

Fig. 8: JVM-reset overhead over vanilla UniAPR

with JVM reset takes only 2.6s to produce the same patch validation results, i.e., 264.47X speedup. Note that we have also marked various peak speedups in the figure to help better understand the effectiveness of UniAPR with JVM reset. While we observe clear speedups for the vast majority of the bugs (and almost no slowdowns), 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 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 [43] between the number of patches for each studied bug and its corresponding speedup for ACS. Shown in Figure 9, the horizontal axis denotes the number of patches, while the vertical axis denotes the per-patch speedup (X) 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 with a clear positive coefficient R of 0.51 and a p value of 0.031 (which is statistically significant at the significance level of 0.05), demonstrating that *UniAPR with JVM reset can also substantially outperform existing patch validation, with larger speedups for larger systems with more patches.*

Meanwhile, we observe that UniAPR with JVM reset has rather close performance compared with the vanilla UniAPR (shown in Figure 6 and Figure 7), indicating that UniAPR with JVM reset has negligible overhead compared with the vanilla UniAPR on all the studied bugs for all the studied APR systems. To confirm our finding, Figure 8 further presents the time cost comparison among the two UniAPR variants on the three APR systems. In the figure, the horizontal axis presents all the bugs studied for each system while the vertical axis presents the time cost; the solid and dashed lines present the time cost for UniAPR with JVM-reset and vanilla UniAPR,

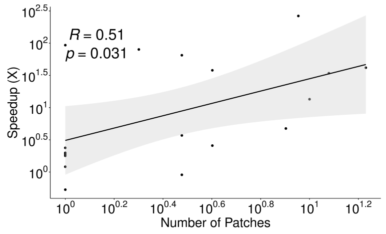


Fig. 9: Correlation between patch number and speedup achieved by UniAPR with JVM reset

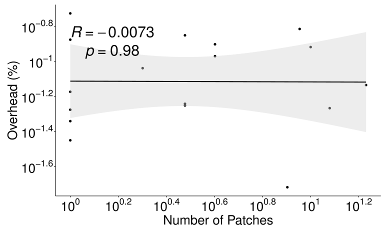


Fig. 10: Correlation between patch number and overhead incurred by JVM-reset over vanilla UniAPR

respectively. Shown in the figure, JVM reset has incurred negligible overhead among all the studied bugs for all three systems on UniAPR, e.g., on average 8.33%/7.81%/1.72% overhead for ACS/CapGen/SimFix. 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 `ConcurrentHashMap`, 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, e.g., our UniAPR with JVM-reset has stable overhead across bugs with different number of patches. To further confirm our finding, we perform the Pearson Correlation Coefficient analysis [43] between the number of patches for each studied bug and the corresponding JVM-reset overhead (over vanilla UniAPR) on the ACS tool with the highest overhead. Shown in Figure 10, the horizontal axis denotes the number of patches, while the vertical axis denotes the overhead (%) incurred; each data point represents one studied bug for ACS. From this figure, we can observe that there is no clear correlation (at the significance level of 0.05), i.e., JVM-reset overhead is not affected by the numbers of patches. In summary, *UniAPR with JVM-reset only incurs negligible and stable overhead (e.g., less than 8.5% for all studied tools) compared to the vanilla UniAPR, demonstrating the scalability of UniAPR with JVM-reset.*

2) Precision: According to our experimental results, UniAPR with JVM-rest produces 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, mitigating the imprecision/unsoundness of vanilla UniAPR.* We now discuss all 4 bugs that UniAPR with JVM reset can fix while vanilla UniAPR without JVM reset cannot fix:

Figure 11 presents the test that fails on the only plausible (also correct) patch of Lang-6 (using CapGen) when running

```
// 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));
   ...
```

Fig. 11: Test failed without JVM-reset on Lang-6

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()` method 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;
      }
      ...
```

Fig. 12: Test failed without JVM-reset on Math-30/41

Figure 12 shows another test that fails on the only plausible (and correct) patch of Math-30 when running vanilla UniAPR with CapGen patches, and fails on 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 that.

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

```
// 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);
      ...
```

Fig. 13: Test failed 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);
      ...
```

Fig. 14: Another test failed without JVM-reset on Math-5

With vanilla UniAPR, all the plausible patches failed on some tests. Figure 13 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 14 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.

C. Discussion

Having single JVM session for validating more than one patch has the immediate benefit of skipping costly JVM restart, reload, and warm-up. As shown by our empirical study, this offers substantial speedups 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 can fix all bugs fixed by the traditional patch validation and opens a new dimension for fast&precise patch validation, such in-memory JVM state reset for only class fields might not be sufficient to handle all cases. Also, 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 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. 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 worth discussion is that HotSwap originally does not support changes in static initializers; interestingly, our JVM-reset approach can naturally help UniAPR 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 promising dynamic class redefinition techniques for implementing our on-the-fly patch validation, such as JRebel [46] and DCEVM [45].

VI. 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 [13] which only works for bytecode APR, this work generalizes on-the-fly patch validation to all existing state-of-the-art APR systems at the source code level. This work shows the first empirical results that on-the-fly patch validation can speed up state-of-the-art representative APR systems, including CapGen, SimFix, and ACS, by over an order of magnitude. Furthermore, 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 with negligible overhead.

Acknowledgements

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

References

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

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

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

[4] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. On the effectiveness of unified debugging: An extensive study on 16 program repair systems. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 907–918, 2020.

[5] CO Boulder. University of cambridge study: Failure to adopt reverse debugging costs global economy $41 billion annually. https://bit.ly/2T4fWnq, 2013. Accessed: August, 2020.

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

[7] Oracle Corporation. Java instrumentation api, 2020. Accessed: August, 2020. URL: https://bit.ly/3czmzFV.

[8] Xuan Bach D Le. *Overfitting in Automated Program Repair: Challenges and Solutions*. PhD thesis, Singapore Management University, 2018.

[9] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *ICST*, pages 65–74, April 2010.

[10] Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *AST*, pages 85–91, 2016.

[11] Apache Software Foundation. Apache maven, 2020. Accessed: August, 2020. URL: http://maven.apache.org/.

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

[13] Ali Ghanbari, Samuel Benton, and Lingming Zhang. Practical program repair via bytecode mutation. In *ISSTA*, pages 19–30, 2019.

[14] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

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

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

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

[18] JodaOrg. Joda time, 2020. Accessed: August, 2020. URL: https://github.com/JodaOrg/joda-time.

[19] René Just, Darioush Jalali, and Michael D Ernst. 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*, pages 437–440, 2014.

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

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

[22] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE TSE*, 38(1):54–72, 2012.

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

[24] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *ISSTA*, pages 169–180, 2019.

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

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

[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. 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)*, 2020.

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

[29] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *FSE*, FSE'15, pages 166–178, 2015.

[30] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *POPL*, POPL'16, pages 298–312, 2016.

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

[32] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. Can automated program repair refine fault localization? a unified debugging approach. In *ISSTA*, 2020. to appear.

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

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

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

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

[37] Martin Monperrus. The living review on automated program repair. Technical report, 2020.

[38] Martin Monperrus, Simon Urli, Thomas Durieux, Matias Martinez, Benoit Baudry, and Lionel Seinturier. Repairnator patches programs automatically. *Ubiquity*, 2019(July), July 2019. https://doi.org/10.1145/3349589 doi:10.1145/3349589.

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

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

[41] Objectweb. Asm bytecode manipulation framework, 2020. Accessed: August, 2020. URL: https://asm.ow2.io/.

[42] Oracle. The java language specification, 2020. Accessed: August, 2020. URL: https://docs.oracle.com/javase/specs/.

[43] K Pearson. Notes on regression and inheritance in the case of two parents proceedings of the royal society of london, 58, 240-242, 1895.

[44] Undo Software. Increasing software development productivity with reversible debugging. https://bit.ly/3c8ccbn, 2016. Accessed: August, 2020.

[45] DCEVM Team. Dynamic code evolution vm for java, 2020. Accessed: August, 2020. URL: http://dcevm.github.io/.

[46] JRebel Team. Jrebel, 2020. Accessed: August, 2020. URL: https://bit.ly/3fQox6Y.

[47] UniAPR team. Uniapr webpage, 2020. Accessed: August, 2020. URL: https://github.com/UniAPR/UniAPR.

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

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

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

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