Inferring Bug Signatures to Detect Real Bugs

Hao Zhong, *Member, IEEE*, Xiaoyin Wang, *Member, IEEE*, and Hong Mei, *Fellow, IEEE*

Abstract—Static tools like Findbugs allow their users to manually define bug patterns, so they can detect more types of bugs, but due to the complexity and variety of programs, it is difficult to manually enumerate all bug patterns, especially for those related to API usages or project-specific rules. Therefore, existing bug-detection tools (*e.g.*, Findbugs) based on manual bug patterns are insufficient in detecting many bugs. Meanwhile, with the rapid development of software, many past bug fixes accumulate in software version histories. These bug fixes contain valuable samples of illegal coding practices. The gap between existing bug samples and well-defined bug patterns motivates our research. In the literature, researchers have explored techniques on learning bug signatures from existing bugs, and a bug signature is defined as a set of program elements explaining the cause/effect of the bug. However, due to various limitations, existing approaches cannot analyze past bug fixes in large scale, and to the best of our knowledge, no previously unknown bugs were ever reported by their work. The major challenge to automatically analyze past bug fixes is that, bug-inducing inputs are typically not recorded, and many bug fixes are partial programs that have compilation errors. As a result, for most bugs in the version history, it is infeasible to reproduce them for dynamic analysis or to feed buggy/fixed code directly into static analysis tools which mostly depend on compilable complete programs. In this paper, we propose an approach, called DEPA, that extracts bug signatures based on accurate partial-code analysis of bug fixes. With its support, we conduct the first large scale evaluation on 6,048 past bug fixes collected from four popular Apache projects. In particular, we use DEPA to infer bug signatures from these fixes, and to check the latest versions of the four projects with the inferred bug signatures. Our results show that DEPA detected 27 unique previously unknown bugs in total, including at least one bug from each project. These bugs are not detected by their developers nor other researchers. Among them, three of our reported bugs are already confirmed and repaired by their developers. Furthermore, our results show that the state-of-the-art tools detected only two of our found bugs, and our filtering techniques improve our precision from 25.5% to 51.5%.

Index Terms—bug fix, bug signature, partial code analysis.

1 Introduction

Static tools are typically shipped with predefined patterns of their target bugs. As it is rather difficult to enumerate all bug patterns, some tools (*e.g.*, Findbugs [35]) allow programmers to customize bug patterns, but it may be difficult for even experienced programmers to define some patterns. For example, Zhong and Su [94] shows that a notable portion of bugs are related to wrong API usages, but a developer can hardly define all illegal patterns because she may not know erroneous usages of the APIs if she typically uses them correctly.

With the rapid development of software, open source repositories have accumulated many bugs and their fixes. Researchers (*e.g.*, [36]) have proposed various approaches that infer bug signatures from bug fixes. Here, as defined by Sun and Khoo [82], a bug signature is a set of program elements that explain the cause or the effect of a bug, and a bug signature can be easily translated to bug patterns of existing static tools. The prior approaches roughly fall into two categories: (1) dynamic approaches [36], [82] mining bug signatures from buggy traces; and (2) static approaches [53], [70] extracting bug signatures from buggy source files. Although the research topic has been intensively studied, two challenges still remain open and less explored:

Challenge I: The large-scale in-depth analysis. Inferring bug signatures effectively requires in-depth analysis of many bugs to understand their common semantic features (*e.g.*, code dependencies). The prior approaches perform either in-depth analysis on a small number of known bug fixes, or shallow analysis (*e.g.*, syntax analysis) on a large number of bug fixes. For the former, manual [18], [43], dynamic [24], and static [53], [82] approaches have inferred bug signatures from only a limited number of bugs. For the latter, the tools (*e.g.*, *spdiff* [17]) are built on shallow code analyses, which considers mainly abstract syntax tree instead of code bindings and dependency. The compilable code is a prerequisite for most semantic code analysis, but Tufano *et al.* [11], [85] show that only 38% commits are compilable. In their evaluations, the former approaches suffer from generating too few bug signatures to catch new bugs, whereas the latter approaches generate over-specific bug signatures.

Challenge II: The differences between known and new bugs. Even after bug signatures are inferred, it is challenging to use them to detect new bugs, because the details of a known bug may never appear in new bugs. As such details can be encoded in bug signatures, matching exact matches of bug signatures can be less effective to detect new bugs, which explains why some prior approaches [53], [70], [82] fail to detect real bugs. To handle this issue, the other prior approaches [53], [70], [82] build abstractions (*e.g.*, dependency graphs) as bug signatures, and match source files at such abstractions. Although it hides trivial differences (*e.g.*, blank spaces), a piece of buggy code can still be ignored, since it includes different code elements.

Challenge III: The locations of buggy snippets. Zhong and Su [94] show that most bug fixes modify several source files, and even if a file is modified, typically, only a small portion of code lines are modified. If we compare a whole buggy file with source files, many detected similar code snippets can be false positives.

- *Hao Zhong and Hong Mei are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. E-mail: zhonghao@sjtu.edu.cn*
- *Xiaoyin Wang is with Department of Computer Science, University of Texas at San Antonio, USA.*

Manuscript received April 19, 2005; revised August 26, 2015.

Due to the above challenges, the prior approaches have difficulties in analyzing large scale bug fixes. For example, in total, CBCD [53] analyzed only 5 Git bugs, 14 PostgreSQL bugs, and 34 Linux bugs when it is evaluated. In addition, researchers often evaluated their approaches only on benchmarks, where bugs are already known. For example, Sun and Khoo [82] evaluated their approach on the Siemens benchmark [39]. In this benchmark, all bugs are manually constructed. Pendlebury *et al.* [69] show that the settings of benchmarks can be biased, and the effectiveness of an approach can be significantly reduced when the settings are adjusted. To understand the true effectiveness of an approach, we must evaluate it under the real setting, where code is the latest and bugs are unknown. However, to the best of our knowledge, in this research field, no prior approaches have ever been evaluated under the real setting, and no real unknown bugs are ever reported.

Our solution. In this paper, we propose a novel approach, called DEPA (Detecting bugs with Past fixes), that detects bugs with past fixes. To overcome the first challenge, we build DEPA upon GRAPA [95], which enables the analysis on thousands of bug fixes that are not compilable. To overcome the second and the third challenges, we build method graphs to represent illegal usages, and use the Hungarian algorithm [48] to locate buggy snippets. Different from the prior approaches [53], [70], [82], our inferred bug signatures are in the format of method graphs (Definition 1), and they focus on method invocations and their dependencies.

Although mining bug signatures is intensively studied, compared with the prior approaches (*e.g.*, [53], [70], [82]), DEPA makes the following major contributions:

- The first approach that is able to infer bug signatures from large scale bug fixes. In their evaluations, the prior approaches (*e.g.*, [53], [70], [82]) analyzed fewer than one hundred bug fixes. In comparison, DEPA has already analyzed more than six thousand real fixes.
- The first approach that uses method graphs to denote bug signatures. The prior approaches [53], [70], [82] use dependency graphs, but DEPA uses method graphs to encode bug signatures. We choose this granularity of methods, since researchers have detected many bugs via mining specs and most mined specs define method usages [73].
- An approach that leverages the Hungarian algorithm [48] in identifying buggy code snippets (Sections 3.1.2); inferring bug signatures (Section3.1.3); and detecting new bugs (Section 3.2). Although the algorithm is classical, we are the first to introduce it to the above applications, and some of its benefits are illustrated in Sections 4.5 and 4.6.

In our evaluation, DEPA inferred bug signatures from 6,048 bug fixes, which were collected from four popular open source projects. Our results are summarized as follows:

- DEPA detected 65 new bugs from the latest versions of real projects. Their programmers were unaware of these bugs, until we reported them. To the best of our knowledge, this is the first time that researchers report real new bugs when they use their inferred bug signatures to detect bugs.
- Our detected bugs cover various categories such as null pointer accesses, wrong method (API) calls, wrong exceptions, overflow, and resource leaks. As most of these bugs are caused by API or project related issues, Findbugs and PMD detected only two of our found bugs. DEPA enriches the bug patterns through its mined bug signatures.

```
1 protected ... getWriteDirectory(long writeSize){
2   ...
3   directory = getDirectories().getWriteableLocation(...);
4   if (directory == null)
5     throw new RuntimeException("Insufficient disk space to
          write " + writeSize + " bytes");
6   return directory;
7 }
```

(a) The buggy code

```
1 protected ... getWriteDirectory(long writeSize){
2   ...
3   directory = getDirectories().getWriteableLocation(...);
4   if (directory == null)
5      throw new FSWriteError(new IOException("Insufficient
           disk space to write " + writeSize + " bytes"), "
           ");
6   return directory;
7 }
```

(b) The fixed code

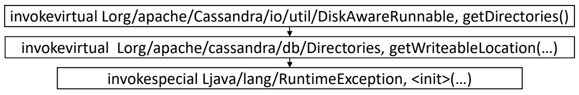
Fig. 1: CASSANDRA-11448

- DEPA ranks real bugs on the top of its bug list, and its precision (51.5%) is higher than the prior studies (17.2% as reported by Legunsen *et al.* [51]).
- A detailed analysis of DEPA. Our results show that (1) our filtering techniques improve our precision from 25.5% to 51.5%, which is comparable or even better than other static approaches; and (2) empirically, we find that the best size parameter is four. Here, the parameter determines the sizes of inferred bug signatures.

2 MOTIVATING EXAMPLE

Cassandra [1] is a popular distributed database. According to its documentation [5], when the disk fails, Cassandra allows its users to choose one of the two policies: `stop` and `best_effort`. In particular, when a disk error occurs, `stop` will shut down the node, and `best_effort` will only blacklist the failed disk. However, a user submitted a bug report [2] saying that when a disk is full, neither the disk is blacklisted nor the node is shut down. Instead, Cassandra throws an exception, when a disk is full. The programmers of Cassandra inspected the code throwing exceptions, and Figure 1a shows the buggy code. In particular, Line 3 tries to retrieve a writable location. If the disk is full, Line 5 will throw a `RuntimeException`. However, the exception does not trigger the disk failure policy. Figure 1b shows the fixed code. In Line 5, instead of an exception, it throws a `FSWriteError`. Cassandra implements a mechanism to handle this type of errors, and it will trigger the disk failure policy.

From the bug fix in Figure 1, DEPA inferred the bug signature in Figure 2a. In a bug signature, a node denotes a method invocation, and an edge denotes either a data dependency or a control dependency. For example, the bug signature in Figure 2a defines that the method call chain of `getDirectories()` → `getWriteableLocation()` → `RuntimeException,<init>()` that indicates a bug. DEPA checked the latest code of Cassandra with the bug signature, and it found that the bug was not fully fixed. For example, Figure 2b shows one of our newly found bugs [3]. As the code in Figure 1a does and as defined by Figure 2a, the code in Figure 2b also throws `RuntimeException`, when it fails to retrieve writable locations. However, as explained in CASSANDRA-11448, the thrown exceptions will not trigger the disk failure policy of Cassandra.


invokevirtual Lorg/apache/Cassandra/io/util/DiskAwareRunnable, getDirectories()
invokevirtual Lorg/apache/cassandra/db/Directories, getWriteableLocation(...)
invokespecial Ljava/lang/RuntimeException, <init>(...)

(a) The inferred bug signature

```java
public ... getWriteDirectory(...) {
  ...
  d = getDirectories().getWriteableLocation(...);
  if (d == null)
    throw new RuntimeException(...);
  return d;
}
```

(b) The buggy code in DirObjectFactoryHelper.java

Fig. 2: Our reported bug

Detecting the bug in Figure 2b requires the knowledge on how Cassandra handles its disk failure policy. As it needs to set up a cluster to reproduce the bug, it is even difficult for outsiders like us to trigger out-of-space disk errors. As a result, we did not submit test cases in our bug report. Fortunately, the programmers of Cassandra took our report bug seriously. They have implemented the test cases that trigger the buggy behaviors, and implemented patches for this bug.

3 Approach

Figure 3 shows the overview of our approach. It consists of two major steps: (1) inferring bug signatures (Section 3.1), and (2) detecting bugs (Section 3.2).

3.1 Generation of Bug Signatures

3.1.1 *Bug Analysis with* GRAPA

DEPA uses GRAPA to build SDGs from bug fixes. GRAPA [95] is a recent advancement in partial code analysis. Given a partial program of a project, GRAPA extracts its called code names, and by matching these names with declared code names in all the released versions of the project, it locates the best approximate version of the partial program. The partial program can contain code names that cannot be resolved by the approximate version, because a fix often appears between two released versions. To handle the problem, GRAPA extends the repair strategies of PPA [28], and thus can resolve more code names. After all the code names are resolved, GRAPA enables WALA [12] to build system Dependency Graphs (SDGs) for the partial program, as it builds SDGs for complete programs. We set WALA to extract both data and control dependencies. As a result, an edge can denote either a data dependency and a control dependency.

With GRAPA, researchers [87], [93] have analyzed thousands of bug fixes to understand their code changes. However, neither GRAPA nor its follow-up studies [87], [93] include any tools that can infer bug signatures or detect bugs. To use GRAPA for bug signature construction, for each historical bug fix, DEPA collects all changed source files and feeds both buggy and fixed versions of these files into GRAPA to generate a pair of system dependency graphs. This graph pair is then simplified and compared for bug signature generation.

DEPA infers bug signatures in the format of method graphs:

Definition 1. A method graph is defined as $g = \langle V, E\rangle$, where V is a set of method invocations, and $E \subseteq V \times V$ is a set of dependencies: A $\langle s_1, s_2\rangle \in E$ edge denotes a direct dependency or transitive dependency from s_1 to s_2.

From an SDG G, DEPA extracts a method graph M whose nodes denote all method invocation nodes in G, and whose edges denote the transitive reachable relationship between method invocation nodes in G. This idea is borrowed from mining specifications (see Section 6 for details) where temporal or other constraints are mined for legal usage of method invocations. However, as DEPA extracts illegal usages which are relatively rare, we do not apply frequency-based mining which is commonly used in mining specifications. This technical choice is in accordance with the prior approaches for mining bug signatures (*e.g.*, [53]). Based on the difference between the buggy and the fixed method graphs, DEPA further extracts a subgraph from the buggy method graph to define buggy behaviors. As a result, our bug signature is a directed graph where each node denotes a method invocation and each edge denotes a control or data dependency between two nodes.

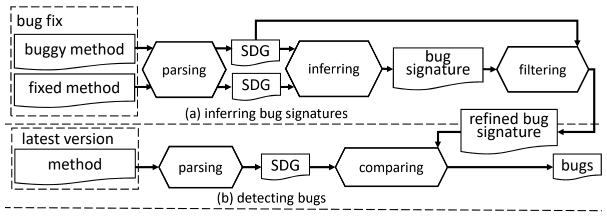


Fig. 3: The overview of DEPA

DEPA's bug signature generation process consists of three steps. First, for each bug fix, DEPA constructs the mapping of methods between the bug version and the fixed version so that the revised methods can be extracted. Second, DEPA infers a bug signature from each revised method by mapping the SDGs generated from the buggy version and the fixed version of the method. The added or removed methods are ignored in this step as DEPA focuses on inner-method bug signatures. Third, DEPA filters the inferred bug signatures based on a set of rules to remove irrelevant bug signatures.

3.1.2 *Construction of Mappings*

Each bug fix contains a set of buggy files and a set of fixed files. DEPA identifies the mappings between a buggy file and its fixed file by their file names. Shi *et al.* [81] show that method names are less stable than file names. In the optimization research, the assignment problem [64] is to assign agents to their proper tasks, and the Hungarian algorithm [48] is a classical algorithm that solves the assignment problem. To handle modified method names, DEPA uses the Hungarian algorithm to build the mappings between the methods in the buggy version and the methods in the fixed version. Initially, the algorithm calculates the distances between buggy methods and fixed methods. We define the distance as the Levenshtein edit distance of their full method names.

3.1.3 *Inferring Bug Signatures*

For each pair of mapped methods, DEPA compares their plain text representations to determine whether the method is modified. If the method is modified in the new version, DEPA uses GRAPA [95] to build two SDGs for the method in the buggy version and the method in the fixed version, respectively. When building the pair of SDGs for a modified method, DEPA conducts an intra-procedure analysis. This strategy does not lose any modifications, in that DEPA compares all modified methods.

After SDGs are built, for each pair of SDGs, DEPA uses the Hungarian algorithm in Section 3.1.2 to build the mappings between their nodes. In the Hungarian algorithm, the distance of two nodes (m and n) is defined as follows:

Algorithm 1: Type Distance Algorithm

```
Input:
    $lt$ is the hastable of the buggy code (node type→ occurrence)
    $rt$ is the hastable of the fixed code (node type→ occurrence)
Output:
    $dis$ is the distance between the two tables
 1: if lt.isEmpty and rt.isEmpty then
 2:     $dis \leftarrow 0$
 3: else if (lt.isEmpty and rt.isNonEmpty) or (lt.isNonEmpty and rt.isEmpty) then
 4:     $dis \leftarrow 1$
 5: else
 6:     $keys \leftarrow lt.keys \cup rt.keys$
 7:     for $key \in keys$ do
 8:         $lv \leftarrow lt.get(key)$
 9:         $rv \leftarrow rt.get(key)$
10:         $dis \leftarrow \frac{|lv-rv|}{max(lv,rv)} + dis$
11:     end for
12:     $dis \leftarrow dis/keys.size$
13: end if
```

$$dis(m,n) = \frac{1}{3}(d_{name}(m,n) + d^{in}_{type}(m,n) + d^{out}_{type}(m,n)) \tag{1}$$

Here, $d_{name}(m,n)$ returns the name distance:

$$d_{name}(m,n) = \begin{cases} d_{code}(m,n), & m, n \text{ are variable accesses} \\ & \text{or method invocations} \\ d_{type}(m,n), & \text{otherwise} \end{cases} \tag{2}$$

As introduced in its manual [10], WALA encodes node labels of SDGs in a language that is close to JVM bytecode. The JVM specification [57] defines four instructions to access fields or class variables (*i.e.*, `getfield`, `putfield`, `getstatic`, and `putstatic`), and five instructions to call a method (*i.e.*, `invokedynamic`, `invokeinterface`, `invokespecial`, `invokestatic`, and `invokevirtual`). For node labels of variable accesses, d_{code} extracts only full names of their types for comparison. It ignores variable names, since they are ad hoc. For node labels of method invocations, d_{code} extracts full method names for comparison. The names of other nodes are less informative. For example, as a `phi` node is a special statement that is inserted by compilers [10]. For these nodes, d_{type} extracts their types for comparison. After extraction, DEPA uses the Levenshtein edit distance [9] to measure distances between string values.

For each node, d^{in} and d^{out} denote its incoming and outgoing neighbors, respectively. When comparing two nodes, the mappings of their neighbors are not determined. As a result, it is infeasible to calculate the accurate cost between their neighbors. Instead, DEPA calculates the approximate cost with Algorithm 1. In the algorithm, for two sets of neighbors, DEPA extracts two hash tables to denote node types and their occurrences. Here, as each node denotes an instruction, we use instruction types to denote node types. The JVM spec defines the complete list of instruction types. Algorithm 1 takes the two hash tables as its inputs, and it calculates the common instructions of two sets of neighbors.

After the distance function is defined, the Hungarian Algorithm is able to build the optimal node mappings between two graphs. We consider that a node is modified when Equation 1 calculates that the distance between the node and its mapped node is nonzero.

TABLE 1: The categories of false alarms.

Project	debug	fun.	global	element	weak	subtle
aries	19	4	13	28	20	4
mahout	2	10	31	7	26	5
derby	23	16	20	3	23	2
cassandra	0	39	1	4	32	4
total	44	69	65	42	101	15

From the modified nodes of an SDG, DEPA extracts all method invocations. For two method invocations (s and t), if there exists at least a path and no path goes through a method invocation, DEPA adds an edge from s to t. In this way, DEPA builds method graphs.

To extract bug signatures from large SDGs, DEPA allows users to set a *size* parameter. In each iteration, DEPA includes the n-depth neighbors of modified nodes into analysis, until the produced bug signature has more nodes than the size parameter. Here, it considers both data dependencies and control dependencies. If it cannot produce such a bug signature after all nodes are included, it produces the maximized graph as the bug signature.

3.1.4 Filtering Bug Signatures

Figure 7 shows that directly using bug signatures to detect bugs results about 80% false alarms. After manual inspection, we found that some bug signatures are superficial. For example, when fixing bugs, programmers can modify code to print logs. A bug signature can be inferred from such a modification, but it does not indicate any bugs. We classified all the found superficial bug signatures, and designed corresponding filtering techniques. Table 1 shows the results. Column "debug" shows false alarms that are related to debugging code, and we define a debug filter for these false alarms. Column "fun." lists false alarms that are due to different functions, and we define a common API filter for these false alarms. Column "global" lists false alarms that are related to global usages, and we define a global change filter for these false alarms. Column "element" lists false alarms that are related to code element other than method invocations, and we define a size filter for these false alarms. Column "weak" denotes false alarms that are related to weak bug signatures, and we define a weak method filter for these false alarms. The details of these filters are as follows:

1) **Debug filter.** DEPA filters a pair of SDGs, if their modified methods are named as "debug", "info", "print", "error", or "println". For example, to fix ARIES-127, programmers added debugging code to a `catch` clause:

```
} catch (Exception e) {
-  //TODO log this
+  _logger.error("There was an error ...);}
```

Neither the original nor modified methods have bugs. Programmers added the `_logger.error` method to save debug information. As this type of modifications do not indicate bugs, DEPA filters its SDGs.

2) **Global change filter.** DEPA filters two SDGs of a buggy class, if their modified nodes access the same field. For example, to fix DERBY-5312, programmers made relevant modifications to multiple methods as follows:

```
+ private ContainerKey currentIdentity;
synchronized void createContainer( ...{
...
+ currentIdentity = newIdentity;
...}
private boolean openContainerMinion( ... {
 ...
+ idAPriori = currentIdentity;
...}
```

As we build an SDG for each method and a single SDG is insufficient to describe the buggy behavior of the above bug, DEPA filters the above SDGs.

3) **Size filter.** DEPA filters bug signatures whose method invocations are fewer than two. For example, ARIES-878 fixed a concurrency bug:

```
- synchronized (list) {
+ new Thread() {
+   public void run() {
        for (ManagedObject cm : list) {
            cm.updated(props);}
+   }
+ }.start()
```

As WALA ignores concurrency code elements, it builds an SDG with only one node for the above code. DEPA filters this SDG, because it is not meaningful.

4) **Common API filter.** We target at detecting bugs in released code of popular projects, and such code is implemented by experienced programmers. We notice that some APIs (*e.g.*, the `java.lang` package) are widely used. We believe that experienced programmers are familiar with their usages, so they are unlikely to introduce bugs, when using such common APIs. In particular, if an inferred bug signature invokes methods that are declared only by the `java.lang` package, the `java.math` package, and the `java.util` package, DEPA filters the bug signature. One example is the enumeration of a list whose type is `java.util.List`, and corresponding code snippets typically call the methods such as `iterator`, `hasNext`, and `next`. Although the enumeration is widely used and can appear in buggy code, it often does not indicate a bug. This filter can ignore some true bugs, but it reduces false alarms.

5) **Weak method filter.** We notice that it is unlikely to infer meaningful bug signatures from some methods. For example, some classes override the `toString` method of the `java.lang.Object` class. Although programmers can revise the method to log more details when fixing bugs, its modifications do not indicate buggy behaviors. DEPA filters SDGs that are built from such methods.

Our filters are not exhaustive, and some false alarms are difficult to be removed by filters. In particular, Column "subtle" lists false alarms that are related to subtle differences. Although their graphs are similar, their subtle differences in code determine that they are not true bugs. We still cannot define an effective filter for this type of false alarms. However, Section 4.6 shows that our filters are able to improve the precision from 25.5% to 51.5%. Our result shows that learning from data is promising to reduce false alarms. Section 5 further discusses this issue.

3.2 Detecting Bugs

After bug signatures are inferred, a detection tool typically determines the matches of a bug signature as suspected bugs. As the existing approaches [53], [70], [82] locate identical matches, they can lose true bugs with minor differences. Instead of exact matches, given a bug signature, DEPA matches its graph with the graphs built from to-check code, and its matching technique allows minor differences.

The matching technique of DEPA is also built on the Hungarian algorithm as introduced in Section 3.1.2. When comparing a method graph, DEPA requires that bug signatures shall have fewer nodes than the method graph has. For a method graph and a bug signature, DEPA builds their node mappings, and the distance function is defined in Equation 1. After the mappings (M) are determined, we calculate the distance value between the method graph and the bug signature as follow:

Algorithm 2: Structure Distance Algorithm

```
Input:
    $t$ is the hastable that defines mappings
    $lg$ is the left-side graph
    $rg$ is the right-side graph
Output:
    $dis$ is the structure distance
1: $count \leftarrow 0$ // The total common edges.
2: $total \leftarrow 0$ // The total edges of the left side.
3: for $l_1 \in m.keys$ do
4:     for $l_2 \in m.keys$ do
5:         if $lg.existEdge(l_1, l_2)$ then
6:             $total \leftarrow total + 1$
7:             $r_1 \leftarrow lt.get(l_1)$
8:             $r_2 \leftarrow rt.get(l_2)$
9:             if $rg.existEdge(r_1, r_2)$ then
10:                $count \leftarrow count + 1$
11:            end if
12:        end if
13:    end for
14: end for
15: if total is zero then
16:     $dis \leftarrow 0$
17: else
18:     $dis \leftarrow 1 - \frac{count}{total}$
19: end if
```

$$d_m(M) = \frac{1}{3}(d_n + \frac{\sum dis(m.source, m.target)}{|M|} + d_s(M)) \tag{3}$$

Here, $m \in M$; $|M|$ is the cardinality of M; and $m.source$ and $m.target$ denote the source node and the target node of the m mapping, respectively. In this equation, d_n denotes the Levenshtein edit distance between the two enclosure method names where the method graph and the bug signature are extracted; $dis()$ is defined in Equation 1; and Algorithm 2 shows the details of $dis_s()$. For a given mapping set, Algorithm 2 calculates edges that appear in the bug signature, and edges that appear in both the bug signature and the method graph. The distance is calculated by dividing the two values. As a bug signature is extracted from a pair of buggy method and fixed method, each bug signature has only an enclosure method. When comparing with source files, DEPA checks a method each time. As a result, d_n compares only two method names, although the two methods can call more methods.

In the literature, researchers have proposed various approaches to compare two pieces of code. The prior approaches [33], [44] typically generate edit scripts to denote their changes. Instead of such edit scripts, DEPA produces a distance to denote to their similarity, which is useful to match code with minor differences. In particular, for a method graph and a set of bug signatures, DEPA produces an increasing-order list based on Equation 3. Each item of the list contains a suspicious value, a bug signature, and corresponding buggy locations. Programmers can manually check the list for bugs. For example, when we use DEPA to detect bugs in the code of Figure 2b, it finds that the distance between the graph from Figure 2b and the graph from Figure 1a is zero. Although the code snippets in Figures 2b and 1a are different, after DEPA

TABLE 2: Subjects.

Name	Past fix		The latest version		
	fix	method	file	method	LOC
aries	542	1,582	2,027	4,717	264,364
mahout	494	2,042	1,181	5,077	172,146
derby	1,604	4,966	2,764	18,568	1,207,970
cassandra	3,408	11,283	2,068	8,046	448,317
total	6,048	19,873	8,040	36,408	2,092,797

builds a method graph from Figure 2b, a subgraph of the method graph is identical with the bug signature in Figure 2a. As a result, other tools (*e.g.*, CBCD) cannot detect this bug, since such tools require identical code snippets.

4 EVALUATION

In this section, we present our research questions (Section 4.1), our data set (Section 4.2), and our results (Sections 4.3 to 4.7).

4.1 Research Question

In our evaluation, we explore the following research questions:

(RQ1) How effectively does DEPA detect bugs in the latest releases of real projects (Section 4.3)?

(RQ2) To what degree is DEPA complementary to static tools such Findbugs and PMD (Section 4.4)?

(RQ3) To what degree does DEPA improve the idea of detecting clones of buggy snippets as bugs (Section 4.5)

(RQ4) What is the impact of our filtering techniques on detecting bugs (Section 4.6)?

(RQ5) How does the size parameter influence the effective of DEPA (Section 4.7)?

RQ1 concerns the overall effectiveness of DEPA. From four popular open source projects, DEPA detected 65 previously unknown bugs (27 are unique) and 38 known bugs in total.

RQ2 concerns the significance of our inferred bug signatures (*i.e.*, whether existing tools already defined their corresponding rule patterns). Our results show that the two state-of-the-art tools (Findbugs and PMD) defines only two of our detected bugs, and they missed all the other our detected bugs.

RQ3 concerns the improvement over detecting clones of buggy snippets as bugs. We used the combination of CCFinderSW and git diff to compare buggy files and source files as we did in RQ1. Although the combined approach detected several bugs, its effectiveness is much poorer than DEPA.

RQ4 concerns the impact of our filters and our precision. Our results show that our filtering techniques improved the precision from 25.5% to 51.5%. We believe this is an acceptable rate, as a study [18] in Google shows that tools with averagely 8% (for high priority bugs, before prioritization) or 36% (for high priority bugs, after prioritization [46]) can be practically very useful.

RQ5 concerns the impact of our size parameter. Our results show that four is the empirical best value. In the other RQs, we set the parameter as four.

4.2 Dataset

The eGit [4] tool is an open source Eclipse plugin that supports Git version control system. We extend the eGit tool to extract commits. Each commit has a message. For example, in the `e424950` commit of aries has a message, "[ARIES-788] Possible NPE when destroying the extender". As most Apache projects carefully write issue number to commit messages [94], our extended eGit extracts such issue number from commit messages ("ARIES-788" in this example). We implemented a web crawler to extract issue trackers. The web crawler is based on XPath [21], which is easy to customized according to the different styles of reported issues. For each commit, our extended eGit compares its issue number with reported issues to determine whether it is a bug fix. If a commit is confirmed as a bug fix, our extended eGit checks out its buggy files and fixed files, and stores them into two separate directories.

Table 2 shows the subjects. Column "Past fix" lists our collected past fixes. For this column, Subcolumn "fix" lists number of fixes, and Subcolumn "method" lists number of modified method pairs. From each pair, DEPA tries to extract a bug signature. Column "The latest version" lists the latest versions of these projects. We check out these versions from the Apache Git [6]. For this column, Subcolumn "file" lists number of source files; Subcolumn "method" lists number of methods; and Subcolumn "LOC" lists lines of code. We checked out the past fixes and the trunks in March, 2017.

In total, we analyzed 6,048 bug fixes, which are already much more than what the prior approaches did, and it is feasible to collect more subjects. Tian *et al.* [84] and Wu *et al.* [90] propose approaches that identify bug fixes even if issue numbers are not written to commit messages. Their approaches allow extracting bug fixes from more sources than Apache projects (*e.g.*, Linux). However, as their approaches can identify false bug fixes, it will take extra effort to manually identify true bug fixes.

4.3 RQ1. Overall Effectiveness

4.3.1 Setting

For each project in Table 2, we used DEPA to infer bug signatures from its past fixes, and to detect bugs in its latest version. For each method, DEPA produced a list of suspected bugs. For each project, we merged the output of all its methods, and inspected the list to identify bugs. Furthermore, we analyzed their types and distribution. We use precisions as our measurement, but do not calculate recalls, since it is infeasible to identify all the bugs of real code. We inspected only the top 50 bugs, because developers usually inspect only the most severe warnings reported by bug detection tools [18]. A bug has a buggy location in the source file and a bug signature. We follow the following protocol to determine whether the location is a true bug. First, we compare the buggy file and the fixed file of the bug signature, and locate the modification of the bug signature. Second, we inspect the modification to learn whether it fixes a bug. If they have bug reports, we further read their bug reports to deepen our understanding on this modification. This step removes superficial modifications that do not fix bugs. If we determine a modification fixes a bug, we analyze whether buggy location has similar problems and whether the modification applies on the buggy location. If they do, we determine the buggy location as a true bug. We list the source files of our true bugs in Section 4.3.2, and we report some bugs to their developers.

In this research question, we use Apache projects as our subjects. As Apache projects are real code and under careful maintenance, their bugs are limited. As DEPA learns bug signatures from past fixes, it mainly detects recurring bugs. It is already noteworthy that a real project can have up to 50 recurring bugs. As a result, inspecting more bugs can lead to more false positives, but we believe that this is a nature of bugs.

TABLE 3: Overall effectiveness.

Project	UB	UUB	KB	S	SD	Precision
aries	18	8	3	16	13	42.0%
mahout	13	5	19	2	16	64.0%
derby	17	5	12	12	9	58.0%
cassandra	17	9	4	8	21	42.0%
total	65	27	38	38	59	51.5%

UB: previously unknown bug; UUB: unique UBs; KB: known bug; S: similar but not bugs; DS: dissimilar and not bugs.
Precision = $\frac{UB+KB}{UB+KB+S+DS}$.

TABLE 4: The results with filters off.

Project	UB	UUB	KB	S	DS	Precision
aries	13	6	5	30	2	36.0%
mahout	2	2	10	38	0	24.0%
derby	3	2	6	41	0	18.0%
cassandra	2	2	10	37	1	24.0%
total	20	12	31	146	3	24.5%

4.3.2 *Result*

Table 3 shows the overall result. Section 2 introduces a bug in the `BlueprintURLContext.java`. Besides this file, DEPA detected the same bug in three other files. In Column “UB”, we count the above files as four previously unknown bugs, and in Column “UUB”, we count them as a unique bug. In total, DEPA detected 65 previously unknown bugs (27 are unique) and 38 known bugs.

The bug in Section 2 throws wrong exceptions. We investigated our detected bugs, and found bugs with more symptoms:

1) **Wrong method calls.** We find that programmers can call wrong methods. For example, we reported ARIES-1703, since a method has the following code:

```
Map<String, Object> result = new HashMap<String,
    Object>(...);
result.put(NAMESPACE, clause.getPath());
result.put(ATTRIBUTE_BUNDLE_VERSION, version.getValue
    ());
```

The last line calls the `getValue()` method to get the version. The method call is wrong, since the correct method is `getVersion()`. DEPA detected this bug, since a past bug, ARIES-1453, fixed a similar problem.

2) **Wrong API calls.** We find that programmers sometimes call wrong APIs. For example, in ARIES-1705, we reported that a method can use a wrong API class, `ArrayList`, to store bundles. DEPA detected this bug, since in a past bug (ARIES-464), programmers fixed a similar bug in another method:

```
_logger.debug(LOG_ENTRY, "...", new Object[]{content
    });
- List<ImportedBundle> result = new ArrayList<
    ImportedBundle>();
+ Set<ImportedBundle> result = new HashSet<
    ImportedBundle>();
```

In the above fix, `ArrayList` is replaced with `Hashset` to remove redundant bundles, since the latter API class does not store duplicated items.

3) **Overflow.** We find that it can cause overflow, if an incorrect type is used to store return values. For example, in MAHOUT-1958, we reported that the following code can overflow:

```
int preferring2 = dataModel.
    getNumUsersWithPreferenceFor(...);
int intersection = dataModel.
    getNumUsersWithPreferenceFor(...);
```

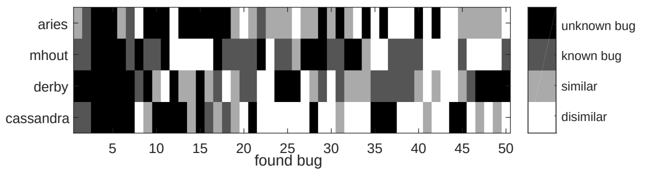


Fig. 4: The distribution of detected bugs

Both statements assign return `long` values to `int` variables. DEPA detected the bug, since a similar previous bug, MAHOUT-738, occurs in another source file.

4) **Unfinished migration.** We find that programmers can fail to update all call sites, when they upgrade to new APIs. For example, in MAHOUT-1427, programmers start migrating from old APIs to new ones. For example, in several files, the `MultipleOutputs` class was replaced to its new version. However, DEPA detected that in other files such as `MatrixMultiplicationJob`, these APIs are not migrated. We reported this issue in MAHOUT-1961.

5) **Resource leak.** We find that programmers can leave exceptions unhandled. For example, in DERBY-6927, we reported the following buggy code:

```
ResultSet rs = conn.getMetaData().getSchemas();
boolean schemaFound = false;
while (rs.next() && !schemaFound)
    schemaFound = schemaName.equals(rs.getString("
        TABLE_SCHEM"));
s.close();
```

In the above code, if the iteration throws exceptions, the `ResultSet` will never be closed. Indeed, a previous bug (DERBY-6297) is similar. The patch is as follow:

```
boolean found=false;
ResultSet result = conn.getMetaData().getSchemas();
+try{
    while(result.next()){
      if(result.getString(1).equals(schema)){
        found=true;
        break; }}
+} finally { result.close();}
return found;
```

Although the above buggy code is different from our reported code, DEPA detected this bug, since their method graphs are similar.

6) **Less-optimized code**. In ARIES-1730, we reported that the following code is less optimized:

```
}finally{
  if(inputStream != null){
    try{
      inputStream.close();
    }catch(IOException ioe){}}}
```

Programmers have implemented the `IOUtils.close` method to handle unclosed resources in `finally` causes. DEPA detected this bug, since a previous bug, ARIES-622, fixed a similar problem.
DEPA also detected known bugs. For example, in DERBY-6946, we find a wrong check:

```
if(rows < 0||(this.getMaxRows()!=0&&rows>this.
    getMaxRows())){ throw newSQLException(...) }
```

Its correct code shall be:

```
if(rows < 0){
    throw newSQLException(...) }
```

DEPA detected the above bug, since DERBY-3573 fixed a similar bug. However, after we reported the bug, their programmers told us that the latest code has fixed the bug already. We notice

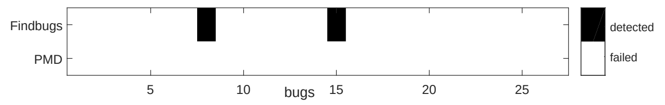


Fig. 5: The results of Findbugs and PMD

that most previous approaches conduct their evaluations on known bugs. For example, Sun and Khoo [82] conduct their evaluation on the Siemens benchmark. In the benchmark, all bugs are known, and their test cases are available. Although detecting known bugs can show the effectiveness of a proposed approach, programmers may be less interested in detecting known bugs, since they have been already repaired in the latest versions. We notice that the repairs of a bug can be similar. As a result, it may be feasible to filter known bugs, if we further compare such suspicious locations with SDGs of fixed code. We plan to explore this issue in our future work.

In Table 3, Column "S" lists false alarms that are similar to bug signatures. For example, we find that a bug signature is mined from the `JavaDriverClient.connect` method:

```
- ... clusterBuilder = Cluster.builder().addContactPoint
(host).withPort(port);
+ ... clusterBuilder = Cluster.builder().addContactPoint
(host).withPort(port).withoutMetrics();
with conflict with our version.
```

As the comment says, the modified code is legal, but the `withoutMetrics()` method has to be added to handle a conflict issue in the driver. As only the driver has the issue, the inferred bug signature introduces false alarms.

Column "SD" lists false alarms that are related to subtle instances. Two pieces of code can have a low distance value, but their subtle differences determine that they are dissimilar.

Column "Precision" shows our precision:

$$precision = \frac{UB + KB}{UB + KB + S + DS} \tag{4}$$

Our precision values over the four projects are largely consistent. Figure 4 shows the distribution of detected bugs. Its horizontal axis shows detected bugs, in an increasing order of distances that are calculated by Equation 3. Its vertical axis shows projects. The distribution shows that it is feasible to further reduce false alarms, if we focus on those top items. It is worth exploring more advanced techniques to reduce false alarms. We further discuss this issue in Section 7.

In summary, DEPA detected various types of bugs from all the four projects, and its false alarms are already sufficiently low. While the prior approaches (*e.g.*, [82]) detected only known bugs from benchmarks, DEPA detected both known and previously unknown bugs from real projects. Huang *et al.* [37] claim that a useful tool shall present true bugs at the top, and shall allow critical programmers to inspect the remaining bugs. DEPA satisfies this criterion, since Figure 4 shows that most true bugs are ranked at the top.

4.4 RQ2. Complementing Findbugs and PMD

4.4.1 Setting

We used two state-of-the-art tools Findbugs and PMD, to detect bugs for the latest versions of the projects as listed in Table 2, and checked their reported suspicious bugs. We chose these two tools as they are the state-of-the-art static tools and support a large variety of known bug patterns. This research question analyzed whether their predefined rules cover our inferred bug signatures.

4.4.2 Result

Figure 5 shows that PMD and Findbugs in total detected only two of our found bugs (*i.e.*, ARIES-1730 and DERBY-6927). Both are related to leaked database resources. The two tools failed to detect more bugs due to two reasons:

1. Insufficient API rules. Due to the obstacles of learning APIs [74], programmers can introduce API-related bugs [94]. For example, as the methods in Figure 2 can return `null` values, programmers cannot call them in a method call chain. Findbugs and PMD fail to detect this bug, since they do not define this rule nor analyze API code.

2. Insufficient domain knowledge. Findbugs and PMD list their predefined rules [7], [8]. Among these rules, we did not find any rules that are project specific. As a result, both tools fail to detect some bugs (*e.g.*, ARIES-1703 in Section 4.3.2), if it needs domain knowledge to detect such bugs.

The bug signatures inferred by DEPA can improve Findbugs and PMD. It is also interesting to compare DEPA directly with the prior approaches [36], [82], but we must manually repair all the compilation errors of bug fixes, since such tools require code without compilation errors. As most buggy versions and fixed versions have compilation errors after they are checked out, we cannot afford the huge effort. As their compilation errors are not removed, the prior approaches can analyze only several bug fixes that have no compilation errors. DEPA enables us to analyze thousands of bug fixes, without repairing their compilation errors. Indeed, saving the effort is a benefit of DEPA.

4.5 RQ3. The Effectiveness of CCFinderSW and Diff

4.5.1 Setting

Code clones are similar code fragments [76]. Section 4.3.2 shows that the fragments of our found bugs are similar to the ones of past buggy files. The similar fragments can be considered as code clones between the past bug files and the current source files. Even if a source file has bugs, most of its code lines are clean. As a result, if we directly feed bug files to a clone detection tool, it will detect many clones of clean code lines. Although this is not a limitation of clone detection, it makes the detection of bugs less effective. To handle the problem, we extract modified snippets from bug fixes, with the support of `git diff` [14]. After that, we use a clone detection tool, CCFinderSW [79], to detect clones between the modified snippets and the current source files. Here, we select CCFinderSW, because it is an extended version of a famous clone detection tool called CCFinder [42] and it is open source on Github [13]. For simplicity, we call the combination of CCFinderSW and `git diff` as the combined approach. A few recent clone detection tools [34], [40] can detect clones across versions. We did not select these tools, because they are not designed to detect bugs. It needs even more modifications on them to detect bugs, but they are not open source.

As clone detection is expensive, it is unaffordable to analyze all bug fixes. In this research question, we analyze only the latest 1,000 bug fixes from each project. We accept the default settings of CCFinderSW, and set the tool to detect clone sets. To build the links from buggy files to the latest source files, we require that each clone set contains at least a fragment from buggy files and a fragment from source files. If a clone set satisfies the above criterion, we call it as a valid clone set. CCFinderSW generates an identification number for a clone set, and saves clone sets in the

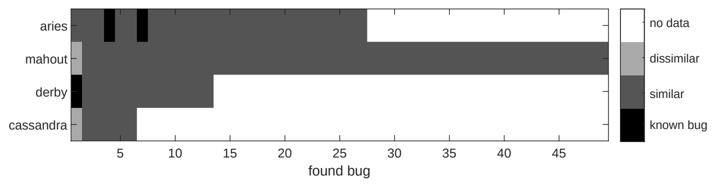


Fig. 6: The distribution of detected code clones

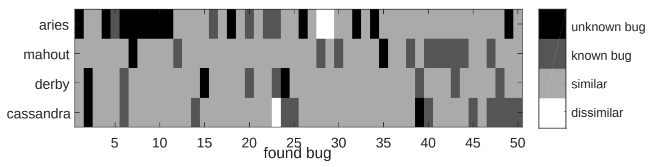


Fig. 7: The distribution of detected bugs with filters off

ascending order of identification numbers. For each project, we also manually inspect the top 50 clone sets to detect bugs.

4.5.2 Result

Figure 6 shows the results. Its labels are of the same meanings with Figure 4. The new label, "no data", denotes that no valid clone set is found. Figure 4 shows that the combined approach detected fewer than 50 suspicious bugs for Aris, Derby, and Cassandra. In particular, it detected only 7 valid clone sets for Cassandra. DEPA compares bug signatures with source files to detect bugs. Although it looks similar to clone detection, the comparison is applied on method graphs and Equation 3 is much more relaxed than the similarities that are defined in clone detection tools. As a result, the combined approach detected fewer valid clone sets. Mahout is an exception. After inspecting its clone sets, we find that most of them are method call chains to initialize parameters:

```
Option dataSourceOpt = obuilder.withLongName("dataSource"
      ).withRequired(
            true).withArgument(
            abuilder.withName("dataSource").withMinimum(1).
                  withMaximum(1).create())
```

These initializers appear in most source files, and are likely to be detected as clones. However, we found that none indicates a bug.

As shown in Figure 6, the combined approach detected many false alarms. Section 3.1.4 illustrates six categories of false alarms where similar code snippets are not bugs. Besides sharing the same categories, the combined approach suffers from two other problems such as trivial modifications and clean lines. For example, a modification is as follows:

```
1 -
2 +
3 Text keyT = new Text(key);
4 Text valueT = new Text(value);
5 currentChunkSize += keyT.getBytes().length + valueT.
      getBytes().length; //
      Overhead
6 writer.append(keyT, valueT);
```

The top two lines delete an invisible tabularor, and the following lines are clean. The combined approach detected clones for the clean lines, but such clones are not bugs.

Although it can still detect several bugs, the effectiveness of the combined approach is much poorer than that of DEPA.

4.6 RQ4. The Impact of Filtering Technique

4.6.1 Setting

We rerun RQ1 with all filters off. After all the data are collected, we rebuild the table of found bugs, and the figure of bug distribution. This setting has two purposes. First, we compare its results with the results in Table 3 and Figure 4. This comparison reveals the impact of our filters. Second, we compare its results with the results in Figure 6. This comparison reveals the impacts of our inference and detection techniques. Compared with the combined approach in Section 4.5, DEPA locates modified code snippets by matching graphs, its identified modified code snippets shall be more accurate than matching texts. In addition, as DEPA uses the Hungarian algorithm to find similar graphs, it can detect code clone with differences. Roy *et al.* [76] show that clone detection tools are typically less effective to detect such clones.

4.6.2 Result

Table 4 shows the results, when we turn off all our filters. Without filters, DEPA detected about one third of previously unknown bugs; one half of unique previously unknown bugs; but most of known bugs. Our filters increase the capability of detecting previously unknown bugs.

Without filters, DEPA produces more false alarms, which are similar to bug signatures but not bugs. Figure 7 shows the distribution. The results show that many false bugs are ranked at the top. Intuitively, a better result leads to a darker figure. Comparing Figure 4 with Figure 7, we find that our filters remove many light gray items, so the figure becomes darker. Our results show that our filtering techniques improve the precision from 25.5% to 51.5%. As references, Kim and Ernst [46] finds that the after their improvements, the precision of Findbugs is around 36%; and Legunsen *et al.* [51] report that when they use mined specs to detect bugs, their precision is 17.2%. Although these precisions may not be directly compared due to different settings (*e.g.*, their different inputs.), they show that our precision is acceptable as tools like Findbugs have been widely adopted in practice.

Although removing filters reduces its effectiveness, with filters off, DEPA still detected much more bugs and reported fewer false alarms than those of the combined approach in Section 4.5.2. The inference technique of DEPA does not suffer from trivial changes and clean lines. From the trivial modification in Section 4.5.2, DEPA inferred no bug signature, in that no method was modified.

In summary, blindly identifying similar code clones can lead to many false positives, and the high ratio of false positives is also notorious in other bug detection tools [41]. Our results show that filtering is a practical way to reduce false positives, but we do not claim that this is a perfect solution to this problem. We further discuss this issue in Section 5.

4.7 RQ5. The Best Size Parameter

4.7.1 Setting

We rerun RQ1, and change the size parameter from two to ten. We analyze their trends of detected bugs to analyze the impact of the size parameter.

4.7.2 Result

As introduced in Section 3.1.3, DEPA has a size parameter that determines the number of nodes inside a bug signature. We rerun RQ1 with different size parameters, and Figure 8 shows the results. Its horizontal axis shows size parameters. The vertical axis shows detected unknown bugs and known bugs. For this parameter, we tried the values from two to ten. Here, it is worth mentioning that the size parameter does not filter bug signatures that call fewer methods than a defined size parameter, as explained in Section 3.1.

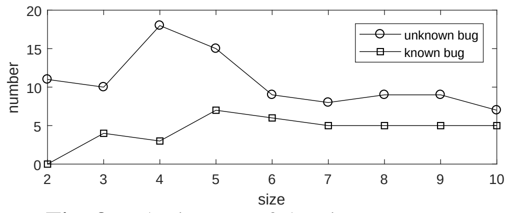


Fig. 8: The impact of the size parameter

For example, the buggy code in Figure 1a calls only three methods. Even with a larger size parameter, its inferred bug signature calls only three methods. However, from other methods that call more methods, it is possible to include unnecessary methods into inferred bug signatures with a larger parameter. In the contrast, a smaller size parameter can lose details of bug signatures. A recent study [92] shows that API usages are typically short, but too fewer nodes may not describe adequate contexts of an API usage. Empirically, our results show that four is the best value for the size parameter.

4.8 Threats to Validity

The external threats to validity include that our findings may be not fully general due to limited subjects. The threat could be reduced by introducing more subjects in future work. The internal threats to validity include the human factors in our study. To reduce the threat, we submit our found bugs to their developers, and other researchers can examine our reported bugs, since our reported bugs are all online. To further reduce the threat, we could invite more participants to verify our results in future work.

5 Discussion and Future Work

Reducing false alarms. The prior studies [27], [41] show that programmers are concerned about false alarms in static bug detection tools. Although our filters have reduced many false alarms, some other ways can also reduce false alarms: (1) the false alarms of static analysis can be removed by dynamic analysis [51], [59] and model checkers [47], and (2) with active learning [80], it can be feasible to use the feedbacks of programmers to remove false alarms. In future work, we plan to explore the above directions.

Inferring context-aware bug signatures. A usage can be associated to a requirement, and is illegal only under specific contexts. Inferring context-aware bug signatures needs a deep understanding on the symptoms and causes of bugs. For a given bug, researchers have proposed various approaches to debug its symptoms [56] and causes [86], [91], which we plan to integrate in future work.

Repairing our detected bugs. DePa detects bugs with past fixes. As a bug fix presents a way to repair a bug, it presents an example to repair a newly found bug. Meng *et al.* [62], [63] systematically locate and apply edits based on a given example of modifications. After we identify bugs, their tool can further learn and apply edits from corresponding bug fixes. In addition, Xuan *et al.* [50] boost automatic program repair with past bug fixes, and Mechtaev *et al.* [61] propose to repair bugs based on a reference implementation. In future work, we plan to analyze to what degree these approaches can handle real bugs.

6 Related Work

Inferring bug signatures. Hsu *et al.* [36] mine bug signature from sequences, but others [24], [53], [70], [82], [97] mine bug signatures from graph representation of code. While the majorities [36], [53], [82], [97] analyze execution traces, a few approaches [53] analyze source files of bugs. Hsu *et al.* [36] and Sun and Khoo [82] use frequency-based mining; Li and Ernst [53] extract subgraphs; and Cheng *et al.* [24] use discriminative graph mining. Kim *et. al.* [43] proposed techniques to mine patching patterns from historical bug fixes for automatic bug repair. Due to various limitations, no previous approaches can analyze large scale bug fixes. It is nontrivial to integrate these tools with Grapa, because Grapa must modify their source files. For example, CBCD [53] is built on CodeSurfer, and as CodeSurfer is a commercial tool, it is infeasible to modify its code. Even if CBCD is boosted to analyze partial programs, it is ineffective to detect bugs with past fixes. From the descriptions of CBCD, we find that it detects only identical code snippets, and the authors of the CBCD paper [53] argue that code with minor differences are not clones. As a result, it is unlikely that CBCD can detect our found bugs (*e.g.*, the one in Figure 2b). DePa compliment the above limitations. Huang *et al.* [38] mine unsafe API calls that lead to crashes. Their approaches need test inputs and compiled code, which are not required by our approach.

Clone detection and its management. Clone detection [75] is a hot research topic. In the literature, researchers [19], [42], [78], [89] have proposed various approaches to detect clones, and their effectiveness is carefully compared [20], [83]. DePa detects similar code across versions, but clone detection tools locate similar code at present. Duala-Ekoko *et al.* [30] propose an approach that tracks the evolutionary clones. Nguyen *et al.* [65] propose an approach that records the modifications on an AST, and applies recorded modifications on its code clone. Lin *et al.* [55] propose an approach that integrates cloned code with the current programming context. Cheng *et al.* [26] extract mappings between code clones and synchronize code clones with extracted mappings. Cheng *et al.* [25] detect cross-language clones. Kim *et al.* [45] show that most clones become less similar during their evolution. Their result shows that the mappings of clones change over time. As a result, it is difficult to prevent bugs that are related to clones. Although our approach is not limited to detect bugs in clones, it provides a treatment for bugs that are introduced during clone evaluation.

Mining specifications. Ammons *et al.* [16] mine automata for APIs. Some researchers [58], [68] refine their approach, and others [66], [67], [96] mine graphs as specs. Robillard *et al.* [73] show that automata and graphs are equivalent. The research in this line can be reduced to the grammar inference problem, and can be solved by corresponding techniques (*e.g.*, the k-tail algorithm [22]). Li and Zhou [54] extract method pairs, and other researchers [77] improve their approach in more complicated contexts. Engler *et al.* [31] extract frequent call sequences, and other researchers [72], [88] improve their approach with more advanced techniques. Furthermore, researchers [52], [60] encode mined sequences as temporal logic. The research in this line can be reduced to sequence mining [15]. All the above approaches concern call sequences. Ernst *et al.* [32] infer invariants to define rules for variables. Researchers [49] combine sequences and invariants for more informative specs, and other researchers [29], [71] use test cases to enrich mined specs. Marc and David [23] mine performance models from runtime traces. Zhong and Mei [92] conduct an empirical study to analyze several open questions for the research line. The above approaches mine legal usages. Although this setting does not apply for us, it can be feasible to borrow their ideas for future improvements.

7 Conclusion

Researchers have explored detecting bugs based on mined knowledge from known bugs. Although their initial results are positive, to the best of our knowledge, no previously unknown bugs were ever detected, due to the challenges of large-scale analysis of bug fixes and selection of proper abstraction levels for bug signatures. In this paper, we propose DePa that overcomes the challenges. We use DePa to mine bug signatures from thousands of bug fixes in four popular open source projects. Our results show that DePa detects 65 bugs that are both previously known (38) or unknown (27) for the four subjects with reasonable precision (51.5%).

References

[1] Cassandra. http://cassandra.apache.org, 2019.
[2] CASSANDRA-11448. https://issues.apache.org/jira/browse/CASSANDRA-11448, 2019.
[3] CASSANDRA-13692. https://issues.apache.org/jira/browse/CASSANDRA-13692, 2019.
[4] eGit. http://www.eclipse.org/egit/, 2019.
[5] Handling disk failures in Cassandra. https://www.datastax.com/dev/blog/handling-disk-failures-in-cassandra-1-2, 2019.
[6] The Apache Git. https://git.apache.org/, 2019.
[7] The bug description of FindBugs. http://findbugs.sourceforge.net/bugDescriptions.html, 2019.
[8] The git-diff command. https://pmd.github.io/pmd-5.8.1/pmd-java/rules/index.html, 2019.
[9] The Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance, 2019.
[10] The manual of WALA. https://github.com/wala/WALA/wiki/Intermediate-Representation-(IR), 2019.
[11] The replicate package of "There and Back Again: Can you Compile that Snapshot?". http://www.cs.wm.edu/semeru/data/breaking-changes, 2019.
[12] WALA. http://wala.sf.net, 2019.
[13] CCFinderSW. https://github.com/YuichiSemura/CCFinderSW, 2020.
[14] The rulesets of PMD. https://git-scm.com/docs/git-diff, 2020.
[15] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. ICDE*, pages 3–14, 1995.
[16] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proc. 29th POPL*, pages 4–16, 2002.
[17] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S. C. Khoo. Semantic patch inference. In *Proc. ASE*, pages 382–385, 2012.
[18] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.
[19] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. ICSM*, pages 368–377, 1998.
[20] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9), 2007.
[21] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon. Xml path language (xpath). *World Wide Web Consortium (W3C)*, 2003.
[22] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 100(6):592–597, 1972.
[23] M. Brünink and D. S. Rosenblum. Mining performance specifications. In *Proc. ESEC/FSE*, pages 39–49, 2016.
[24] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *Proc. ISSTA*, pages 141–152, 2009.
[25] X. Cheng, Z. Peng, L. Jiang, H. Zhong, H. Yu, and J. Zhao. Mining revision histories to detect cross-language clones without intermediates. In *Proc. ASE*, pages 696–701, 2016.
[26] X. Cheng, H. Zhong, Y. Chen, Z. Hu, and J. Zhao. Rule-directed code clone synchronization. In *Proc. ICPC*, pages 1–10, 2016.
[27] M. Christakis and C. Bird. What developers want and need from program analysis: an empirical study. In *Proc. ICSE*, pages 332–343, 2016.
[28] B. Dagenais and L. J. Hendren. Enabling static analysis for partial Java programs. In *Proc. OOPSLA*, pages 313–328, 2008.
[29] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proc. ISSTA*, pages 85–96, 2010.
[30] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. ICSE*, pages 158–167, 2007.
[31] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. 18th SOSP*, pages 57–72, 2001.
[32] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.
[33] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Transactions on Software Engineering*, 33(11):725–743, 2007.
[34] N. Göde and R. Koschke. Incremental clone detection. In *Proc. CSMR*, pages 219–228, 2009.
[35] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA*, pages 132–136, 2004.
[36] H.-Y. Hsu, J. A. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *Proc. ASE*, pages 439–442, 2008.
[37] Q. Huang, X. Xia, and D. Lo. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *Proc. ICSME*, pages 159–170, 2017.
[38] S. Huang, J. Guo, S. Li, X. Li, Y. Qi, K. Chow, and J. Huang. Safecheck: Safety enhancement of Java unsafe API. In *Proc. ICSE*, page to appear, 2019.
[39] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proc. ICSE*, pages 191–200, 1994.
[40] J. Jang, A. Agrawal, and D. Brumley. Redebug: Finding unpatched code clones in entire OS distributions. In *Proc. S&P*, pages 48–62, 2012.
[41] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proc. ICSE*, pages 672–681, 2013.
[42] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
[43] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proc. icse*, pages 802–811, 2013.
[44] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *Proc. ICSE*, pages 333–343, 2007.
[45] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. ESEC/FSE*, pages 187–196, 2005.
[46] S. Kim and M. D. Ernst. Which warnings should I fix first? In *Proc. ESEC/FSE*, pages 45–54, 2007.
[47] Y. Kim, J. Lee, H. Han, and K.-M. Choe. Filtering false alarms of buffer overflow analysis using smt solvers. *Information and Software Technology*, 52(2):210–219, 2010.
[48] H. W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.
[49] T. Le, X. Le, D. Lo, and I. Beschastnikh. Synergizing specification miners through model fissions and fusions. In *Proc. ASE*, pages 115–125, 2015.
[50] X. B. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *Proc. SANER*, pages 213–224, 2016.
[51] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov. How good are the specs? a study of the bug-finding effectiveness of existing Java API specifications. In *Proc. ASE*, pages 602–613, 2016.
[52] C. Lemieux, D. Park, and I. Beschastnikh. General LTL specification mining. In *Proc. ASE*, pages 81–92, 2015.
[53] J. Li and M. D. Ernst. CBCD: Cloned buggy code detector. In *Proc. ICSE*, pages 310–320, 2012.
[54] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. ESEC/FSE*, pages 306–315, 2005.
[55] Y. Lin, X. Peng, Z. Xing, D. Zheng, and W. Zhao. Clone-based and interactive recommendation for modifying pasted code. In *Proc. ESEC/FSE*, pages 520–531, 2015.
[56] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong. Feedback-based debugging. In *Proc. ICSE*, pages 393–403, 2017.
[57] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.
[58] L. M. Lo, David and M. Pezzè. Automatic steering of behavioral model inference. In *Proc. ESEC/FSE*, pages 345–354, 2009.
[59] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proc. SOSP*, pages 103–116, 2007.
[60] S. Maoz and J. O. Ringert. GR(1) synthesis for LTL specification patterns. In *Proc. ESEC/FSE*, pages 96–106, 2015.

[61] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury. Semantic program repair using a reference implementation. In *Proc. ICSE*, pages 129–139, 2018.

[62] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *Proc. PLDI*, pages 329–342, 2011.

[63] N. Meng, M. Kim, and K. S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 502–511, 2013.

[64] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[65] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, 2012.

[66] H. V. Nguyen, H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mining interprocedural, data-oriented usage patterns in JavaScript web applications. In *Proc. ICSE*, pages 791–802, 2014.

[67] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proc. ESEC/FSE*, pages 383–392, 2009.

[68] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *Proc. ICSE*, pages 815–825, 2012.

[69] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. $TESSERACT$: Eliminating experimental bias in malware classification across space and time. In *Proc. USENIX Security*, pages 729–746, 2019.

[70] N. H. Pham, T. T. Nguyen, H. A. Nguyen, X. Wang, A. T. Nguyen, and T. N. Nguyen. Detecting recurring and similar software vulnerabilities. In *Proc. ICSE*, volume 2, pages 227–230, 2010.

[71] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *Proc. ICSE*, pages 288–298, 2012.

[72] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *Proc. ICSE*, pages 240–250, 2007.

[73] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.

[74] M. P. Robillard and R. DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[75] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, 2007.

[76] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[77] A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining multi-level API usage patterns. In *Proc. SANER*, pages 23–32, 2015.

[78] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC: scaling code clone detection to big-code. In *Proc. ICSE*, pages 1157–1168, 2016.

[79] Y. Semura, N. Yoshida, E. Choi, and K. Inoue. CCFinderSW: Clone detection tool with flexible multilingual tokenization. In *Proc. APSEC*, pages 654–659, 2017.

[80] B. Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[81] L. Shi, H. Zhong, T. Xie, and M. Li. An empirical study on evolution of API documentation. In *Proc. FASE*, pages 416–431, 2011.

[82] C. Sun and S.-C. Khoo. Mining succinct predicated bug signatures. In *Proc. ESEC/FSE*, pages 576–586, 2013.

[83] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with bigclonebench. In *Proc. ICSME*, pages 131–140, 2015.

[84] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Proc. ICSE*, pages 386–396, 2012.

[85] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4), 2017.

[86] H. Wang, Y. Lin, Z. Yang, J. Sun, Y. Liu, J. S. Dong, Q. Zheng, and T. Liu. Explaining regressions via alignment slicing and mending. *IEEE Transactions on Software Engineering*, 2019.

[87] Y. Wang, N. Meng, and H. Zhong. An empirical study of multi-entity changes in real bug fixes. In *Proc. ICSME*, pages 287–298, 2018.

[88] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. ESEC/FSE*, pages 35–44, 2007.

[89] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proc. ASE*, pages 87–98, 2016.

[90] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proc. ESEC/FSE*, pages 15–25, 2011.

[91] Z. Xu, S. Ma, X. Zhang, S. Zhu, and B. Xu. Debugging with intelligence via probabilistic inference. In *Proc. ICSE*, pages 1171–1181, 2018.

[92] H. Zhong and H. Mei. An empirical study on API usages. *IEEE Transactions on Software Engineering*, 2018.

[93] H. Zhong and N. Meng. Towards reusing hints from past fixes -an exploratory study on thousands of real samples. *Empirical Software Engineering*, 2018.

[94] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proc. ICSE*, pages 913–923, 2015.

[95] H. Zhong and X. Wang. Boosting complete-code tools for partial program. In *Proc. ASE*, pages 671–681, 2017.

[96] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proc. ASE*, pages 307–318, 2009.

[97] Z. Zuo, S.-C. Khoo, and C. Sun. Efficient predicated bug signature mining via hierarchical instrumentation. In *Proc. ISSTA*, pages 215–224, 2014.