Fixing Recurring Crash Bugs via Analyzing Q&A Sites

Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, Hong Mei
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, School of Electronics Engineering and Computer Science,
Peking University, Beijing, 100871, P. R. China
{gaoqing11, zhanghs12, wangjie14, xiongyf04, zhanglu, meih}@sei.pku.edu.cn

Abstract—**Recurring bugs are common in software systems, especially in client programs that depend on the same framework. Existing research uses human-written templates, and is limited to certain types of bugs. In this paper, we propose a fully automatic approach to fixing recurring crash bugs via analyzing Q&A sites. By extracting queries from crash traces and retrieving a list of Q&A pages, we analyze the pages and generate edit scripts. Then we apply these scripts to target source code and filter out the incorrect patches. The empirical results show that our approach is accurate in fixing real-world crash bugs, and can complement existing bug-fixing approaches.**

I. Introduction

Many bugs are recurring bugs. Recurring bugs are bugs that occur often in different projects, and are found common, accounting for 17%-45% of the bugs [1, 2]. One important reason for bugs to recur is that many modern programs depend on a certain framework, e.g., Android, Spring, and Hadoop. Problems may occur when certain constraints in these frameworks are violated. For example, one framework may require calling a specific method to initialize an object before using it, otherwise a crash may occur. Programmers of different applications may all forget to call the specific method, leading to recurring crash bugs.

The recurrence of bugs gives us opportunities to fix them automatically. Recently, different approaches are proposed to fix bugs automatically by exploiting the recurrence. One of the most influential approaches is GenProg [3], which copies code pieces from other parts of the software project to fix the current bug. However, this approach does not work if a correct fix cannot be formed from the current project. PAR [4] uses ten manually defined fix templates to fix bugs, and thus is not confined by the code in the current project. However, since the templates are extracted manually, only limited types of bugs can be fixed. In real-world programs, bug-fixing patterns can be numerous, and can vary from one framework to another. It is impractical to write every such template manually.

To overcome the problem of manual fix-pattern extraction, in this paper we aim to infer fixes automatically via analyzing Q&A sites. We observe that, many recurring bugs have already been discussed over the Q&A sites such as Stack Overflow, and we can directly obtain the fixes from the Q&A sites. Furthermore, it is common for programmers to search the Q&A sites when they encounter a bug with respect to a certain framework, which indicates that Q&A sites are more or less a reliable source for obtaining fixes for a large portion of bugs.

As the first step of fixing recurring bugs via analyzing Q&A sites, we focus on a specific class of bugs: crash bugs. Crash bugs are among the most severe bugs in real-world software systems, and a lot of research efforts have been put into handling crash bugs, including localizing the causes of crash bugs [5], keeping the system running under the presence of crashes [6], and checking the correctness of fixes to crash bugs [7]. However, despite the notable progress in automatic bug fixing [8, 9, 4, 10, 11, 12], there is no approach that is designed to directly fix crash bugs within our knowledge.

It is not easy to automate the bug fixes via Q&A sites. First, we need to locate a suitable Q&A web page that describes a bug of the same type and contains a solution. It is easy for humans to come up with a few keywords, query a web search engine, and read through the returned pages to find the most suitable one. However, it is not easy to do it automatically. Second, even if we can locate a correct Q&A web page, it is still difficult to extract a solution from a page where questions and answers are described in a natural language.

To overcome the first problem, we utilize the fact that a Q&A page discussing a crash bug usually has a crash trace, which contains certain information about the bug, such as an error message and a call stack. We could construct a query using such information and ask a web search engine to locate suitable pages. However, it is not feasible to directly construct such a query from a crash trace, because texts in a crash trace usually contain a lot of project-specific information, such as a project method name and the name of a problematic variable. The project-specific texts would not match the bug appearing in the Q&A web site. To overcome this problem, we further filter out project-specific texts.

To overcome the second problem, we utilize a fact obtained by studying Q&A web pages: many Q&A pages contain code snippets, and it is enough to fix many bugs by only looking at the code snippets on the pages. In this way we can avoid complex natural language processing and use almost only program analysis. For example, a developer asking a question about a bug may post his/her source code snippet, and a reply answering the question may contain a fixed version of the code snippet. By comparing the two code snippets, we can directly obtain a fix.

However, even only analyzing code snippets is not easy. Due to the fuzzy nature of Q&A pages, there may not be a clear correspondence between the buggy and fixed versions

Yingfei Xiong is the corresponding author.

of the code. Furthermore, we cannot directly apply the fix described in the web page to the target project, as the code in the web page is usually different from the source code in the target project. To overcome these difficulties, we systematically combine a set of existing techniques, including partial parsing [13, 14], tree-based code differencing [15, 16], and edit script generation [17]. These techniques together allow us to deal with the fuzzy nature of the web code as well as the gap between the project and the web page.

In summary, our contributions are as follows:

- We propose an approach to fixing recurring crash bugs via analyzing Q&A sites. To our knowledge, this is the first approach for automatic program repair using Internet resources.
- We demonstrate that fixes in Q&A sites can be obtained and applied by combining a set of fuzzy program analysis techniques, without complex natural language processing.
- We evaluate our approach with real-world crash bugs from GitHub, and manually verify the correctness of the generated patches. Our evaluation shows that our approach is effective in fixing real-world recurring crash bugs, and can complement existing bug-fixing approaches.

II. Approach Overview

We first introduce the general structure of our approach in Section II, and then introduce each step in Section III. Our current approach is implemented in Java, but is not limited to a specific programming language.

When a program crashes, it has a crash trace. Our approach uses the source code and the crash trace as input, and consists of four steps: Q&A page extraction, edit script extraction, patch generation, and patch filtering. Fig. 1 shows the overview of our approach. The number on each arrow is the step number.

The first step of our approach is Q&A page extraction. Given a crash trace, we extract keywords, and give it to a search engine. The search engine then returns a list of Q&A pages. In the second step, we isolate code snippets from each Q&A page, and combine them to buggy & fixed code pairs, in which a fixed code snippet may contain a fix to a buggy code snippet. After reducing code size in each code pair, we build mappings between code snippets in each pair, and generate edit scripts that indicate how to transform the buggy code to the fixed code. In the third step, we extract source code snippets by using the crash trace and buggy code snippets, and apply each edit script to each source code snippet. In the last step, we filter generated patches, and report the fixing result. In the following section we will describe each step in detail.

III. Approach Detail

We now explain our approaches in detail. We use a running example taken from a real-world crash bug[1] in an Android application, and the crash trace is shown in Fig. 2. In the crash trace, Line 1 and Line 13 are two error messages that describe the crash. Line 2-12 and Line 14-17 represent two call stacks. Fig. 3 shows a source code snippet in this example, and we use the word "location" to indicate the line number of an individual statement. In this example, Line 31 in Fig. 3 is the faulty location of the source code. The root cause of the bug is that the method `OnReceive()` passes `context` to the method `level()` of `BatteryHelper`. The method `level()` makes the parameter register a receiver. However, for `context` this is not allowed. We describe Q&A page extraction in Section III-A, edit script extraction in Section III-B, patch generation in Section III-C, and patch filtering in Section III-D.

```
29   public void onReceive (final Context context, final Intent intent) {
30       final int action = intent.getExtras().getInt(KEY_ACTION, -1);
31       final float bl = BatteryHelper.level(context);
32       LOG.i("AlarmReceiver invoked: action=%s bl=%s.", action, bl);
33       switch (action) {
...          ...
51       }
52   }
```

Fig. 3: The source code snippet

A. Q&A Page Extraction

To fix the bug, our approach begins with Q&A page extraction. In this step, we generate a query, and give the query to a web search engine to obtain a list of Q&A pages. Based on our observation, the first line of the crash trace can be used as the query, as it usually contains (1) the exception type, and (2) an error message about the crash. Both are information unique to the current bug. For example, Line 1 in Fig. 2 indicates the exception is a `RuntimeException` and the cause of the problem is that "`IntentReceiver` components are not allowed to register to receive intents".

However, we cannot directly use the whole first line, because some words in the error message are project specific, and if we include these words, the search engine will hardly return any answer. In the example, The word `com.vaguehope.onosendai.update.AlarmReceiver` is a class name defined in the target project.

To overcome this problem, we observe that project-specific items are usually reported in full qualified names, and thus we can filter out such items using the root package of the project. Basically, we filter out all words which contain a substring equal to the name of the root package. In our example, we generate the query `java.lang.RuntimeException: Unable to start receiver IntentReceiver components are not allowed to register to receive intents`. We give the query to a search engine, and obtain a ranking list of Q&A pages.

B. Edit Script Extraction

The second step of our approach is edit script extraction. An edit script is a sequence of edit operations that describes how to transform one code snippet to another. In our work we use a tree-based edit script generation algorithm, in which an edit script describes operations on the Abstract Syntax Tree (AST). We can add, delete, update, or move a node in an AST in one edit operation.

We extract edit scripts in three steps: buggy & fixed code pair extraction, buggy & fixed code reduction, and edit script generation.

[1]https://github.com/haku/Onosendai/issues/100

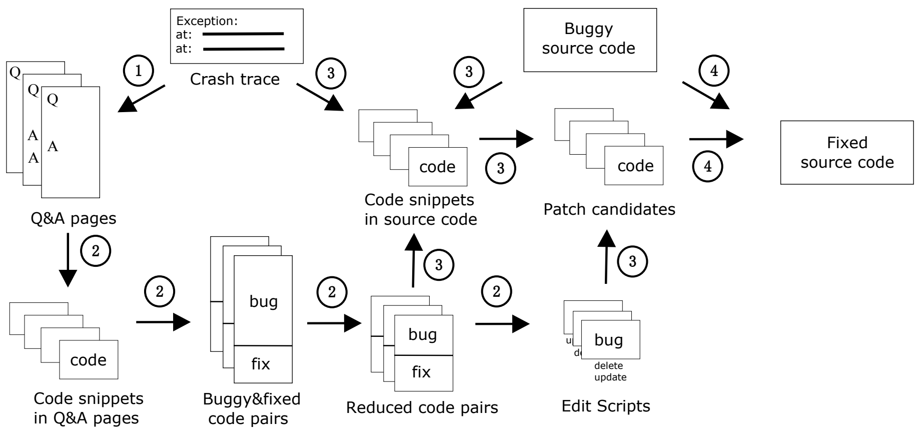
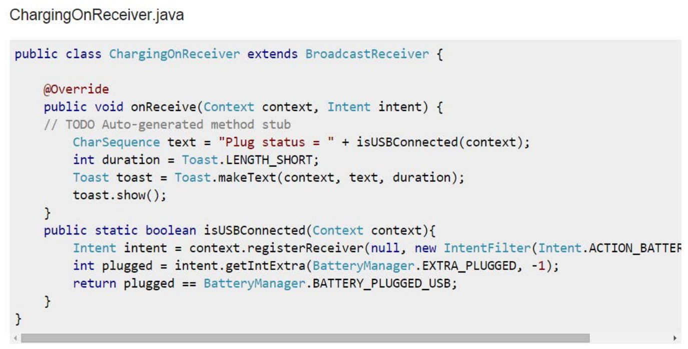


Fig. 1: Overview of our approach

```
1   java.lang.RuntimeException: Unable to start receiver com.vaguehope.onosendai.update.AlarmReceiver:
    android.content.ReceiverCallNotAllowedException: IntentReceiver components are not allowed to register to receive intents
2   at android.app.ActivityThread.handleReceiver(ActivityThread.java:2126)
3   at android.app.ActivityThread.access$1500(ActivityThread.java:123)
4   at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1197)
5   at android.os.Handler.dispatchMessage(Handler.java:99)
6   at android.os.Looper.loop(Looper.java:137)
7   at android.app.ActivityThread.main(ActivityThread.java:4424)
8   at java.lang.reflect.Method.invokeNative(Native Method)
9   at java.lang.reflect.Method.invoke(Method.java:511)
10  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:784)
11  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:551)
12  at dalvik.system.NativeStart.main(Native Method)
13  Caused by: android.content.ReceiverCallNotAllowedException: IntentReceiver components are not allowed to register to receive intents
14  at android.app.ReceiverRestrictedContext.registerReceiver(ContextImpl.java:118)
15  at android.app.ReceiverRestrictedContext.registerReceiver(ContextImpl.java:112)
16  at com.vaguehope.onosendai.update.AlarmReceiver.onReceive(AlarmReceiver.java:31)
17  at android.app.ActivityThread.handleReceiver(ActivityThread.java:2119)
18  ... 10 more
```

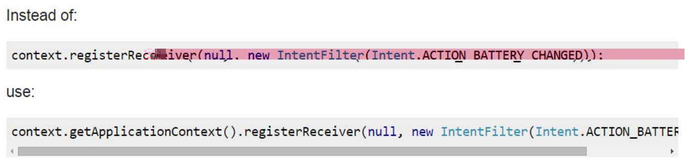
Fig. 2: An example of a crash trace


ChargingOnReceiver.java

```
public class ChargingOnReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
    // TODO Auto-generated method stub
        CharSequence text = "Plug status = " + isUSBConnected(context);
        int duration = Toast.LENGTH_SHORT;
        Toast toast = Toast.makeText(context, text, duration);
        toast.show();
    }
    public static boolean isUSBConnected(Context context){
        Intent intent = context.registerReceiver(null, new IntentFilter(Intent.ACTION_BATTER
        int plugged = intent.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
        return plugged == BatteryManager.BATTERY_PLUGGED_USB;
    }
}
```

(a) Part of a question post


Instead of:

```
context.registerReceiver(null, new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
```

use:

```
context.getApplicationContext().registerReceiver(null, new IntentFilter(Intent.ACTION_BATTER
```

(b) Part of an answer post

Fig. 4: Part of a Q&A page for the crash trace in Fig. 2

1) Buggy & Fixed Code Pair Extraction: In the ranking order of the Q&A pages, we first extract code snippets in each page. Fig. 4 shows part of a Q&A page returned from a search engine given the above query. In this figure, there are three code snippets, one in the question post, and the other two in the answer post.

To extract buggy & fixed code pairs, we first need to isolate code snippets from natural language descriptions in each post. We isolate code snippets by taking the snippets from inside the HTML tag pair `<code>` and `</code>` (grey in Stack Overflow as shown in Fig. 4). This may miss some code snippets which are not tagged, but according to our observation most of the code snippets are in this type of tag pairs.

Then we combine different code snippets to buggy & fixed code pairs. A buggy & fixed code pair may be either of the following:

1) Both buggy code and fixed code are in the same answer post.
2) Buggy code is in the question post, and fixed code is in the answer post.

To identify the first type of code pairs, we identify answer posts that have more than one code snippets, and use keyword matching to distinguish the buggy code and the fixed code.

The keywords are commonly used by humans to explain comparison relationship, such as “instead of” and “change...to...”. If such keywords exist, we combine the two code snippets into one code pair, and distinguish buggy and fixed code snippets according to the keywords.

To identify the second type of code pairs, we take each code snippet in the question post and each code snippet in the answer post as a buggy & fixed code pair.

Because the first type of code pairs is more likely to be a buggy and fixed code pair, we rank this type of code pairs before the second type of code pairs. As a result, we obtain three code pairs for the running example: one taken from only the answer post, and two taken from both the question and the answer posts. The fixed code snippet in the answer post suggests inserting `getApplicationContext()` to `context`, which fixes the buggy code snippet.

2) Buggy & Fixed Code Reduction: The buggy and fixed code snippets are often not similar in size. In Fig 4, there are more than ten lines in the buggy code snippet of the question post, while only one line in the fixed code snippet of the answer post. The big difference in the code will influence the correctness of the generated edit script. Therefore, before analyzing the code pair, we reduce the size of both buggy code and fixed code according to their similarities.

First, we parse a buggy & fixed code pair and get two Abstract Syntax Trees (ASTs). Each node in the AST has a label, which indicates the type of the node (e.g., method invocation), and for each leaf node it also has a value (e.g., the name of a variable). The code snippets are usually not complete. Therefore, we use partial parsing [13] techniques to parse the code snippets into ASTs.

Then we calculate the similarities between each statements in the code pair, and filter out those statements that has only low similarity scores. Here we consider two types of similarities:

1) Text similarity. We calculate the edit distance between each line, and denote the length of buggy code as len_buggy, and the length of fixed code as len_fixed. We use the following formula:

$$Sim(Text) = 1 - \frac{edit\ distance}{\sqrt{len\_buggy * len\_fixed}}$$

2) Structure similarity. We calculate the AST similarity using the ratio of common AST leaf nodes among all the leaf nodes in two ASTs. We denote the number of common leaf nodes as *num_common,* and the total number of leaf nodes in two ASTs as *num_total.* We use the following formula:

$$Sim(Structure) = \frac{num\_common}{num\_total}$$

In both cases, we only take code elements with one of the similarity scores more than a pre-defined threshold. In this way we can reduce the size of each code snippet in the code pair greatly. The reduced code pairs are shown in Fig. 5. Each code snippet above the line is a considered as a buggy code snippet, and each code snippet under the line is considered as a fixed code snippet.

```
context.registerReceiver(...);
context.getApplicationContext().registerReceiver(...);
```

(a) Code pair from the same answer post

```
Intent intent = context.registerReceiver(...);
context.registerReceiver(...);
```

```
Intent intent = context.registerReceiver(...);
context.getApplicationContext().registerReceiver(...);
```

(b) Code pairs from both the question and answer post

Fig. 5: Reduced code pairs from Fig. 4

3) Edit Script Generation: We leverage a state-of-art edit script generation technique, GumTree [16], to generate edit scripts for buggy code snippets. By applying the edit script to a buggy code snippet, we shall get the corresponding fixed code snippet. Given two ASTs, GumTree works in two steps. First, it builds mappings between the nodes of the ASTs. A leaf or inner node of one AST can be mapped to a leaf or inner node of the other AST, and each node can only be mapped once. There may be nodes that do not have any mapping. Second, it generates exactly one edit script using an existing linear optimal algorithm [17]. The edit script contains four types of edit operations on a node (including leaf and inner) of an AST, namely add, delete, update, and move. Here we explain these operations using the definitions in the corresponding paper [16]:

- $add(t, t_p, i, l, v)$: Add a new node t in the AST. If t_p is not null and i is specified then t is the i^{th} child of t_p. Otherwise t is the new root node and has the previous root node as its only child. Finally, l is the label of t and v is the value of t.
- $delete(t)$: Delete a leaf node t of the AST.
- $update(t, v_n)$: Replace the old value of a node t by the new value v_n.
- $move(t, t_p, i)$: Move a node t and make it the i^{th} child of t_p. Note that all children of t are moved as well, and therefore this actions moves a whole subtree.

Let us denote the AST of the buggy code snippet as *buggyAST*, and the AST of the fixed code snippet as *fixedAST*. We use the code pair in Fig. 5(a) for explanation. The mappings generated by GumTree between *buggyAST* and *fixedAST* for this code pair is shown in Fig 6. Long-dotted and short-dotted lines indicate mappings built by GumTree in its different steps, and are considered the same in our approach. Suppose in *buggyAST* the node corresponding to `context` is C, and the parent node of C is P, while in *fixedAST*, the node that is mapped to C is C', the node corresponding to `getApplicationContext` is G', and the parent node of C' and G' is M', which is labeled as “MethodInvocation”, corresponding to `context.getApplicationContext()`. The edit script is as follows. For simplicity, we omit the last two parameters of *add* operations, and use “equivalent” to indicate that the label and value of a newly added node are the same as those of an existing node.

1) $add(M, P, 1, ...)$, where M is equivalent to M'

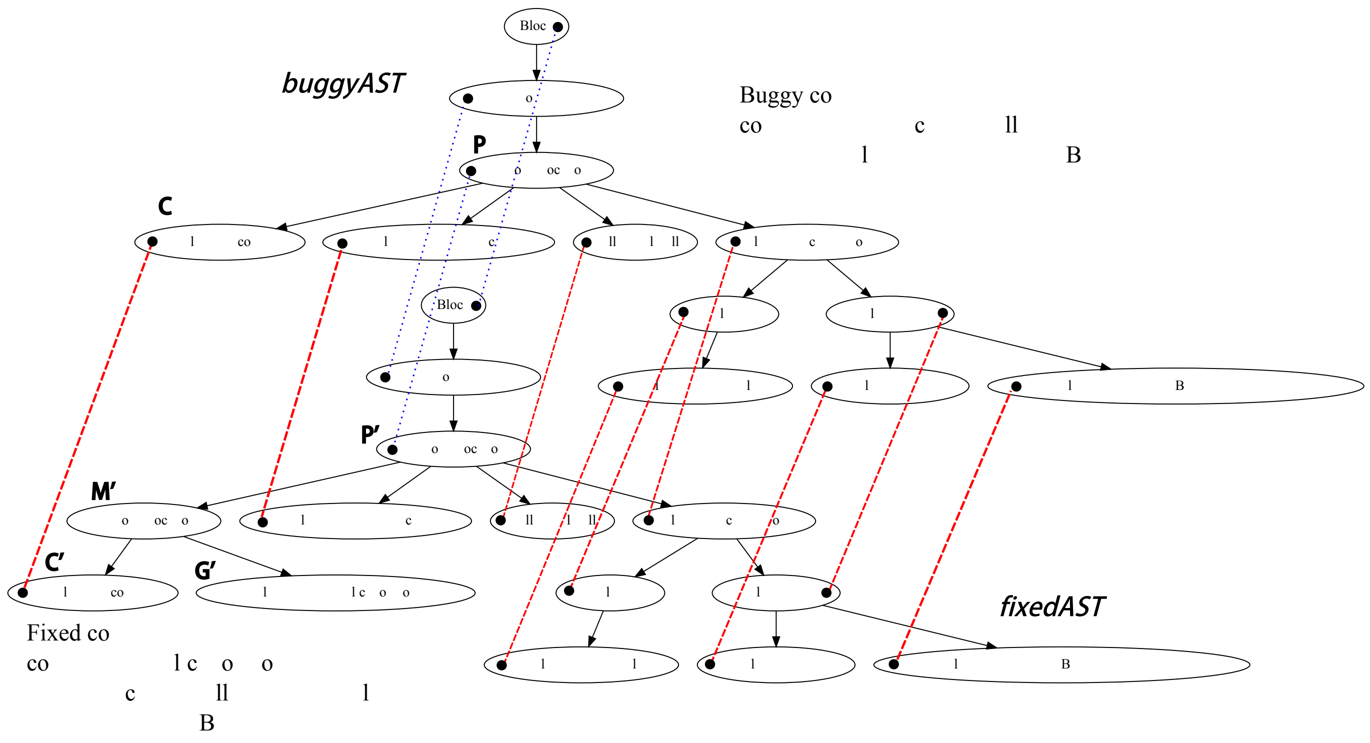


Fig. 6: The mappings between *buggyAST* and *fixedAST* built by GumTree

2) $move(C, M, 1)$
3) $add(G, M, 2, ...)$, where G is equivalent to G'

The above edit script aims to *reflect* the changes of two ASTs, rather than to *apply* the changes to new code context. Consider a new code snippet, Line 31 in Fig. 3. In the AST of this code snippet, the parent node of `context` (denoted as P'') has three children, which are `BatteryHelper`, `level`, and `context`. If we directly apply the above edit script to this AST, P'' will have the following three children: `context.getApplicationContext()`, `BatteryHelper`, and `level`. The corresponding source code fails to compile.

This is a result of differences in two code snippets containing the same recurring bug, although the fix pattern is the same. The differences are mainly in two cases: changed position and renamed variable. Suppose we apply an *add* operation to the AST of a new code snippet, denoted as *newAST*. In the changed position case, as the example shows, the position of the added node should be changed from the 1st child to the 3rd child of the parent node. In the renamed variable case, the variable

Algorithm 1 Generating a *replace* operation

```
$n'$: the non-root node of $fixedAST$
$p'$: the parent node of $n'$
$i$: the index of $n'$ in $p'$
$newNode$: a newly added node
$N.mappedNode$: the mapped node of $N$ in $buggyAST$

if $n'.hasMapping$ then
    if $n'.label \neq n'.mappedNode.label$ then
        return $replace(newNode, n'.mappedNode, n')$
else
    if $p'.hasMapping$ then
        $p := p'.mappedNode$
        if $p.childNum>i$ then
            $n := p.getChild(i)$
            for each $e' \leftarrow p'.children$ do
                if $e'.mappedNode == n$ then
                    return NULL
            if $n.label \neq n'.label$ then
                return $replace(newNode, n, n')$
return NULL
```

$copy(newNode, p, i, f)$, where $newNode$ is a newly added node.

For the running example, we generate a *replace* operation. The edit script is shown below. We use it instead of the original edit script generated by GumTree.

1) $replace(M, C, M')$
2) $move(C, M, 1)$
3) $add(G, M, 2, ...)$, where G is equivalent to G'

C. Patch Generation

We first extract source code snippets from the target project, and combine them with buggy code snippets from Q&A pages to obtain a list of buggy & source code pairs. Then we apply each edit script of the buggy code snippet to the corresponding source code snippet to obtain patches.

1) Buggy & Source Code Pair Extraction: We extract buggy & source code pairs as follows. First, we perform file-level fault localization. We take all files in the project that appear in the call stacks of the crash trace from top down, and obtain a list of candidate files. In Fig. 2, the call stacks suggests that there is only one candidate file `AlarmReceiver.java`. Second, we extract buggy & source code pairs. The buggy code snippets come from existing buggy & fixed code pairs, while the source code snippets come from candidate files just extracted. If a buggy code snippet is a method, we search for a method with the same name in the candidate files, and combine these two methods as a buggy & source code pair. If the buggy code snippet is a block, which is in most of the occasions, the algorithm consists of three steps explained below.

First, we use call stacks and the buggy code snippet to pinpoint faulty locations. A call stack already contains a list of line numbers, and thus we take each line number of the corresponding candidate file in the call stack from top down. The buggy code snippet may also help us find a faulty location. We first calculate similarity scores between each statement in the candidate files and each statement in the buggy code snippet using the formulae in Section III-B2. Then we filter out statements in candidate files with similarity scores less than the same pre-defined threshold, and sort faulty locations indicated by the rest of the statements in descending order of similarity scores. We rank the faulty locations obtained from call stacks before those obtained from the buggy code snippet.

Second, according to the size of the buggy code snippet, we expand each faulty location inside the candidate files, and combine them to obtain a buggy & source code pair. Specifically, we expand each faulty location forward in the corresponding candidate file to obtain a possible block whose size is the same as the size of the buggy code block. Now we have a list of buggy & source code pairs.

Third, since the faulty location for a crash is not necessarily the exact line number identified, for the source code snippet in each buggy & source code pair, we also choose the previous location and the next location with the same block size as two additional source code snippets. Therefore, for each buggy code snippet we obtain two additional buggy & source code pairs. In the source code fragment in Fig. 3, we extract 3 buggy & source code pairs for the buggy code snippet in Fig 5(a). The source code snippet in these pairs are Line 31, Line 30, and Line 32 in order.

2) Edit Script Application: We denote the AST of a source code snippet as *srcAST*. Given a buggy & source code pair, we use GumTree again to build mappings between *buggyAST* and *srcAST*. According to the mappings, each operation in the edit script on a node of *buggyAST* is now effective on the mapped node of *srcAST*. If there is an unmapped node in the edit script, we do not generate a fix. In the example, GumTree maps C in *buggyAST* to `context` in *srcAST* (denoted as C'') in Fig 3. The edit script is transformed to the following to operate nodes of *srcAST*:

1) $replace(M'', C'', M')$
2) $move(C'', M'', 1)$
3) $add(G'', M'', 2, ...)$, where G'' is equivalent to G'

For each buggy & source code pair in order, we apply each transformed edit script to *srcAST*, and transform the edited AST back to code. Finally we obtain a ranking list of generated patches. The patches are naturally sorted as our analysis proceeds. Therefore, it is sorted by the Q&A page ranking, and code pairs in the same answer post is ranked higher than those in both question and answer post. In addition, faulty locations identified by the call stack is ranked higher than those identified by the buggy code.

D. Patch Filtering

In previous steps, we may generate multiple patches for one bug. However, some of them may be incorrect. We filter out the patches using the following two rules:

1) Merging. Our approach may generate multiple patches that are equivalent. We check the equivalence at the AST level, and merge them as one patch.
2) Compiling. If there is a compilation failure, we filter out the patch.

In the end, we report the first k patches in the list to the programmer. If there is no patch generated, it means that our approach fails to fix the crash bug. Our experiment shows that we have high accuracy in generating the first patch as a correct patch. Therefore, we set k=1.

In the running example, the project compiles successfully and the code becomes the following.

```
final float bl=BatteryHelper.level(
                context.getApplicationContext());
```

Therefore, we get one fix for this crash bug.

IV. Evaluation

Our evaluation aims to answer two questions:

RQ1: Effectiveness. How effective is our approach in fixing real-world recurring crash bugs?

RQ2: Usefulness. Can our approach complement state-of-art fixing approaches?

A. Experiment Setup

We have implemented our approach in Java as an open source tool, *QACrashFix*[2]. We used Google as the search engine to obtain Q&A pages, and added a constraint "site:stackoverflow.com" into the keywords to retrieve only the web pages in Stack Overflow. We used Eclipse AST parser [18] to parse code snippets to ASTs, and re-implemented GumTree [16] to build mappings and to generate edit scripts.

Through our experiments, we set $Sim(Text)$ to 0.8, and $Sim(Structure)$ to 0.3 to achieve the best results. Different threshold may lead to different size of reduced code snippets, or introduce different number of source code snippets. In both cases we may have more false positives or false negatives.

To evaluate our approach, we need a set of crash bugs as evaluation subjects. Here we focus on a specific framework, Android, because Android is one of the most widely-used framework, and on GitHub there are a large number of android projects.

To collect the subjects, we looked into the Android projects on GitHub. GitHub provides four rankings for projects: best match, most stars, most forks, and recently updates. We obtain the top 1000 Android projects from each ranking, and get in total 2529 projects, containing 73868 issues. Then we filter the issues based on three criteria: (1) the issue contains a crash trace, which indicates that it is a crash bug, (2) the issue has an associated patch, so that we can evaluate the generated patches by comparing them with developers' patch, and (3) the exception causing the crash is thrown from Android framework, which indicates that the crash is a recurring bug related to Android. After filtering, we got 90 projects with 161 issues.

Next, we manually examined these issues to determine which crash bugs can be fixed by humans via searching Stack Overflow, i.e., we identified those bugs whose recurrences existed at Stack Overflow. Our rule for judging this is to use the same method as our approach to generate a query, and manually examine top 10 Q&A pages at Stack Overflow. For each page, we checked whether we could fix the bug using the information on the page. In the end, we got 25 issues whose recurrences exist on Stack Overflow. The recurrence ratio is 15.5%, which is less than but similar to the recurring bug rate in recent research (17%-45% [1, 2]). This is because in repositories there are sufficient resources. By only searching in Stack Overflow we found a large number of recurring bugs, which also indicates the effectiveness of using Q&A sites.

[2]available at http://sei.pku.edu.cn/%7gaoqing11/qacrashfix

For each issue, we downloaded and deployed the project version before the patch was applied, and wrote building scripts for automating the compilation process. Because we cannot compile one project, we only chose the remaining 24 issues (corresponding to 24 bugs) as our final benchmark.

Finally, we used our approach to generate a patch for each bug. Then we manually verified the correctness of the patches by comparing each generated patch with each patch written by developers. Note that a lot of existing research on bug-fixing adopted "passing all tests" as a criterion for evaluating the correctness of generated patches. However, we did not adopt this method because recent research [19] found that although test cases are effective in filtering out many erroneous patches, many test suites in practice are weak and are not enough to guarantee the correctness of patches.

All our experiments were executed on Windows 7, with a dual-core 2.50GHz Intel Core5 processor and 8GB memory. In the following subsections we discuss the result in detail with respect to our research questions.

B. RQ1: Effectiveness

The details of the benchmark and the experimental results are summarized in Table I, sorted by the number of lines of code. Column "Project" shows the project name. Column "Issue No." shows the issue number in GitHub. Column "Loc" shows the total number of lines of code in the respective project. Column "#Edit Scripts" shows the number of edit scripts (i.e., how many buggy & fixed code pairs) we generated from each web page. Column "Initial" shows patches initially generated on the target project without any filtering. Column "Equivalent" shows the number of patches that are equivalent. Column "Compile Error" shows the number of patches that fail to compile. Column "Remaining" shows the number of the remaining patches, which are the final patches of our approach. Column "Correct" shows whether the first filtered patch can fix the bug or not. Column "Total" shows the time used to generate all the patches, and Column "Compilation" shows the time used for compilation. We also recorded the time to obtain the first filtered patch, shown in Column "First". We make the following observations.

First, the column of remaining fixes shows that, for 14 of bugs, we did not generate any fixes. For the resting 10 bugs, our tool generated at least one fix.

Second, our tool generated a relative large number of initial fixes, which shows that there are a good number of code snippets in Stack Overflow pages that lead to fix generation.

Third, our tool may generate equivalent patches. This is because Stack Overflow pages may contain the same answer several times. Since the code snippets in the page are the same, we generate equivalent patches.

Fourth, a large number of patches can be filtered out by compilation. For example, in `TextSecure`, we generated 40 initial fixes for each bug, and filtered out all of them by compilation. In total we filtered out 127 patches by compilation, accounting for 74% among all the generated patches.

Fifth, many edit scripts did not lead to a patch to the source code. This is because no mapping was built between the buggy code in the web page and the original source code for many edit scripts.

TABLE I: Details of generated fixes

Project	Issue No.	Loc	#Edit Scripts	#Patches					Time (sec)		
				Initial	Equivalent	Compile Error	Remaining	Correct	First	Total	Compilation
Calligraphy	41	406	0	0	0	0	0	–	0.001	0.001	0
screen-notifications	23	846	6	1	0	1	0	–	30.205	30.205	12.187
TuCanMobile	27	2,849	8	20	2	12	6	Y	10.619	83.447	54.866
OpenIAB	62	7,053	8	1	0	0	1	Y	37.106	53.433	35.905
Android-Universal-Image-Loader	660	11,829	8	0	0	0	0	–	12.629	12.629	0
couchbase-lite-android	292	12,004	5	9	0	9	0	–	71.361	71.361	52.914
Onosendai	100	17,821	6	12	2	3	7	Y	6.845	70.080	62.945
LNReader-Android	62	21,276	3	1	0	0	1	Y	13.136	25.987	10.496
the-blue-alliance-android	252	24,094	5	1	0	1	0	–	15.949	15.949	7.099
open-keychain	217	31,038	9	9	1	6	2	Y	9.409	106.799	65.869
Ushahidi_Android	100	33,574	9	2	0	2	0	–	54.665	54.665	29.888
cgeo	457	36,963	8	11	1	3	7	N	15.500	93.372	62.235
cgeo	887	42,814	8	13	5	6	2	Y	5.729	43.697	34.343
TextSecure	1397	46,469	9	40	0	40	0	–	229.263	229.263	211.488
cgeo	2537	54,765	6	0	0	0	0	–	24.537	24.537	0
WordPress-Android	688	62,344	9	8	0	8	0	–	106.533	106.533	66.409
WordPress-Android	780	62,455	0	0	0	0	0	–	0.001	0.001	0
WordPress-Android	1320	62,895	9	5	1	3	1	Y	18.209	74.008	36.374
WordPress-Android	1484	65,307	1	0	0	0	0	–	9.133	9.133	0
WordPress-Android	1122	65,539	6	0	0	0	0	–	27.392	27.392	0
gnucash-android	221	68,158	11	0	0	0	0	–	7.146	7.146	0
cgeo	3991	68,202	12	8	0	3	5	Y	18.411	155.640	122.389
WordPress-Android	1928	71,485	8	1	0	0	1	N	14.122	35.444	12.891
calabash-android	149	93,146	10	30	0	30	0	–	161.855	161.855	143.842
Total	–	963,332	164	172	12	127	33	8	899.756	1492.577	1022.140

```
69 -        dialog.dismiss();
   +        if(dialog.isShowing())
   +            dialog.dismiss();
70      }
-----------------------------------------------------------
69 -        dialog.dismiss();
   +        if (dialog != null && dialog.isShowing())   dialog.dismiss();
70      }
```

Fig. 7: Patches for TuCanMobile #27

We further give some examples of generated patches[3]. In each of the figures shown below, the top one represents the original patch generated by the developers, and the bottom one represents the first generated patch by our tool.

First, for 7 of the 10 bugs, our tool generated correct patches. Among them, patches for 3 bugs are identical to those written by humans, and patches for 4 bugs are not identical, but are still correct. For example, in Fig. 7, the generated patch has one more condition that checks `dialog` is not null. This is a useful check that ensures no NullPointerException before using `dialog`.

Second, for 1 of 10 bugs, our tool generated a patch using `try` and `catch` blocks as suggested in the Stack Overflow page, shown in Fig. 8. The human patch invokes `isFinishing()` and returns when finished. In our patch, we surround `finish()` with `try/catch`, which deals with the same root cause. However, because the patch is different from the human patch, we consider it as a correct but not acceptable patch.

Third, for the rest 2 of the 10 bugs, our tool did not generate correct patches. For example, we generated a patch that deals with the same root cause as suggested by a Stack Overflow crash trace. Sometimes a crash trace contains a very long list of method invocations and the buggy file may be omitted in the bug report. Third, the remaining 2 bugs cannot be fixed because of compilation errors. For example, a buggy code snippet has a method declaration which should return an integer, while in the question post it is actually a `void` method. Developers could do a manual transformation while our approach cannot.

The time used is shown in the last three columns of Table I. We did not include the time to query Google because (1) such time greatly depends on the network condition and varies from locations to locations, and (2) the query time is very small compared to the total time and can be neglected. In our experiment, the time for querying Google for each bug is around several hundred milliseconds.

As Column "Total" in Table I shows, the longest time spent on a bug is 230s, and the time for each bug is 62.2s on average. Compilation accounts for 68.5% of the time, and costs 6.39s for each compiled patch on average. In addition, if we report the result as soon as we generate the first filtered patch, we can reduce the total time by 39.7% to 900s, about 37.5s for each bug on average.

Although compilation time does increase with respect to project size, project size is not the main deciding factor of the total executing time. The main deciding factor is how many generated patches need to be tested. This is mainly related to the number of code snippets in Q&A pages and the number of project files in call stacks. These factors are not directly related to the size of the source code. Therefore, our approach is able to scale to very large applications.

C. RQ2: Usefulness

To answer the second research question, we did a qualitative analysis to check whether we can complement state-of-art automatic bug fixing approaches. We examine existing approaches for bug fixing and identify four approaches that apply to our case: GenProg [3], RSRepair [9], PAR [4] and SPR [12]. Other approaches either cannot scale to the projects in our experiment [10, 20], or have special requirements such as contracts [11].

Existing search-based techniques such as GenProg and RSRepair assume that patches already exist in the project code. PAR uses human-written templates, and to instantiate templates it also searches in the project code. SPR uses condition synthesis to repair defects, and in other occasions the search space is also within the project code. We did not generate any patches that can be synthesized using only existing templates in PAR or condition forms in SPR, and therefore, for each bug that our approach successfully fixed, we first used a representative substring in both the human patch and our patch to check if there is any match in the source code using the *grep* command, and then manually analyzed the returned search list to see whether a patch can be synthesized. The result is shown in Table III. The first column shows each issue with at least one filtered patch generated by our approach. The second column shows the grep command we use. The third column shows whether a patch can be synthesized.

There is only one case where an identical patch can be synthesized. For cgeo #3991, we got a large number of `try` and `catch` blocks, which indicates that GenProg, RSRepair and SPR can fix the bug by inserting the blocks. In addition, PAR

TABLE III: Keyword matching in source code

Issue	Grep Command	Result
TuCanMobile #27	grep ”isShowing” -R .	N
OpenIAB #62	grep ”super.onDestroy” -R .	N
Onosendai #100	grep ”context.getApplicationContext” -R .	N
open-keychain #217	grep ”dismissAllowing” -R .	N
cgeo #887	grep ”image/jpeg” -R .	N
cgeo #887	grep ”image/*” -R .	N
LNReader-Android #62	grep ”super.onDestroy” -R .	N
Wordpress-Android #1320	grep ”commitAllowingStateLoss” -R .	N
cgeo #3991	grep ”isFinishing” -R .	N
cgeo #3991	grep ”\btry\b” -R .	Y
cgeo #3991	grep ”\bcatch\b” -R .	Y

does not contain the `try/catch` template, and cannot create a patch of this form. The result indicates that our approach can complement existing bug-fixing approaches. Note that a bug can be fixed in many different ways, so being unable to synthesize the patches in the above procedure does not necessarily indicate that they cannot fix the bug. Therefore, we are not concluding that our approach is "better" than other approaches, but showing that our approach can complement them. In essence, we are dealing with a defect class [21] different from other approaches.

D. Threats to Validity

The main threat to external validity is that the benchmark we use is small and may not be representative of real world benchmarks. However, all bugs we use are real world bugs, are from different projects, and throw different exceptions, which may cover a large class of real world bugs. Note that many existing studies [10, 22, 23] use generated bugs to evaluate their approaches, and many [3, 10] evaluated on real world bugs have benchmarks whose sizes are similar to or much smaller than ours.

The main threat to internal validity is that our manual validation of the patches may be wrong. To alleviate this threat, three authors mutually checked the result, and any patch with a slight doubt was not considered as correct.

V. Discussion

The number of crash bugs that can be fixed by humans via exploring Q&A sites are relatively small in *GitHub*. This is due to two reasons. First, in open repositories like *GitHub*, issues are not well maintained in many projects, and we only investigate bugs that contain patches. This greatly reduces the number of investigated bugs. Second, developers may encounter crash bugs during development, and may fix them immediately instead of creating an issue. While our approach can fix crash bugs that can be found in issue repositories, our approach can be used by developers in the development stage, or can be deployed to automatically fix crash bugs newly founded by testing.

Our approach is limited to situations that humans can fix the bug via looking into Q&A sites. As a result, if there are no correct patches in Q&A sites, we cannot generate a correct patch. However, because recurring bugs are common and the resources on Q&A pages continuously increase, our approach has the potential to fix more bugs than can be fixed currently.

In our experiment we did not run the projects. However, in the presence of test cases, our approach can be run automatically to filter out more erroneous patches, which can further increase the accuracy of our approach.

VI. Related Work

1) Automatic Bug Fixing: Recently there has been much progress on fixing general types of bugs. Existing research uses specifications [23, 11, 24] or test cases [8, 25, 22, 9, 4, 11, 24, 12] to evaluate the correctness of patches and guide the process of patch generation. GenProg [8, 25, 22] and RSRepair [9] assume that patches exist in the current project, and use search-based techniques to find the patches. PAR uses human-written templates to generate patches. AutoFix-E [11] and AutoFix-E2 [24] rely on contracts present in the software to generate fixes. SemFix [10] and DirectFix [20] use component-based program synthesis techniques to synthesize a correct patch. SPR [12] instantiates transformation schemas to repair program defects by using condition synthesis. Prophet [26] uses machine learning over a large code database to learn a probabilistic model that characterizes successful human patches, and uses this model to prioritize the search for correct patches. Fischer et al. [27] propose a semantics-based approach that turns a given program into one whose evaluations under the error-admitting semantics agree with those of the given program under the error-compensating semantics. Gopinath et al. [23] use behavioral specifications to generate likely bug fixes. WAutoRepair [28] reduces patch validation time by only recompiling the altered components of a program. MintHint [29] is a semi-automatic approach that generates repair hints to help developers complete a repair, and it uses statistical correlation analysis to identify expressions that are likely to appear in the patches. Our work is different from these approaches in that we handle the defect class of recurring bugs whose fixes can be found in Q&A sites, and can complement the above approaches. Nguyen et al. [2] also study recurring bug fixes for Object-Oriented programs, but they do not analyze Q&A sites like us.

Automatic approaches to fixing specific types of bugs also exist. Jin et al. [30, 31] automate the whole process of fixing concurrency bugs. Xiong et al. [32] propose a new language to support the fixing of MOF models. Wang et al. [33] propose a dynamic-priority based approach to fixing inconsistent feature models. Rangefix [34] generates range fixes for software configuration. Caramel [35] generates non-intrusive fixes for performance bugs. Leakfix [36] generates safe fixes for memory leaks. Our work aims to fix crashes, different from the existing research.

2) Fault Localization: Before fixing the bugs, it is essential to locate where the bug occurs. A typical technique is spectra-based fault localization [37, 38, 39, 40], which uses program spectrum collected during execution. Because crash bugs have crash traces which contain location information, in our work we use this information to locate crash bugs statically.

CrashLocator [5] locates faulty functions by using crash traces and expanding the stack in a static call graph. Similar to spectra-based approaches, it calculates the suspiciousness score for each function and return a ranking list. However, this approach only ranks functions instead of statements, and thus cannot be used in our approach.

Another main line of fault-localization research is bug-report-oriented fault localization [41, 42, 43, 44, 45], which aims to find a small subset of source files that is related to a bug report among the entire code base. Because we focus on using only call stacks instead of bug reports for file-level fault localization, we do not leverage these approaches.

3) Q&A Site Retrieval and Analysis: Q&A sites contain rich resources for software engineering. Regarding retrieval from Q&A sites, SeaHawk [46] and Prompter [47] construct queries based on the code context, and retrieve API names and code-like words from Stack Overflow. However, for crash bugs it is difficult to retrieve Q&A pages with code context query. Rigby et al. [14] extract essential code elements from informal documentation such as Stackoverflow. Because Q&A pages related to bug fixes often contain code snippets in HTML tab pairs, we only use heuristics to extract code snippets. Cordeiro et al. [48] process crash traces and use it to retrieve Q&A resources. This approach uses exceptions and references of the crash trace as a query, and cannot distinguish the messages from the client and the framework.

There is also much research in analyzing Q&A sites. In artificial intelligence research, there are approaches for finding similar questions [49, 50], or finding the most appropriate answer [51, 52, 49, 53, 54]. In software engineering research, Henβ et al. [55] propose an approach to extracting FAQs from mailing lists and forums automatically. Wong et al. [56] propose an automatic approach to generating comments by mining Q&A sites. These approaches tackle different problems in analyzing Q&A sites compared to ours.

4) Code Differencing: The technique we use in analyzing Q&A sites is code differencing. ChangeDistiller [15] is a widely-used approach that builds mappings and generates edit scripts at AST level. GumTree [16] improves ChangeDistiller by removing the assumption that leaf nodes contain a significant amount of text, and it detects move actions better than ChangeDistiller. Chawathe et al. [17] propose an optimal and linear algorithm that generates edit scripts based on AST mappings. We chose GumTree for edit script generation, because it is the state-of-art work in this area.

Sydit [57] and LASE [58] generate program transformations from one or multiple examples. They generate context-aware, abstract edit scripts and then apply the edit scripts to new locations. Because in Q&A sites there is often no complete code snippet and sometimes only one statement, it is difficult to abstract the context, making the approaches not applicable in our work.

VII. Conclusion and Future Work

This paper proposes an automatic approach to fixing crash bugs via analyzing Q&A sites. By extracting queries from the framework and using a search engine to get a list of Q&A pages, we analyze the code in each page, obtain and apply edit scripts to source code. After that, we filter out redundant and incorrect patches, and only report the first patch to the developers. The experiments in real-world crash bugs show that our approach is accurate and scalable in large programs. Our approach complements existing bug fixing techniques by handling a different defect class.

In the future, we could study empirically on a larger dataset to investigate how many bugs could be fixed using our approach.

Acknowledgement

This work is supported by the National Basic Research Program of China under Grant No. 2014CB347701, and the National Natural Science Foundation of China under Grant No. 61202071, 61225007, 61432001, 61332010, 61421091.

References

[1] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of bug fixes," in *SIGSOFT '06/FSE-14*, 2006, pp. 35–45.

[2] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *ICSE '10*, 2010, pp. 315–324.

[3] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 54–72, Jan 2012.

[4] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE '13*, 2013, pp. 802–811.

[5] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: Locating crashing faults based on crash stacks," in *ISSTA 2014*, 2014, pp. 204–214.

[6] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *OOPSLA*, 2003, pp. 78–95.

[7] H. Seo and S. Kim, "Predicting recurring crash stacks," in *ASE*. ACM, 2012, pp. 180–189.

[8] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE '09*, 2009, pp. 364–374.

[9] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *ICSE 2014*, 2014, pp. 254–265.

[10] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *ICSE '13*, 2013, pp. 772–781.

[11] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *ISSTA 2010*, 2010, pp. 61–72.

[12] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *ESEC/FSE'15*, 2015.

[13] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *OOPSLA '08*, 2008, pp. 313–328.

[14] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *ICSE '13*, 2013, pp. 832–841.

[15] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *Software Engineering, IEEE Transactions on*, vol. 33, no. 11, pp. 725–743, Nov 2007.

[16] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus, "Fine-grained and accurate source code differencing," in *ASE '14*, 2014, pp. 313–324.

[17] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," *SIGMOD Rec.*, vol. 25, no. 2, pp. 493–504, Jun. 1996.

[18] "Eclipse ast parser," http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html.

[19] Z. Qi, F. Long, S. Achour, and M. Rinard, "Efficient automatic patch generation and defect identification in kali," in *ISSTA*, 2015, p. to appear.

[20] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *ICSE '15*, 2015.

[21] M. Monperrus, "A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair," in *ICSE 2014*, 2014, pp. 234–242.

[22] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE '12*, 2012, pp. 3–13.

[23] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using sat," in *TACAS'11/ETAPS'11*, 2011, pp. 173–188.

[24] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," in *ASE '11*, 2011, pp. 392–395.

[25] W. Weimer, Z. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, 2013, pp. 356–366.

[26] F. Long and M. Rinard, "Prophet: Automatic patch generation via learning from successful human patches," MIT, Tech. Rep. MIT-CSAIL-TR-2015-019, 2015.

[27] B. Fischer, A. Saabas, and T. Uustalu, "Program repair as sound optimization of broken programs," in *TASE 2009*, 2009, pp. 165–173.

[28] Y. Qi, X. Mao, Y. Wen, Z. Dai, and B. Gu, "More efficient automatic repair of large-scale programs using weak recompilation," *Science China Information Sciences*, vol. 55, no. 12, pp. 2785–2799, 2012.

[29] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "Minthint: Automated synthesis of repair hints," in *ICSE 2014*, 2014, pp. 266–276.

[30] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *PLDI '11*, 2011, pp. 389–400.

[31] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *OSDI'12*, 2012, pp. 221–236.

[32] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, "Supporting automatic model inconsistency fixing," in *ESEC/FSE '09*, 2009, pp. 315–324.

[33] B. Wang, Y. Xiong, Z. Hu, H. Zhao, W. Zhang, and H. Mei, "A dynamic-priority based approach to fixing inconsistent feature models," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, 2010, vol. 6394, pp. 181–195.

[34] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, "Range fixes: Interactive error resolution for software configuration," *Software Engineering, IEEE Transactions on*, vol. 41, no. 6, pp. 603–619, June 2015.

[35] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and fixing performance problems that have non-intrusive fixes," in *ICSE '15*, 2015.

[36] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe memory-leak fixing for c programs," in *ICSE '15*, 2015.

[37] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02, 2002, pp. 467–477.

[38] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *SIGPLAN Not.*, vol. 38, no. 5, pp. 141–154, May 2003.

[39] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, “An evaluation of similarity coefficients for software fault localization,” in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, ser. PRDC ’06, 2006, pp. 39–46.

[40] D. Hao, L. Zhang, T. Xie, H. Mei, and J.-S. Sun, “Interactive fault localization using test information,” *Journal of Computer Science and Technology*, vol. 24, no. 5, pp. 962–974, 2009.

[41] J. Zhou, H. Zhang, and D. Lo, “Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports,” in *Software Engineering (ICSE), 2012 34th International Conference on*, June 2012, pp. 14–24.

[42] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, “Combining probabilistic ranking and latent semantic indexing for feature identification,” in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 2006, pp. 137–148.

[43] ——, “Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval,” *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 420–432, Jun. 2007.

[44] D. Kim, Y. Tao, S. Kim, and A. Zeller, “Where should we fix this bug? a two-phase recommendation model,” *Software Engineering, IEEE Transactions on*, vol. 39, no. 11, pp. 1597–1610, Nov 2013.

[45] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, “Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis,” in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, Sept 2014, pp. 181–190.

[46] L. Ponzanelli, A. Bacchelli, and M. Lanza, “Leveraging crowd knowledge for software comprehension and development,” in *CSMR ’13*, 2013, pp. 57–66.

[47] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, “Mining stackoverflow to turn the ide into a self-confident programming prompter,” in *MSR 2014*, 2014, pp. 102–111.

[48] J. Cordeiro, B. Antunes, and P. Gomes, “Context-based recommendation to support problem solving in software development,” in *RSSE ’12*, 2012, pp. 85–89.

[49] V. Jijkoun and M. de Rijke, “Retrieving answers from frequently asked questions pages on the web,” in *CIKM ’05*, 2005, pp. 76–83.

[50] J. Jeon, W. B. Croft, and J. H. Lee, “Finding semantically similar questions based on their answers,” in *SIGIR ’05*, 2005, pp. 617–618.

[51] R. Burke, K. Hammond, V. Kulyukin, S. Lytinen, T. N., and S. Schoenberg, “Question answering from frequently asked question files: Experiences with the faq finder system,” *AI magazine*, vol. 18, no. 2, 1997.

[52] C. Kwok, O. Etzioni, and D. S. Weld, “Scaling question answering to the web,” *ACM Trans. Inf. Syst.*, vol. 19, no. 3, pp. 242–262, Jul. 2001.

[53] S. Harabagiu and A. Hickl, “Methods for using textu-