Automated Fixing is the Future

OpenRefactory is advancing software development by providing a sophisticated service called Intelligent Code Repair (iCR) to help programmers develop higher quality, more secure software in less time.

Software is expensive to create and support, while flaws cause significant customer problems due to losses in sales or revenue and distrust of the software provider. iCR reduces developer cost and risk while removing potential security holes and eliminating common bugs caused by human error.

iCR is unique because using automation to accurately rewrite code very difficult. This White Paper provides technical context and detail surrounding the core principles of iCR to help developers understand how it works.

The Problem

Software is created by human beings, who, being human, are prone to making mistakes—lots of mistakes. Fred Brooks, in his classic text on the challenges of software development, says this about programmers:

> *First, on*e must perform perfectly. The computer resembles the magic of legend in this respect, too. If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work. Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program.[1]

The mistakes show up as bugs in the software. Sometimes the effects are insignificant, but at other times the errors can be catastrophic. For example, the famous Heartbleed security flaw of 2016 was estimated to have caused around half a billion dollars to correct.

Coralogix[2], the data logging analytics company, studied the issue of developer productivity and makes the following claims:

> “5 amazing facts on exactly how much time is spent on debugging and code fixing in the software industry:
>
> 1. On average, a developer creates 70 bugs per 1000 lines of code (!)
> 2. Fifteen bugs per 1,000 lines of code find their way to the customers
> 3. Fixing a bug takes 30 times longer than writing a line of code
> 4. 75% of a developer’s time is spent on debugging (1500 hours a year!)
> 5. In the US alone, $113B is spent annually on identifying and fixing product defects
>
> So if you thought that your developers are spending their time on making your dream a reality… you better think again, most of your budget is spent on debugging, and when debugging takes a lot of time, versions are delayed.”

Many of these bugs are logic bugs associated with the intended behavior of the software. A large number, though, arise from common coding errors. And some of these coding errors can have serious security consequences, such as the Heartbleed bug noted earlier, where it was shown that the problem was in the code for two years before being detected.

Developers don't set out to write buggy code and, what's more, they hate fixing bugs. The compressed time frame under which they work is to blame. What is needed is a tool that enables developers to work as quickly and error free as possible, a tool that truly solves the problem of bug fixing. This would lead to two major positive outcomes: happier developers (key to retention in the face of fierce competition for talent) and a better product (more and better features; bug free).

Intelligent Code Repair (iCR)

Automated tools are available to detect programmer errors. The tools generate a swarm of warnings that developers must analyze to determine if the warnings are valid. This task is laborious and time-consuming, partly because of the high number of false positives produced by the tools.

OpenRefactory's iCR is challenging this established industry by asking, "What if there is a way to not just discover possible coding errors, but to actually repair them, automatically and correctly every time?" The mature detection-only services do help lower the total number of bugs released in the product, but do nothing to reduce the time it takes to achieve a quality product.

The confluence of key technologies

Until now, there was a sound reason why automation has only addressed the detection side of the issue. Fixing the flaw is much harder than detecting it. As an analogy, it is much easier for a dentist to identify a cavity than it is to drill it out and fill it.

In terms of automation, a key development has been the integration of machine intelligence into the process of altering source code. At the University of Illinois at Urbana-Champaign (UIUC), work on Code Refactoring has been going on for nearly three decades. OpenRefactory's CEO and cofounder, Dr. Munawar Hafiz, received his Ph.D. from UIUC in 2009, where he focused on using emerging code refactoring techniques to accurately rewrite fragments to fix security problems in software.

His approach to refactoring, however, challenged the conventional understanding of refactorings. Refactorings are usually behavior-preserving changes to improve the maintainability of the code. Dr. Hafiz, on the other hand, introduced behavior-enhancing refactorings that not only preserve behavior in the normal path of a program, but also modify the behavior to fix a security, reliability, or a compliance issue.

For example, a program with an SQL injection problem will allow an attacker to access unathorized data. A behavior-enhancing refactoring modifies the program so that a user's database queries are processed normally, but an attacker's malicious database queries are filtered by the program.

The major roadblock that behavior-enhancing refactoring faced was that the level of code analysis it requires is more extensive than that typically performed by traditional refactoring and detection-only technologies. Dr. Hafiz's dissertation and subsequent research work has demonstrated that by augmenting code refactoring with deep analysis and increased machine intelligence behavior-enhancing refactoring (Intelligent Code Repair) can be realized.

The outcome of Dr. Hafiz's research of a decade ago is the "now practical" implementation of iCR. It has made the automatic fixing of coding bugs possible on an enterprise level.

The Creation of Fixers

The core of the iCR service is the Analysis Engine, which incorporates a broad suite of behavior-enhancing refactorings. These are referred to as Fixers.

Each fixer addresses a specific class of security, reliability, or compliance problem:

Security Fixers target a specific security problem to prevent attackers from taking control of the system, stealing data, and/or crashing applications. These will target the most important problems in each programming language, as described by the lists created by OWASP, SANS, etc. For example, a fixer for Java programs addresses SQL Injection issues that may allow an attacker to access or steal information.

Reliability Fixers target problems that cause an application to crash or slow down, or hamper the user experience. For example, a fixer for Java programs addresses a resource leak that may allow an attacker to unexpectedly crash the application.

Compliance Fixers target compliance issues. Standards organizations such as CERT define guidelines to eliminate insecure coding practices. Some of these may be structural issues that involve code smell, while others may be associated with exploitable vulnerabilities. Compliance fixers address these issues to make code more robust. For example, a fixer for Java programs may check that the fields of a Java class are not directly set from outside. That is, a field declared as public and directly accessed from other classes should be modified to be a private field and should only accessible through getter/setter methods.

Fixers address the entirety of what a code repair would entail. For example, in the aforementioned case of a Java field variable being declared public when it should be private, the fixer first corrects the declaration of the variable in the class where it is declared by changing it to private. Then the fixer repairs all references to the variable and corrects them from variable references to the appropriate set or get accessor methods. This demonstrates the complexity of the task of accurate code correction .

Why iCR?

While it may seem intuitive that automatically fixing bugs is a good thing, there are tangible reasons why automated fixing is more than just an incremental tool.

Accuracy

iCR uses proprietary mathematical models to determine how an error is to be corrected. While one might think that a correction is a 'simple text replacement' process, this is definitely not the case. Such a method will only work in limited cases and is also prone to error. Instead, iCR Fixers use models of the semantic behavior of the code in order to ensure that the replaced code will behave the same.

Well, almost the same. Traditional code refactoring models are used to ensure that refactored code semantics exactly match the code they are replacing. With iCR, the replaced code is adjusted slightly to remove the offending code. This method ensures that the correction is always accurate

In order to ensure that the developer does not miss a critical issue, current detection services err on the side of more false positives. Typical numbers in the industry for false positives range in excess of 50%. This means that one out of every two (or more) warnings is a false positive. These warnings must be manually reviewed, leading to these issues:

- Engineering time is spent reviewing the warning. Additional engineers are usually part of the triage process;
- If it is determined by the engineering team that the warning is false, the team must teach the tool to "silence" the warning;
- A problem may be incorrectly tagged as a false positive when it is an actual bug. In a worst-case scenario, the problem is "silenced" so that the bug remains in all future releases of the software.

Because iCR requires extensive knowledge of the source code in order to rewrite it, the analysis it performs and the metadata that it collects are much more extensive than current detection-only technologies.

Quality

One of the main objectives in performing static analysis is to improve code quality. As noted in the Coralogix report, typical software releases contain, on average, 15 bugs per 1,000 lines of code. This occurs despite rigorous development practices that include design reviews, code reviews, and extensive testing. Having a tool that can detect and correct potentially dangerous security vulnerabilities not only saves time in development, it also reassures your customers that your delivered software is safe from potential breaches.

Productivity

Highlighted in the Coralogix blog[3] is the finding that about 75% of a developer's time is spent debugging. This translates to about 1500 hours per year. About 30% of the bug fixing time (~23% of total time) is spent on fixing security bugs.[4]

Conservatively assuming that OpenRefactory tools will be able to fix 60% of the bugs and developers will accept 80% of these fixes, the use of these tools will translate into about an 11% savings in terms of developer hours. An 11% savings on effort means this time can be used in building more features, fixing other bugs that may have been left unfixed due to limited development time, or a quicker time to market.

Innovation

Developers don't like chasing bugs, and they really don't like triaging for false positives. Using existing tools with high false positive rates not only consumes valuable time, it is tiresome for developers. In many cases, the warnings are for issues that the development managers consider trivial and too time-consuming to fix manually. With iCR handling these minor issues, false positives no longer bog down developers.

This frees up developer time allowing them to spend more time doing what they enjoy the most: creating value and improving your product.

What Does iCR Contain?

The iCR service can operate on a transactional basis for single scans of a project, referred to as an Audit. Or, iCR can be integrated into a regular CI/CD operation as a service. The iCR software is comprised of an iCR Analysis Engine and an iCR Reviewer.

Both the iCR Analysis Engine and the iCR Reviewer run as Docker images that can be easily installed and executed on any laptop or desktop computer with the Docker container infrastructure installed. The container framework is widely accepted by developers as a safe and reliable mechanism for running third party software on a developer's system. From the Docker site: "Docker provides a way to run applications securely isolated in a container, packaged with all its dependencies and libraries." Learn more about Docker at: https://docs.docker.com.

The iCR Analysis Engine

OpenRefactory's iCR Analysis Engine examines a project using OpenRefactory's Fixers. Each fixer is a proprietary algorithm that the Engine uses to detect and correct specific classes or types of bugs. Currently, OpenRefactory provides iCR Fixers for both Java and C. Support for other languages will be forthcoming in future releases of the service.

The Engine parses source code and generates metadata to be used for analysis and code rewriting. Then, the fixers are executed that scan the source for known coding errors and a set of code alterations are produced. From that set, source code diff files are created, used both to present the changes to the developer and to create the information needed to effect the changes in the source code.

An Example of iCR for Java

The code fragment below implements a method called `getMaterialCost()`. It returns either a value or -1 if there is an error.

```
private int getMaterialCost(ItemStack itemstack) {
    if (itemStack.getItem() instanceof ItemTool) {
        ItemTool tool = (ItemTool) itemStack.getItem();
        return ToolMaterial.valueOf(tool.getToolMaterialName()).
                ordinal();
    } else {
        return -1;
    }
}
```

The code fragment below invokes this method. Note that the call does not check for the error return code. This is an opportunity for the error to cause a problem.

```
} else if (v == value) {
    int c = getMaterialCost(itemStack);
    if (c < materialCost) {
        value = v;
        best = i;
        materialCost = c;
    }
}
```

iCR detects that the method returns one or more values that are not being checked by the programmer. Having done so, iCR flags the offending fragment and inserts a code block to handle the error return. If it happens that the method might return multiple error codes, iCR can even add code blocks to make sure that all return paths are checked.

The last code block shows the corrected code with the warning that the developer should insert the behavior to be executed on the error. iCR's machine-learning component will add error handling code in a future release.

```
} else if (v == value) {
    int c = getMaterialCost(itemStack);
    // OR Warning: Handle Error Code
    if (c == -1) {
    }
    if (c < materialCost) {
        value = v;
        best = i;
        materialCost = c;
    }
}
```

An Example of iCR for C

It is common for C programs to have signedness (using a signed integer in an unsigned context and vice versa) and widthness (assigning an integer with larger memory size to an integer with a smaller memory size) problems from using variable types in incorrect contexts. These errors derive from incorrectly declared variables.

The Change Integer Type fixer changes the declared type of variables so that the uses of the variable are not conflicting with the declaration.

The following shows a vulnerability in `libpng` v1.4.9 (CVE-2011-3026), in lines 267-290 in `pngrutil.c` file.

```
267 int ret, avail;
268 …
276 avail = png_ptr->zbuf_size-png_ptr->zstream.avail_out;
277 …
283 if (output != 0 && output_size > count)
284 {
285      int copy = output_size - count;
286      if (avail < copy) copy = avail;
287      png_memcpy(output + count, png_ptr->zbuf, copy);
288 }
289
290 count += avail;
291 …
```

The fixer determines that `avail` should be an `unsigned int` based on the fact that `zstream.avail_out` was previously understood to be of type `unsigned int`. It then identifies that the variables `avail` and `copy` are declared as integers but are used as unsigned integers in all important contexts. They should be declared as `unsigned int`.

The next code fragment shows how the C fixer detected the mismatched type declarations and corrected the code for declaring both `avail` and `copy` as `unsigned int` types.

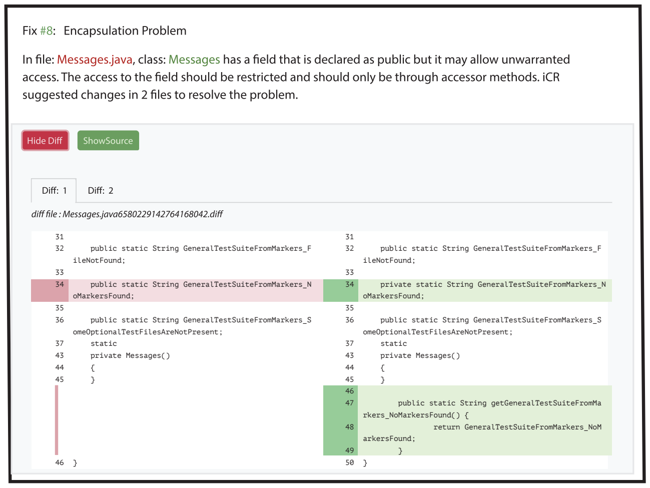
```
267 int ret, avail;
268 …
267 int ret;
268 unsigned int avail;
269 …
277 avail = png_ptr->zbuf_size-png_ptr->zstream.avail_out;
278 …
284 if (output != 0 && output_size > count)
285 {
286      unsigned int copy = output_size - count;
287      if (avail < copy) copy = avail:
288      png_memcpy (output + count, png_ptr->zbuf, copy);
289 }
290
291 count += avail;
292 …
```

A more detailed discussion of this example and others can be found in the paper, *Program Transformations to Fix C Integers*[5].

The ICR Reviewer

Once the iCR Analysis Engine has executed its scan of the code, the iCR Reviewer is used to browse through the fixes that were generated. The Reviewer uses a "diff" window so the developer can see the original code alongside the fixes that were generated. The developer can also use the Reviewer to browse all the source code in the affected file if desired.

The Reviewer allows the developer to examine a flaw in the original code and compare it with the correction. Although the fixes are generated automatically, the process allows the developer to make the final decision to accept a change or not. If accepted, the developer can apply the changes to the source itself. This example of the Reviewer window shows a summary of the problems discovered, the original code and the corrections that were automatically generated.


Fix #8: Encapsulation Problem

In file: Messages.java, class: Messages has a field that is declared as public but it may allow unwarranted access. The access to the field should be restricted and should only be through accessor methods. iCR suggested changes in 2 files to resolve the problem.

diff file : Messages.java6580229142764168042.diff

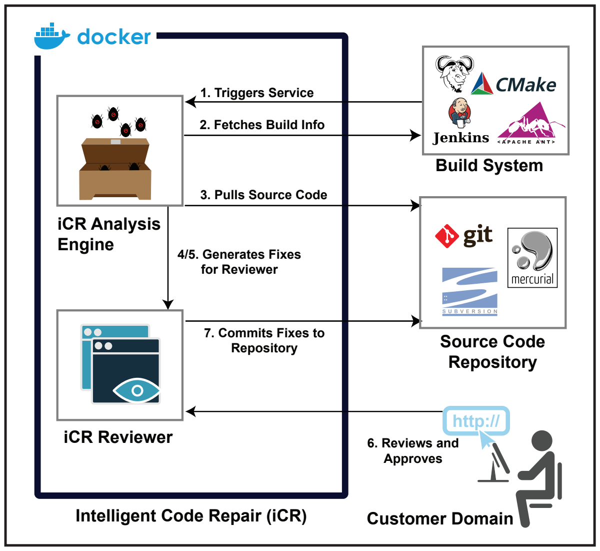
```

    public static String GeneralTestSuiteFromMarkers_F
ileNotFound;

    public static String GeneralTestSuiteFromMarkers_N
oMarkersFound;

    public static String GeneralTestSuiteFromMarkers_S
omeOptionalTestFilesAreNotPresent;
    static
    private Messages()
    {
    }
}
```

```

    public static String GeneralTestSuiteFromMarkers_F
ileNotFound;

    private static String GeneralTestSuiteFromMarkers_N
oMarkersFound;

    public static String GeneralTestSuiteFromMarkers_S
omeOptionalTestFilesAreNotPresent;
    static
    private Messages()
    {
    }

        public static String getGeneralTestSuiteFromMa
rkers_NoMarkersFound() {
                return GeneralTestSuiteFromMarkers_NoM
arkersFound;
        }
}
```

How Does It Work?

iCR can be operated either in a single-scan transactional mode or integrated into a regular CI/CD process. In either case, the sequence of operations is much the same.

One of the key principles in using iCR is the preservation of the privacy of the developer's source code. Most software companies would balk at allowing some kind of external access to their source code in order for it to be analyzed. Therefore, iCR is deployed on the developer's site and installed into safe Docker containers. The diagram shows how the process flow is arranged.



The developer either triggers the Audit by invoking the Analysis Engine directly from within Docker, or the SaaS service may be triggered in a CI/CD system automatically by build scripts (Step 1). In either case, the Analysis Engine is invoked.

The Analysis Engine, from arguments given to it on invocation, can access the build information (Step 2). This allows the Engine to locate necessary files such as classpaths for Java or #include files for C. With that information, the Engine can access the source files themselves (Step 3) and analysis begins.

The Engine parses and builds intermediate metadata for analysis. The time this step takes depends on the size of the project, with large projects possibly requiring many hours, due to the complex and deep analysis the Engine performs. The fixers execute within the iCR Analysis Engine and many run in parallel (Step 4). However, some are dependent upon certain internal metadata and so may take longer to complete, as they are serialized behind internal tasks.

When the analysis is complete, the fixers generate corrections (Step 5). The results are saved in a file outside the Docker container so that they are available for review once the Engine terminates. The Reviewer can then be invoked, and the developer can review each set of corrections (Step 6). The developer may then choose to accept the fix and apply it back to the source code repository (Step 7).

Although changes can be accepted automatically, OpenRefactory recommends that the manual step of approval be performed. This makes sure that a human developer is aware of all the fixes that were generated, and may also help the developer learn how better code can be created at the outset. The educational value of iCR is an additional benefit.

[1] *The Mythical Man-Month*, Frederick Brooks, p. 8

[2,3] Coralogix Blog: *"This is what your developers are doing 75% of the time, and this is the cost you pay"*;
https://coralogix.com/log-analytics-blog/this-is-what-your-developers-are-doing-75-of-the-time-and-this-is-the-cost-you-pay/

[4] OpenRefactory customer discovery study, 2016. Presented at 2016 NSF SBIR/STTR Phase I Grantee Fall Workshop

[5] Z. Coker and M. Hafiz; *Proceedings of 35th International Conference on Software Engineering* (ICSE 2013)

Learn More about iCR

Please contact us at info@openrefactory.com if you have questions or suggestions on the operation of iCR for C or iCR for Java.