Weak-Assert: A Weakness-Oriented Assertion Recommendation Toolkit for Program Analysis

Cong Wang1, Yu Jiang1∗, Xibin Zhao1, Xiaoyu Song2, Ming Gu1, Jiaguang Sun1
1 Beijing National Research Center for Information Science and Technology (BNRist)
School of Software, Tsinghua University, Beijing, China
2 Department of ECE, Portland State University, Portland, Oregon, 97201, USA

ABSTRACT
Assertions are helpful in program analysis, such as software testing and verification. The most challenging part of automatically recommending assertions is to design the assertion patterns and to insert assertions in proper locations. In this paper, we develop Weak-Assert1, a weakness-oriented assertion recommendation toolkit for program analysis of C code. A weakness-oriented assertion is one of discovered weakness. In this code snippet, variable 

There are only a few tools for assertion generation. Some of them focus on the hardware area. Shobha et al present GoldMine, a methodology for generating assertions [15]. In their method, they generate assertions based on RTL (Register Transfer Level Design). Zhang et al [6, 18] generate security critical properties to verify the hardware processor. In the software area, Long et al [11] generate assertions in Java programs by using active learning techniques. Their work focuses on the correlation between the learned assertions and the occurrence of a test case failure.

An assertion, which can help to find potential program weaknesses, is called a weakness-oriented assertion. The most challenging part of recommending such assertions is to design the weakness patterns and to insert assertions in proper locations. Fig. 1 shows an example of discovered weakness. In this code snippet, variable data is declared as an integer in Line 6. This means the value of data relies on the first parameter of main function. The parameter can be negative. A weakness occurs when the system function memcpy is called in Line 14. memcpy requires an "unsigned int" value for the third parameter, but this example uses variable data, which means a negative input may trigger a weakness in Line 14. We can insert an assertion "assert(data>=0)" (Line 13). This assertion can help to find the weakness by using program testing or verification techniques. Therefore this assertion is a weakness-oriented assertion.

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 int main(int argc, char* argv[])
5 {
6     int data = atoi(argv[1]);
7     char source[100];
8     char dest[100] = "";
9     if(data<100)
10     {
11         memset(source,"A",100-1);
12         memcpy(dest,source,data);
13         dest[data] = '\0';
14     }
15     return 0;
16 }

Figure 1: Example for Discovered Weakness

We develop Weak-Assert, a weakness-oriented assertion recommendation toolkit for program analysis of C code. Weak-Assert applies to several types of program weaknesses, which qualify for inclusion in Common Weakness Enumeration (CWE)2. Weak-Assert uses well-designed patterns (such as CWE134, CWE195, etc)

1Yu Jiang is the correspondence author, and the work is supported by Tsinghua-Huawei Collaboration Project YBN2017010031
2The address of the tool, user manual and online version is: http://congwang92.cn/wa

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
https://doi.org/10.1145/3183440.3183471
to match the abstract syntax trees of source code automatically. It collects meaningful messages (such as variable names, line numbers, etc.) from trees and inserts assertions into proper locations of programs. These assertions can be verified by using program analysis techniques, such as program testing and verification. In short, Weak-Assert needs a C source file and specified CWE types as input. Automatically, Weak-Assert inserts weakness-oriented assertions (if needed) into the source code and finally produce a new file, which can be used to test or verify. We package Weak-Assert as a toolkit, which can be used in a command line and online website conveniently.

The experiments are set up on Juliet test suite [13] and several actual projects in Github. These actual projects consist of 26 distinct C language projects, containing 38383 C files in total. Experimental results show that Weak-Assert helps to find 125 program weaknesses in 26 actual projects. These weaknesses are confirmed manually to be triggered by some test cases.

2 RELATED WORK

Assertion Generation. There are only a few tools for assertion generation. Some of them focus on the hardware area. Shobha et al. present GoldMine, a methodology for generating assertions [15]. In their method, they generate assertions based on RTL (Register Transfer Level Design). GoldMine is able to generate many assertions per output in very reasonable runtimes. They claim that their work is the first attempts to generate assertions through data mining and static analysis of RTL source code. Zhang et al. [6, 18] generate security critical properties to verify the hardware processor. Their approach uses known design errata and machine learning techniques to find invariants of hardware processor. Generated assertions can be used in program verification and are critical to security. In the software area, Long et al. [11] generate assertions in Java programs by using active learning techniques. Their method tries to avoid heavy-weight techniques like the symbolic execution. Their work focuses on the correlation between the learned assertions and the occurrence of a test case failure. Our previous work uses machine learning techniques to decide whether programs need assertions [16].

Program Analysis. There exist many works on program analysis, including program verification, program testing, etc. On program verification, Henzinger [5] and Jhala [7] and Yu [8, 17] use model checking techniques to verify the property of programs. They check temporal logic properties of the models which are constructed from programs. Cater et al. [2] provide a software verification ecosystem (SMACK) based on LLVM compiler. SMACK is chosen to be integrated into our tool because it is a state-of-the-art program verification tool to win the championship of ReachSafety Track of SV-COMP’17 (2017 6th International Competition on Software Verification). As for program testing, there also exist many powerful techniques [9] [3]. Our tool generates assertions automatically. Program testing approaches can be used manually to check whether programs’ behaviors satisfy these assertions.

Our Difference of Weak-Assert. Previous assertion generation techniques are little relevant to program weakness. We develop a weakness-oriented assertion recommendation toolkit for program analysis. Our tool is able to recommend specific assertions, which can help to find potential program weaknesses.

3 WEAK-ASSERT DESIGN

The overall framework of Weak-Assert is shown in Fig. 2. It contains three modules: input processing module, code parsing module and assertion recommend module. Firstly users can set the initial parameters of Weak-Assert. Through specifying the parameters, users provide the path of the source file or directory and choose the weakness types. Then Weak-Assert parses the code of source files to generate the abstract syntax trees. Finally, Weak-Assert matches the abstract syntax trees with designed patterns and insert assertions into proper locations automatically.

3.1 Main Modules

Input processing module. Firstly, the initial parameters of the execution are set in the input processing module. Users are able to apply Weak-Assert to a directory (the entrance of C language projects) or a single source file. Program weakness types also can be chosen. Weak-Assert supports six different types (shown in Table 1) of common weakness defined in CWE (Common Weakness Enumeration) [12]. Besides, users can choose to verify the assertions inserted by Weak-Assert. We use an open source program verification tool, SMACK [2], to verify the assertions.

Table 1: Supported Weakness Types

<table>
<thead>
<tr>
<th>ID</th>
<th>Weakness Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>134</td>
<td>Use of Externally-Controlled Format String</td>
</tr>
<tr>
<td>195</td>
<td>Signed to Unsigned Conversion Error</td>
</tr>
<tr>
<td>823</td>
<td>Use of a String Pointer in a Location to NULL</td>
</tr>
<tr>
<td>824</td>
<td>Uninitialized Variable</td>
</tr>
<tr>
<td>789</td>
<td>Uncontrolled Memory Allocation</td>
</tr>
<tr>
<td>845</td>
<td>Loop with Unreachable Exit Condition (Infinite Loop)</td>
</tr>
</tbody>
</table>

The weakness types are shown in Table 1. 1. For example, the code snippet in Fig. 1 may suffer from the weakness "Signed to Unsigned Conversion Error" (ID.195). Variable data is not defined as an unsigned integer, which gives it the chance to be negative. Also, we can give an example (shown in Fig. 3) for ID.134. This code snippet suffers from "Use of Externally-Controlled Format String". Executing this program in a command line, we get “Segmentation fault (core dumped)" because "%s, %s" in Line 6 is a kind of code attack to read values from stacks.
```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    char* data;
    char dataBuffer[100] = "";
    data = dataBuffer;
    strcpy(data, "%s,%sfix%edstringtest");
    fprintf(stdout, data);
}
```

Figure 3: Example for Weakness ID.134

**Code parsing module.** In code parsing module, Weak-Assert integrates libClang [14], which provides a C interface to an abstract syntax tree. The abstract syntax tree is filtered to remove those which are included from other files (no matter source files or head files).

**Assertion recommend module.** In assertion recommend module, Weak-Assert matches the abstract syntax trees with patterns, which are corresponding to weaknesses. Through traversing the entire tree, we capture the weakness types, locations and variable names. Then Weak-Assert inserts assertions to generate new files. These assertions are inserted into specified locations in order to be detected before the weaknesses are triggered. Finally, if users want to verify the assertions, which are recommended, Weak-Assert is able to call SMACK [2] to finish the verification tasks automatically.

To design the matching patterns is a significant task in our work. Every pattern corresponds to a type of program weakness. The pattern should identify the pivotal node in an abstract syntax tree. For example, weakness ID.674 “Uncontrolled Recursion” needs to ensure that the recursion would stop after a number of loops. We choose the node who has the attribute `kind` as `CursorKind.FUNCTION_DECL`. Then the function name can be captured from the attribute `displayname` of the node. The function name is saved temporarily before come to `CursorKind.FUNCTION_DECL` node. Thanks to the depth-first search strategy, we could identify whether this function is recursive, just before we come to another program function. Fig. 4 shows an example of this type of weakness. The statement in Line.4 recalls the program function itself but reserves no exits.

```c
void helperBad()
{
    /* FLAW: this function causes infinite recursion */
    helperBad();
}
```

Figure 4: Example for Weakness ID.674

Matching is not enough. Automatically Weak-Assert captures messages from the pivotal node in an abstract syntax tree. The messages include variable names and locations. We use the locations to slice the source file and insert the assertions spliced by variable names and string template. For example, the code snippet in Fig. 4 is transformed to Fig. 5. Manually we set a maximum number of iteration (Line. 4).

3.2 Usage

Weak-Assert can be used as a command line tool. The executing parameters are defined in Table 2. For example, users can execute a script as: `weakassert -d /XXX -w 134,195,789`. This script calls our tool to recommend assertions for C source files in directory "/XXX" especially for program weakness "CWE134, 195 and 789".

**Table 2: Parameter Description of Weak-Assert**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d &lt;directory&gt;</td>
<td>&lt;directory&gt; is the entrance of C language projects.</td>
</tr>
<tr>
<td>-f &lt;file&gt;</td>
<td>&lt;file&gt; is a single source file.</td>
</tr>
<tr>
<td>-w &lt;weakness ID&gt;</td>
<td>&lt;weakness ID&gt; is the list of weakness types, separated with comma.</td>
</tr>
<tr>
<td>-v</td>
<td>if you want to verify the assertions, use this parameter.</td>
</tr>
</tbody>
</table>

Besides, we provide an online version (Weak-Assert Online), which holds brief functions of Weak-Assert. Users can access it from web explorers [3]. Fig. 6 shows the interface of Weak-Assert Online. Users can write their C programs on the left side and press “Execute” button to insert weakness-oriented assertions. The new code snippet is displayed on the right side.

```c
int iterator_tempvalue = 0;
static void helperBad()
{
    iterator_tempvalue += 1;
    assert(iterator_tempvalue <= 100000);
    /* FLAW: this function causes infinite recursion */
    helperBad();
}
```

Figure 5: Modified Example for Weakness ID.674

4 EXPERIMENT

To evaluate the performance of Weak-Assert, we apply the tool to real-world projects. In this section, we describe the experimental setup and present the weaknesses detected by Weak-Assert.

**Experimental Setup** Weak-Assert is trained on Juliet test suite. The weaknesses in programs of specified type (ID.134, 195, 674, 690, 789, 835) can be detected by testing recommended assertions. We apply Weak-Assert on real-world projects, which are downloaded from Github [4]. Table. 3 shows the information of the projects. It contains 26 real C projects, with 38383 C language source files. Our experiment is to apply Weak-Assert on these projects to insert assertions. Then these assertions are verified by test cases manually.
is defined as an integer, which could be less than zero.


S. Schaub and B.A. Malloy. 2014. Comprehensive analysis of C++ applications

CongWang, FeiHe, XiaoyuSong, YuJiang, MingGu, and JiaguangSun. 2017. As-

Roy Budhai, Brian Chen, Teresa Su, and Sheldon Sequeira. 2016. Testing applica-

Yu Jiang, Hehua Zhang, Han Liu, William Hung, Xiaoyu Song, Ming Gu, and

Shobha Vasudevan, David Sheridan, Sanjay Patel, and David Tcheng. 2010. Gold-

Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre F Baldi.


Model Checking Software

Hehua Zhang, Yu Jiang, William NN Hung, Xiaoyu Song, Ming Gu, and Jiaguang


Disk(MB) C Files Weak(ID.195)

1.7 162 2

3.4 106 1

4.7 1022 1

1.7 162 2

1.1 50

1.0 0.868 20

1.7 92 2

3.1 267 1

3 106 1

2 72 2

1.7 84 2

1.5 162 2

1.1 50 2

0.872 4 1

0.686 20 1

0.768 25 4

0.764 34 1

0.48 36 3

0.316 12 1

0.284 14 1

1125.152 38383 125

Table 3: Actual Projects in Github

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Disk(MB)</th>
<th>C Files</th>
<th>Weak(ID.195)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.git-master</td>
<td>785</td>
<td>25568</td>
<td>33</td>
</tr>
<tr>
<td>2.php-src-master</td>
<td>76</td>
<td>1022 1</td>
<td></td>
</tr>
<tr>
<td>3.FFmpeg-master</td>
<td>54</td>
<td>2431 27</td>
<td></td>
</tr>
<tr>
<td>4.Cygwin-master</td>
<td>53</td>
<td>3374 10</td>
<td></td>
</tr>
<tr>
<td>5.scripts-master</td>
<td>47</td>
<td>2412 9</td>
<td></td>
</tr>
<tr>
<td>6.Telegram-master</td>
<td>21</td>
<td>521 4</td>
<td></td>
</tr>
<tr>
<td>7.AzureLabs-master</td>
<td>19</td>
<td>189 1</td>
<td></td>
</tr>
<tr>
<td>8.ohh-studio-master</td>
<td>12</td>
<td>668 2</td>
<td></td>
</tr>
<tr>
<td>9.vim-master</td>
<td>12</td>
<td>135 2</td>
<td></td>
</tr>
<tr>
<td>10.h2o-master</td>
<td>11</td>
<td>356 3</td>
<td></td>
</tr>
<tr>
<td>11.git-master</td>
<td>8.7</td>
<td>403 4</td>
<td></td>
</tr>
<tr>
<td>12.cppc-master</td>
<td>6.5</td>
<td>326 6</td>
<td></td>
</tr>
<tr>
<td>13.lily-master</td>
<td>3.1</td>
<td>267 1</td>
<td></td>
</tr>
<tr>
<td>14.pipeless-master</td>
<td>3</td>
<td>92 2</td>
<td></td>
</tr>
<tr>
<td>15.osfida-master</td>
<td>3</td>
<td>106 1</td>
<td></td>
</tr>
<tr>
<td>16.alliance-master</td>
<td>2</td>
<td>72 2</td>
<td></td>
</tr>
<tr>
<td>17.nmap-master</td>
<td>1.7</td>
<td>84 2</td>
<td></td>
</tr>
<tr>
<td>18.mini-master</td>
<td>1.7</td>
<td>162 2</td>
<td></td>
</tr>
<tr>
<td>19.pech7-master</td>
<td>1.1</td>
<td>50 2</td>
<td></td>
</tr>
<tr>
<td>20.Patchy-master</td>
<td>0.872</td>
<td>4 1</td>
<td></td>
</tr>
<tr>
<td>21.pg-master</td>
<td>0.868</td>
<td>20 1</td>
<td></td>
</tr>
<tr>
<td>22.modeshaded-master</td>
<td>0.768</td>
<td>25 4</td>
<td></td>
</tr>
<tr>
<td>23.getty-master</td>
<td>0.764</td>
<td>34 1</td>
<td></td>
</tr>
<tr>
<td>24.The-Act-OfProgram</td>
<td>0.48</td>
<td>36 3</td>
<td></td>
</tr>
<tr>
<td>25.the_silver_searcher</td>
<td>0.316</td>
<td>12 1</td>
<td></td>
</tr>
<tr>
<td>26.wifi-master</td>
<td>0.284</td>
<td>14 1</td>
<td></td>
</tr>
<tr>
<td>Data in total</td>
<td>1125.152</td>
<td>38383</td>
<td>125</td>
</tr>
</tbody>
</table>

Fig. 7: Example of Weakness Detected in Actual Projects

typedef struct {
  unsigned char *bytes;
  int max, count;
} Buffer;

void expand_buf(Buffer *buf, int len) {
  if (buf->max+len) {
    buf->max=len+20;
    if (buf->bytes) {
      buf->bytes=(unsigned char *) realloc (buf->bytes, buf->max);
    } else {
      assert (buf->max>0);
      buf->bytes=(unsigned char *) malloc (buf->max);
    }
  } else {
    if (buf->bytes) {
      buf->bytes=(unsigned char *) realloc (buf->bytes, buf->max);
    } else {
      assert (buf->max>0);
      buf->bytes=(unsigned char *) malloc (buf->max);
    }
  }
}

Experimental Results. We apply Weak-Assert on 26 different actual projects to insert assertions corresponding to CWE.195 (Signed to Unsigned Conversion Error). As shown in Table. 3, real weaknesses, which can be triggered by some test cases, are shown in column "Weak(ID.195)". Totally, we find 125 real weaknesses by applying Weak-Assert to insert assertions.

We classify the weaknesses into three groups based on the reasons: "result of undetermined calculation", "signed definition" and "influenced by input". "result of undetermined calculation" refers to the situation when the value of a variable is calculated by an undetermined operation. The operation may result in negative values. "signed definition" refers to the situation when the variable is defined as a signed one. In most cases, these variables are parameters of program functions, which have the chance to be less than zero. "influenced by input" refers to the situation when the value of variable might be influenced by users’ input, such as file contents and "argv". These 125 weaknesses consist of 70 "result of undetermined calculation", 51 "signed definition" and 4 "influenced by input". Details of these weaknesses can be accessed online.

Fig. 7 shows an example of "signed definition" weakness detected in actual projects. It is a code snippet in file "crlf.c" of the project "Cygwin-master". The assertion in Line 11 may fail because buf->max is defined as an integer, which could be less than zero.

5 CONCLUSION

In this paper, we present Weak-Assert, a weakness-oriented assertion recommendation toolkit for program analysis. Weak-Assert can help to generate weakness-oriented assertions automatically. It uses well-designed patterns to match the abstract syntax trees of source code automatically, collects significant messages from trees and inserts assertions into proper locations of programs. These assertions can be checked by using program analysis techniques, such as program testing and verification. The experiments are set up on Juliet test suite and several actual projects in Github. Experimental results show that Weak-Assert helps to find 125 program weaknesses in 26 actual projects. These weaknesses are confirmed manually to be triggered by some test cases.

REFERENCES


