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

Cong Wang[1], Yu Jiang[1]*, Xibin Zhao[1], Xiaoyu Song[2], Ming Gu[1], Jiaguang Sun[1]
[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-Assert* [1], a weakness-oriented assertion recommendation toolkit for program analysis of C code. A weakness-oriented assertion is an assertion which can help to find potential program weaknesses. Weak-Assert uses well-designed patterns to match the abstract syntax trees of source code automatically. It collects significant messages from trees and inserts assertions into proper locations of programs. These assertions can be checked by using program analysis techniques. 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.

The address of the abstract demo video is:
https://youtu.be/_RWC4GJvRWc

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**; **Formal software verification**;

KEYWORDS

assertion recommendation, program weakness, formal program verification, program testing

1 INTRODUCTION

Assertions are helpful in program analysis. Program testing [1] and verification [2] requires assertions to check specific properties of programs. However, it is reported in [10] that the assertion statements are only 0.03% among surveyed source code. Meanwhile, it is time-consuming and error-prone to write assertions manually.

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          int data = atoi(argv[1]);
6          char source[100];
7          char dest[100]="";
8          memset(source,'A',100-1);
9          source[100-1] = '\0';
10         if (data<100){
11                 assert(data>=0);
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)

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

[2]Common Weakness Enumeration (CWE) is a community-developed list of common software security weaknesses. http://cwe.mitre.org/index.html

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5663-3/18/05.
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

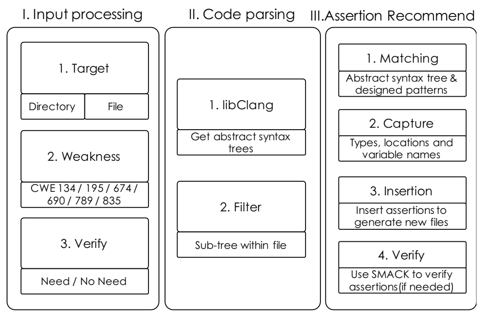

I. Input processing
II. Code parsing
III.Assertion Recommend
1. Target
Directory
File
2. Weakness
CWE 134 / 195 / 674 / 690 / 789 / 835
3. Verify
Need / No Need
1. libClang
Get abstract syntax trees
2. Filter
Sub-tree within file
1. Matching
Abstract syntax tree & designed patterns
2. Capture
Types, locations and variable names
3. Insertion
Insert assertions to generate new files
4. Verify
Use SMACK to verify assertions(if needed)

Figure 2: Framework of Weak-Assert

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 support 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

ID	Weakness Name
134	Use of Externally-Controlled Format String
195	Signed to Unsigned Conversion Error
674	Uncontrolled Recursion
690	Unchecked Return Value to NULL Pointer Dereference
789	Uncontrolled Memory Allocation
835	Loop with Unreachable Exit Condition ('Infinite Loop')

The weakness types are shown in Table. 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.

```
#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.

```
1  void helperBad(){
2  /* FLAW: this function causes infinite recursion */
3      helperBad();
4  }
```

Figure 4: Example for Weakness ID.674

```
1  int iterator_tempvalue = 0;
2  static void helperBad(){
3      iterator_tempvalue+=1;
4      assert(iterator_tempvalue<=100000);
5  /* FLAW: this function causes infinite recursion */
6      helperBad();
7  }
```

Figure 5: Modified 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).

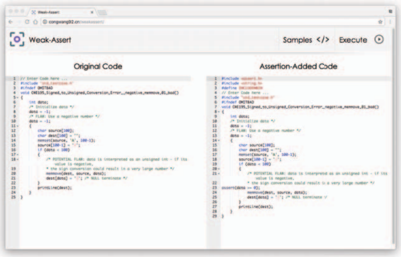
3.2 Usage

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

Parameter	Description
-d <directory>	<directory> is the entrance of C language projects.
-f <file>	<file> is a single source file.
-w <weakness ID>	<weakness ID> is the list of weakness types, separated with comma.
-v	if you want to verify the assertions, use this parameter.

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.



```
// Enter Code here ...
#include "std_testcase.h"
#ifndef OMITBAD
void CWE195_Signed_to_Unsigned_Conversion_Error__negative_memmove_01_bad()
{
    int data;
    /* Initialize data */
    data = -1;
    /* FLAW: Use a negative number */
    data = -1;
    {
        char source[100];
        char dest[100] = "";
        memset(source, 'A', 100-1);
        source[100-1] = '\0';
        if (data < 100)
        {
            /* POTENTIAL FLAW: data is interpreted as an unsigned int - if its
               value is negative,
             * the sign conversion could result in a very large number */
            memmove(dest, source, data);
            dest[data] = '\0'; /* NULL terminate */
        }
        printLine(dest);
    }
}
```

```
#include <assert.h>
#include <string.h>
#define [illegible]
// Enter Code here ...
#include "std_testcase.h"
#ifndef OMITBAD
void CWE195_Signed_to_Unsigned_Conversion_Error_negative_memmove_01_bad()
{
    int data;
    /* Initialize data */
    data = -1;
    /* FLAW: Use a negative number */
    data = -1;
    {
        char source[100];
        char dest[100] = "";
        memset(source, 'A', 100-1);
        source[100-1] = '\0';
        if (data < 100)
        {
            /* POTENTIAL FLAW: data is interpretd as an unsigned int - if its
               value is negative,
             * the sign conversion could result in a very large number */
assert(data >= 0);
            memmove(dest, source, data);
            dest[data] = '\0'; /* NULL terminate */
        }
        printLine(dest);
    }
}
```

Figure 6: Interface of Weak-Assert Online

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.

Table 3: Actual Projects in Github

Project Name	Disk(MB)	C Files	Weak(ID.195)
1.linux-master	785	25568	33
2.php-src-master	76	1022	1
3.FFmpeg-master	54	2431	27
4.Cygwin-master	53	3374	10
5.emscripten-master	47	2412	9
6.Telegram-master	21	521	4
7.Arduino-master	19	189	1
8.obs-studio-master	12	668	2
9.vim-master	12	135	2
10.h2o-master	11	356	3
11.git-master	8.7	403	4
12.mpv-master	6.5	326	6
13.libuv-master	3.1	267	1
14.ijkplayer-master	3	92	2
15.netdata-master	3	106	1
16.toxcore-master	2	72	1
17.masscan-master	1.7	84	2
18.tmux-master	1.7	162	2
19.torch7-master	1.1	50	2
20.JSPatch-master	0.872	4	1
21.jq-master	0.868	20	1
22.memcached-master	0.768	25	4
23.twemproxy-master	0.764	34	1
24.The-Art-Of-Programm	0.48	36	3
25.the_silver_searcher	0.316	12	1
26.wrk-master	0.284	14	1
Data in total	1125.152	38383	125

```
1  typedef struct {
2          unsigned char *bytes;
3          int max, count;
4  } Buffer;
5  void expand_buf(Buffer *buf, int len){
6          if (buf->max<len){
7          buf->max=len+20;
8          if (buf->bytes)
9                  buf->bytes=(unsigned char *)realloc(
                           buf->bytes, buf->max);
10         else{
11                 assert(buf->max>=0);
12                 buf->bytes=(unsigned char *)malloc(buf
                           ->max);
13         }
14 }
```

Figure 7: Example of Weakness Detected in Actual Projects

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 [4].

[3]http://congwang92.cn/weakassert/

[4]http://congwang92.cn/wa/result.html

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

[1] Roy Budhai, Brian Chen, Teresa Su, and Sheldon Sequeira. 2016. Testing application code changes using a state assertion framework. (2016).

[2] Montgomery Carter, Shaobo He, Jonathan Whitaker, and Michael Emmi. 2017. SMACK software verification toolchain. In *Ieee/acm International Conference on Software Engineering Companion*. 589–592.

[3] Yao Hua Dong and Ji Dong Peng. 2010. The Realization of Page Load-stress Testing with LoadRunner. *Journal of Jiangxi University of Science & Technology* (2010).

[4] Github. [n. d.]. Github. https://github.com/. ([n. d.]).

[5] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2003. Software verification with BLAST. In *Model Checking Software*. Springer, 235–239.

[6] Matthew Hicks, Cynthia Sturton, Samuel T King, and Jonathan M Smith. 2015. Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs. *ACM SIGPLAN Notices* 50, 4 (2015), 517–529.

[7] Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Computing Surveys (CSUR)* 41, 4 (2009), 21.

[8] Yu Jiang, Hehua Zhang, Han Liu, William Hung, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2014. System reliability calculation based on the run-time analysis of ladder program. *IEEE Transactions on Industrial Electronics* (2014).

[9] Nick Langley. 2003. Winrunner automates app testing. *Computer Weekly* (2003).

[10] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre F Baldi. 2008. Mining internet-scale software repositories. In *Advances in neural information processing systems*. 929–936.

[11] H. Pham Long, Ly Ly Tran Thi, and Jun Sun. 2017. Assertion Generation through Active Learning. In *Ieee/acm International Conference on Software Engineering Companion*. 155–157.

[12] Robert A Martin. 2007. Common weakness enumeration. *Mitre Corporation* (2007).

[13] NIST. [n. d.]. Software Assurance Reference Dataset. https://samate.nist.gov/SRD/testsuite.php. ([n. d.]).

[14] S. Schaub and B.A. Malloy. 2014. Comprehensive analysis of C++ applications using the libClang API. (2014).

[15] Shobha Vasudevan, David Sheridan, Sanjay Patel, and David Tcheng. 2010. GoldMine: Automatic assertion generation using data mining and static analysis. 46, 2 (2010), 626–629.

[16] Cong Wang, Fei He, Xiaoyu Song, Yu Jiang, Ming Gu, and Jiaguang Sun. 2017. Assertion Recommendation for Formal Program Verification. In *Computer Software and Applications Conference*. 154–159.

[17] Hehua Zhang, Yu Jiang, William NN Hung, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2014. Symbolic analysis of programmable logic controllers. *IEEE Trans. Comput.* 63, 10 (2014), 2563–2575.

[18] Rui Zhang, Natalie Stanley, Christopher Griggs, Andrew Chi, and Cynthia Sturton. 2017. Identifying Security Critical Properties for the Dynamic Verification of a Processor. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 541–554.