USE OF REPAIR TOOLS FOR FIXING SECURITY VULNERABILITIES

EDWIN LESMANA TJIONG

(B.Eng. (Hons), NTU)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2018

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Edwin Lesmana Tjiong

April 6, 2018

Acknowledgement

I would like to express my gratitude to my advisor, Prof. Abhik Roychoudhury, for his continuous guidance from the beginning to completion of my thesis. I am also thankful for my fellow teammates who have helped me along the way. Among them are: Sergey Mechtaev, who have helped me by configuring his repair tool for my experiment. Andrew Santosa, who guided me by reviewing and discussing related works. Gao Xiang, who helped me for data collection.

Table of Content

List of Figures

List of Tables

Name: Edwin Lesmana Tjiong

Degree: Master of Computing (Infocomm Security)

Supervisor(s): Prof. Abhik Roychoudhury

Department: Department of Computer Science

Thesis Title: Use of Repair Tools for Fixing Security Vulnerabilities

Abstract

Automated patch generation approaches have been shown to be able to address defects in real-world programs, including security vulnerabilities. Some of the automatic repair tools are designed to repair general software bugs, while others are used for security-related bugs such as integer overflow or buffer overrun. To the best of our knowledge, there exists no work that tried to assess capability and patch quality of general purpose tools against security-related bugs.

Fuzzing is a well-known technique that is effective in discovering security-related bugs. Recently, the OSS-Fuzz framework has been introduced, which combines modern fuzzing techniques using tools such as AFL with scalable distributed execution to provide reporting of the bugs found by the fuzzers continuously being executed by the framework. OSS-Fuzz's bug tracking system thus becomes a suitable source for security-related bug data, for the purpose of analysing capabilities of general purpose automated repair tools.

In this article, we present the results of a case study on automated repair of bugs reported by the OSS-Fuzz framework. Our data set consists of 240 OSS-Fuzz bug reports and their manual fixes. We have done two experiments. First, we have categorized bugs by its error type and

analysed its manual fix pattern for most common bugs that will be insightful for program repair developers. We also run general -purpose repair tool (F1X) to repair these bugs and analysed capabilities of this tool in repairing security bugs. The result showed it is necessary for general-purpose tool to have additional information by sanitizer to fix security-specific bugs and a test suite does not helpful in strengthening patch quality. Instead of test suite, we propose a comparison with successful manual fix patterns to strengthen patch quality of repair tools.

1. Motivation

The complexity of computer software is continuously increasing with tremendous speed. It is common for large-scale software to have code base of millions line-of-code. The maintenance of large-scale software requires a significant effort even for expert programmers. Especially for security-related bugs, it is desirable to quickly discover and fix bugs before it can be exploited by malicious parties. Therefore, two technologies have been devised to assist human and automate maintenance process: Fuzzing and automated program repair.

Fuzzing is a testing technique by using program that contains initial input file to be fed into program under test. Program under test is executed and at the end of each execution, the input file is mutated and then the program under test is re-executed with new input file. This process is repeated until bugs are found or the fuzzing program is terminated by user. Fuzzing technique is used to automate bug discovery process. By using this technology, many security bugs have been found successfully.

However, overwhelming numbers of bugs found by this method may not be balanced by quick response of programmers which are assigned to fix these bugs. Therefore, the next logical step to handle these bugs are automated repair, which is used to automate bug fixing process. A good automated repair tool can provide a temporary fix for the system while the programmer analyse and find the root cause of the security bug before applying more proper fix accordingly. Afterward, a series of these successful human fixes can be used to assess and improve quality of fixes generated by automated repair tools.

1.1. Fuzzing

Fuzzing is an automated testing technique that is effective in discovering security-related bugs, such as integer overflow [1] [2] [3] and buffer overflow [4]. Compared to normal testing, it supplies unusual inputs to the software under test (SUT) to stress-test it. More advanced fuzzing approaches support automated mutation of the original inputs to explore more execution paths of the code. Most recent developments in the fuzzing arena resulted in greybox fuzzing approach [5] [6], where lightweight instrumentation of the SUT is used to guide the mutation towards less-explored code region. The greybox fuzzing approach has been very successful in the discovery of software bugs [5], which is increasing in effectiveness due to research progress in the area [7]. Recently, the OSS-Fuzz [8] framework has been introduced, which combines modern fuzzing techniques including greybox fuzzing approaches of AFL [5], libFuzzer [6], and AFLGO [9], with scalable distributed execution to provide reporting of the bugs found by the fuzzers continuously executed by the framework. Internally at Google, OSS-Fuzz has been successfully deployed to fuzz Chrome components and discovered hundreds of security vulnerabilities and stability bugs [8]. Together with the bug discovery report, OSS-Fuzz provides a failing test case to reproduce the bugs as well as links to the bug fixes in versioning software (Git, SVN, Mercurial, etc.) in its bug tracking system. The bug tracking system thus becomes a suitable source for security-related bug data, for the purpose of analysing capability of automated program repair tools.

1.2. Automated Repair

Automated Program Repair (APR) is a technique of fixing target program by using another program as repair tool to automate bug patching process. The process of automated repair is basically consisted of four major parts: bug identification, patch generation, patch validation and patch prioritization. The repair tools are differed and classified based on techniques they use in each of these parts. We will give an overview of the prevalent techniques for each of these parts.

1.2.1. Bug Identification

For program repair tools, the first obvious step to fix a bug is to have an ability to identify that the target program indeed contains a bug. To do so, there is a need to have concrete expectation about program behaviour when certain conditions are satisfied. We called this thing as *specification* – a set of expected behaviours. By using specification, repair tools are able to identify characteristics of the bug and subsequently generate appropriate fixes. According to work of Monperrus [10], there are three types of specification used for bug identification by program repair tools:

1) Test oracles

Test oracle is a specification of the result of correct program execution. The oracle is a part of specification that only related to expected output. There are two oracles which are most commonly used by repair tools.

One of the popular kind of oracle is ***test suite***. A test suite is a set of pair of expected input and expected output based specification. One of a classical repair tool that used test-suite is a GenProg [11]. Since then, a lot of repair tools have been developed using the same bug identification method as GenProg, such as Semfix [12], PAR [13], and SPR [14].

Another type of oracle is ***formal specification***, like a use of state machine for modelling variables during program execution. By formal specification, we can specify things such as allowed object states or function calls. For example, in [15], the author designed a patch technique that requires an input of a safety policy (i.e. a typestate property or an API usage rule) and a control-flow graph of a method.

2) Static analysis

Static analysis technique detects potential bugs during compilation process *before* target program is executed. Upon bug identification, static analysis suggest appropriate fixes accordingly. The technique targets certain patterns inside target program that are associated by certain error types. For examples, IntPTI [16] is used to identify integer overflow, BovInspector [17] is used to identify buffer overflow, and LeakFix [18] to identify memory leak.

3) Crashing inputs

A common implicit expectation for users of any program is that the program will not crash for any given input. If a certain given input is able to crash the program then it is called *crashing input*. Once crashing input is identified, repair tool is able to start their bug fixing process by identifying error lines from execution traces and generate fixes accordingly.

The main difference between crashing inputs and test suite oracle is that a test suite contains both passing test cases and failing test cases with assertions of expected values. In contrast, crashing input only refers to a violation of implicit expectation "The program should not crash". For example: Gao et al [19] repairs crashing exceptions based on Stack Overflow. Their system, QACrashFix, mines pairs of buggy and fixed code on Stackoverflow, in order to extracts an edit script. The edit scripts are tried in sequence in order to suppress crashing exception.

1.2.2. Patch Generation

Based on its patch generation technique, program repair tools can be classified into heuristic based repair and semantics based repair.

1) Heuristic-based Repair

Heuristic based repair are based on theoretical analysis that are codified inside repair tools before the patches are validated during runtime execution. For examples, GenProg [11] heuristically searches for repairs by genetic algorithm technique, PAR [13] generates repairs based on templates that are extracted by studying human written patches, and Prophet [20] used machine learning model to generate patches.

2) Semantic-based Repair

Semantic-based techniques use information obtained during runtime execution and program synthesis to generate patches. Semantic repair, however, cannot include the concept of patch prioritization or only include simple patch ranking criteria such as syntactic structural difference [21]. The techniques include symbolic execution and constraint solving, including tools such as Semfix [12], SPR [14], Qlose [22] and Angelix [23].

1.2.3. Patch Validation

After automated repair tools generate multiple patches, there will be a need to check correctness and validity of these patches. For such purpose, most programs will have a test suite. Note that previously test suite is used as bug identification purpose. However, here, it also can be used as patch validation. In each test suite, multiple pair of expected inputs with expected outputs are given. Automated repair tools will apply generated patches into program under repair and run test suite afterward to see if patched program will pass all test cases inside the test suite. Only patches that pass test suite will be selected as candidates for program repair.

1.2.4. Patch Prioritization

Most of these patches are not really correct just because they passed the test suite. They can give incorrect results for the other input outside the test suite. This phenomenon is called *overfitting*. To resolve this issue, there are many techniques used to assess quality of generated patches so that we can do patch prioritization. By that, patches can be sorted accordingly and only the best patches can be applied to the target program. The followings are main techniques that are used for patch prioritization in most repair tools:

1) Syntactic distance

Syntactic distance is a measurement of difference between original program statements with proposed patch solution at the Abstract Syntax Tree (AST) [24] level. Here, one component of the distance is the number of minimum transformation applied to AST of original program, such as insert, delete, update or move, to arrive at AST of proposed patch solution. Several other components are also considered as the number and position of variables and constants. Some examples of the repair tools using syntactic distances are F1X [25] and S3 [21].

2) Semantic distance

In contrast of syntactic distance which measures distance at the static AST level, semantic distance measures how similar are the behaviours during runtime execution between original program and program after applying generated patch. An example of such repair tool is Qlose [22], it defined semantic distance as the difference of between execution trace of original program and patched program for running each test case inside test suite.

3) Similarity with human patches

The latest approach on patch prioritization is by assigning a rank to each patch candidate based on certain heuristic score on how similar this patch candidate to previous successful human patches obtained from versioning software. One of the tool that implemented this approach is Prophet [20]. Prophet identified some key characteristics (called *features*) derived from past successful patches and it will assign a probabilistic score to the patch candidate based on the number of features from training set of successful patches which present on current patch candidate. Hence, more similar patch candidate (with more identical features with past patches) will be assigned higher probabilistic score and will be prioritized during validation by test suite of target program. Overall, it will have higher chance to be selected as final patch choice.

1.3. Repair Scope

Automated Program Repair (APR) tools are developed to address bugs in general and not aimed towards bugs with certain patterns. On the contrary, the software bugs in security context are often classified based on their nature. For example, integer overflow is a bug type in which result of integer arithmetic operation exceeds capacity of assigned data type. The repair techniques which aimed to fix specific security bugs are widely available and well researched. However, research to develop the repair tools which aim to fix all type of bugs in general is just recently started. Hence in our study, we want to examine the effectiveness of general repair tools to fix security related bugs.

However, note also that general repair tools only can fix bugs for which behaviour for correct program are well-defined unambiguously. For example, it cannot fix system performance (slowness) or concurrency-related (incorrect value of shared variable) bugs. Additionally, general repair tools only work if we give them access to source code of target program. Hence, they are unable to fix the program in binary for which the source code is not present.

1.4. Experiment

In this article, we present the results of a case study on automated repair of bugs reported by the OSS-Fuzz framework. Our data set consists of 240 OSS-Fuzz bug reports and their fixes collected from five SUTs: FFMPEG, Wireshark, PROJ.4, OpenJPEG, and Libarchive. From experiment, we categorized and analysed security vulnerabilities based on their types and fix patterns. We also run a general-purpose repair tool called F1X to analyse its capabilities and limitations to fix security-related bugs.

The rest of this thesis is organized by the following outline:

- In Section 2 (Overview), we provide a brief overview on two technologies we used in our experiment: a repair tool called F1X, continuous fuzzing platform called OSS-Fuzz, and security technology named sanitizer.
- In Section 3 (Experiment Setup), we provide an overview on the methodology we use to collect dataset based on OSS-Fuzz for our experiment.
- In Section 4 (Experiment Dataset), we present a statistical result of our dataset collection and findings of the manual fix patterns.
- In Section 5 (Experiment Result), we present a result of running F1X on the dataset presented in Section 4, derive and analyse findings from that experiment.
- In Section 6 (Related Works), we provide a list of related works on automated repairs and how our work differ from those works.
- In Section 7 (Future Works), we elaborate on possible future works based on our findings in experiment result.
- In Section 8 (Conclusion), we summarize our entire case study.

2. Overview

In this section, we will provide a brief overview on the key tools we use for our experiment: general repair tool named F1X, automated fuzzing platform named OSS-Fuzz, and technology used for detecting security bugs named sanitizer.

2.1. F1X

F1X [25] is a generate-and-validate repair tool which utilizes a dynamic instrumentation to identify buggy statements. It uses six types of transformation schemas (modifying expression, changing conditions, adding if-guard, adding assignment statement, memory initialization and editing function call).

A unique ability of F1X is its optimization of patch generation and validation by dividing its patches into test-equivalence classes. Basically, it determines during program execution if currently generated patch is equivalent to previously generated patches and skip the execution otherwise. This ability makes F1X execution faster compared to latest novel repair tools (Angelix [23] and Prophet [20]).

For patch prioritization, F1X uses a combination of both syntactic and semantic distance. Syntactic distance is defined as a number of added, modified and deleted AST nodes whereas semantic distance measure similarity of execution trace between positive test cases and patched program.

We choose F1X as a representation of general-purpose repair tool in our experiment to generate fixes for security bugs because of its speed. In our experiment, we want to examine if general-purpose tool is able to generate fix for security bugs. In this experiment design, we do not assume any particular technique used to bug identification before fix generation. By this way, our finding from this case study can be applied to any template-based repair tools other than F1X.

2.2. OSS-Fuzz

OSS-Fuzz [8] is continuous fuzzing platform hosted in Google's cloud infrastructure. This fuzzing platform is used to test many open-source projects to find security vulnerabilities. It compiled those open source projects with many incorporated fuzzers and when it manages to find a bug, OSS-Fuzz will add it into an issue tracker repository. The maintainer of corresponding open-source projects will receive a notification about the bug and he will be given a specific amount of times to respond to the bug report. In the bug report, there will be a failing test case provided. By using this failing test case with specified fuzzer, the bug can be reproduced.

After the programmer has fixed the issue, he will release a fix into online versioning software (Git, mostly) and will specify bug id corresponded to the report in OSS-Fuzz issue tracker. Hence, for each reported bug we are able to get a failing test case and its corresponding bug fix (if any). By mining these bug fixes in OSS-Fuzz, we are able to analyse fix patterns for security-related bugs which can also be used for improving patch quality generated by automated repair tools. A failing test case for each bug can also be used as an input for any generate-and-validate repair tools to test their repair capability.

More detail workflow of OSS-Fuzz can be seen from the figure below:

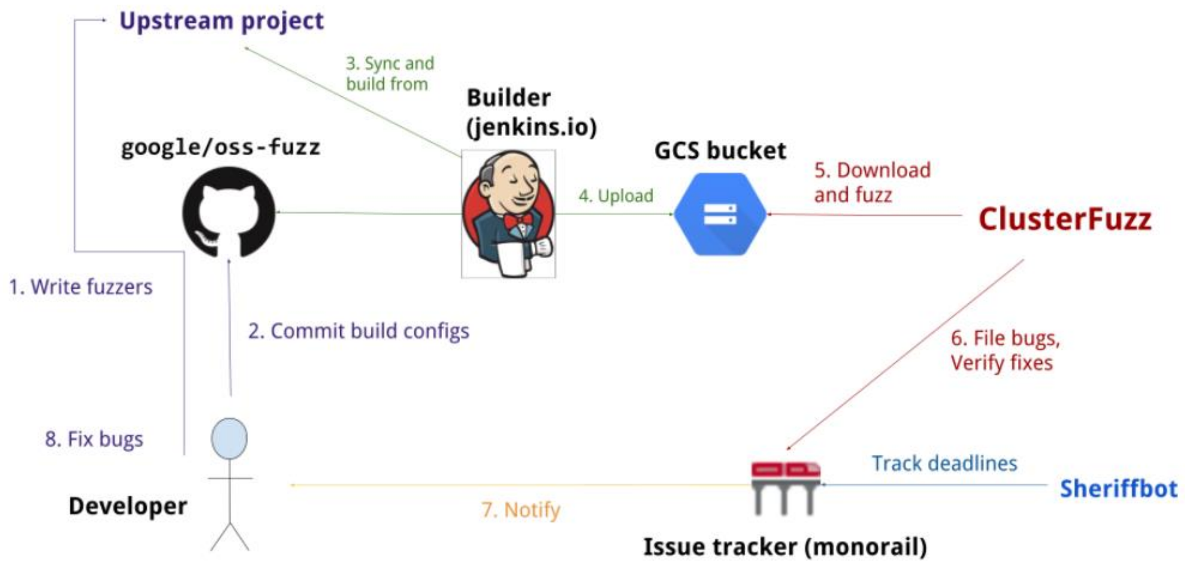


Figure 1: OSS-Fuzz Detail Workflow

2.3. Sanitizer

For detection of security bugs, OSS-Fuzz uses security technology called sanitizer. Many security bugs happen in unsafe languages, such as C, and the bugs do not halt program execution or generate exception when they occur. The work of a sanitizer is based on compiler instrumentation. When the program under test is compiled, the compiler will specify an option to activate sanitizer. Sanitizer will then put a check for the certain specified operations inside source code and trigger error message when that error happen during runtime execution.

OSS-Fuzz is using three types of sanitizer, as the following:

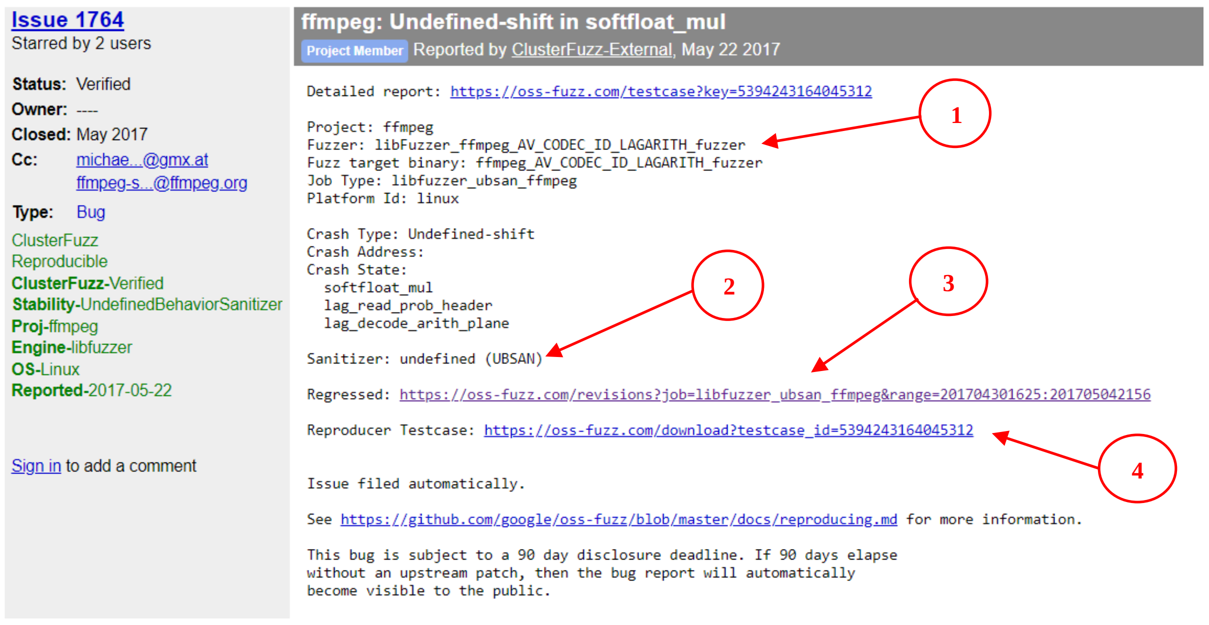
1) **Address Sanitizer (asan)**: is a fast memory error detector. This tool can detect the following types of security bugs: Out-of-bound access (buffer overflow), use-after-free, use-after-return, use-after-scope, double-free, invalid free, and memory leak.
2) **Memory Sanitizer (msan)**: is a detector of uninitialized reads, such as the use of uninitialized variables or pointers.

3) **Undefined Sanitizer (ubsan):** is a fast undefined behaviour detector. This tool can detect the following bugs: using misaligned or null pointer, signed integer overflow, conversion to, from, or between floating-point types which would overflow the destination.

3. Experiment Setup

To collect data for our experiment, we looked first into OSS-Fuzz bug tracker list for our target subjects (FFMPEG, Wireshark, PROJ.4, OpenJPEG and Libarchive). On the bug tracker list, OSS-Fuzz list down necessary configurations needed to reproduce each bug such as:

1) Appropriate fuzzer engine to use (shown as number 1). Each fuzzer engine can only be used to reproduce a specific type of bugs.
2) Appropriate sanitizer to use (shown as number 2). There are three sanitizers: address (asan), memory (msan), and undefined (ubsan). Mostly, the type of sanitizer is specified explicitly. Each of them will produce specific error message for certain bug types.
3) A link to versioning software site from which we can find a fix for the specific bug (shown as number 3). The commit will contain message like ’Found by OSS-Fuzz’
4) A failing test case (shown as number 4). It will be used for repair tools which work based on generate-and-validate (F1X).



Issue 1764
Starred by 2 users

Status: Verified
Owner: ----
Closed: May 2017
Cc: michae...@gmx.at
ffmpeg-s...@ffmpeg.org
Type: Bug
ClusterFuzz
Reproducible
ClusterFuzz-Verified
Stability-UndefinedBehaviorSanitizer
Proj-ffmpeg
Engine-libfuzzer
OS-Linux
Reported-2017-05-22

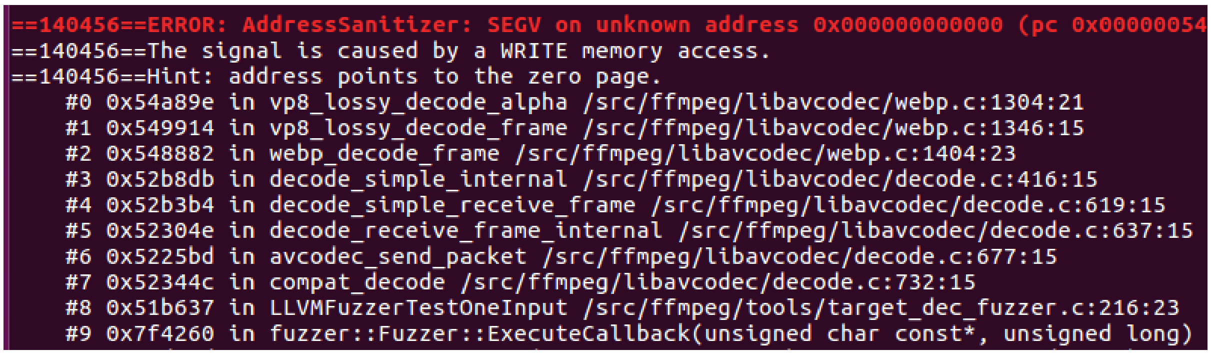
Sign in to add a comment

ffmpeg: Undefined-shift in softfloat_mul
Project Member Reported by ClusterFuzz-External, May 22 2017

```
Detailed report: https://oss-fuzz.com/testcase?key=5394243164045312

Project: ffmpeg
Fuzzer: libFuzzer_ffmpeg_AV_CODEC_ID_LAGARITH_fuzzer
Fuzz target binary: ffmpeg_AV_CODEC_ID_LAGARITH_fuzzer
Job Type: libfuzzer_ubsan_ffmpeg
Platform Id: linux

Crash Type: Undefined-shift
Crash Address:
Crash State:
  softfloat_mul
  lag_read_prob_header
  lag_decode_arith_plane

Sanitizer: undefined (UBSAN)

Regressed: https://oss-fuzz.com/revisions?job=libfuzzer_ubsan_ffmpeg&range=201704301625:201705042156

Reproducer Testcase: https://oss-fuzz.com/download?testcase_id=5394243164045312


Issue filed automatically.

See https://github.com/google/oss-fuzz/blob/master/docs/reproducing.md for more information.

This bug is subject to a 90 day disclosure deadline. If 90 days elapse
without an upstream patch, then the bug report will automatically
become visible to the public.
```

Figure 2: OSS-Fuzz Bug Report Page

By using item number 1 and number 4, we are able to construct ***driver test file*** for each bug. Driver test file is the file which contain execution command that trigger execution of target subject, compiled with specified fuzzer (item 1), with the parameter of failing test case (item 4). Upon execution, the specified bug will appear immediately, resulting in stack trace as shown below:

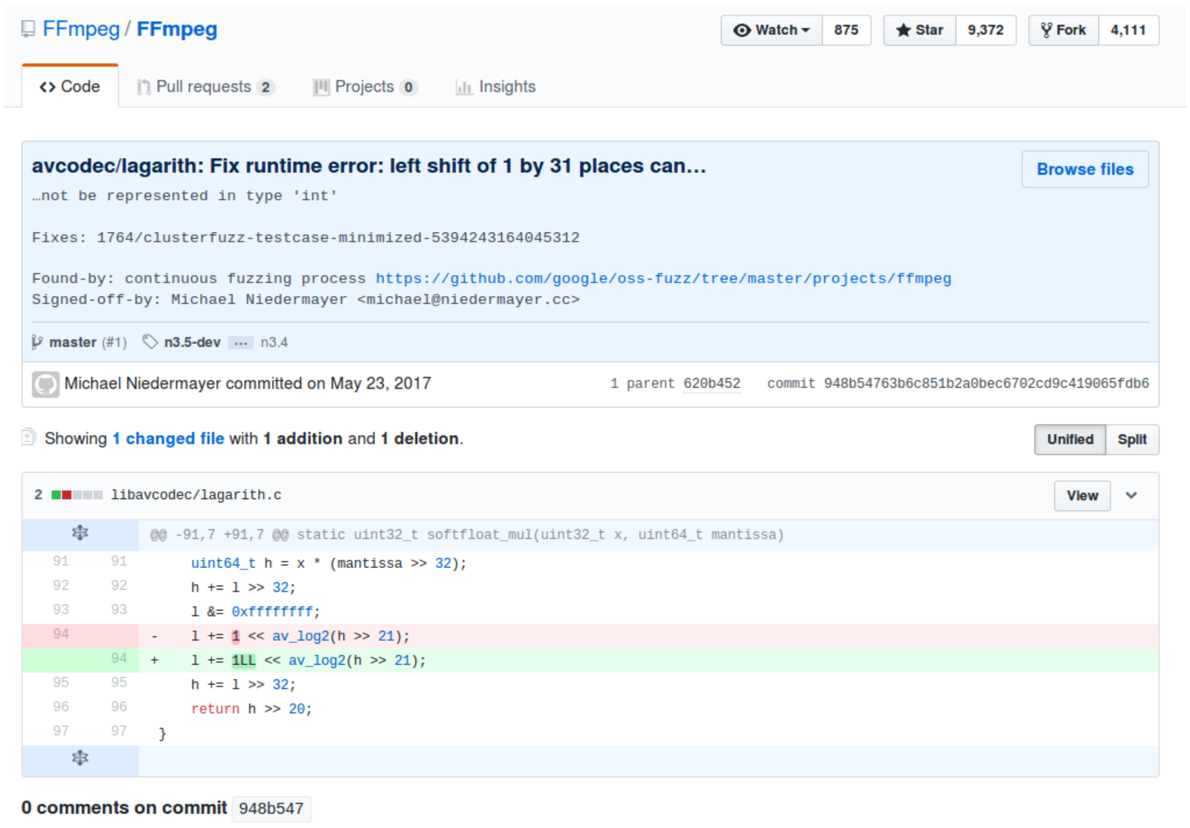


```
==140456==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x00000054
==140456==The signal is caused by a WRITE memory access.
==140456==Hint: address points to the zero page.
    #0 0x54a89e in vp8_lossy_decode_alpha /src/ffmpeg/libavcodec/webp.c:1304:21
    #1 0x549914 in vp8_lossy_decode_frame /src/ffmpeg/libavcodec/webp.c:1346:15
    #2 0x548882 in webp_decode_frame /src/ffmpeg/libavcodec/webp.c:1404:23
    #3 0x52b8db in decode_simple_internal /src/ffmpeg/libavcodec/decode.c:416:15
    #4 0x52b3b4 in decode_simple_receive_frame /src/ffmpeg/libavcodec/decode.c:619:15
    #5 0x52304e in decode_receive_frame_internal /src/ffmpeg/libavcodec/decode.c:637:15
    #6 0x5225bd in avcodec_send_packet /src/ffmpeg/libavcodec/decode.c:677:15
    #7 0x52344c in compat_decode /src/ffmpeg/libavcodec/decode.c:732:15
    #8 0x51b637 in LLVMFuzzerTestOneInput /src/ffmpeg/tools/target_dec_fuzzer.c:216:23
    #9 0x7f4260 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long)
```

Figure 3: Stack trace obtained from feeding failing test case to FFMPEG program

Item number 2 (Type of sanitizer) is used during installation of OSS-Fuzz target subjects. For each target subject, OSS-Fuzz provided us with Dockerfile to install them inside Docker container. The helper script is provided to specify type of sanitizer we want to use to compile target subject with. Only by specifying correct sanitizer type, the above stack trace for each bug can be correctly reproduced. We developed another web crawler to extract sanitizer type from OSS-Fuzz bug tracker to construct ***docker command file*** to build docker container for each bug.

Lastly, we create another web crawler to check link of item 3 (containing range of commits in their Github account) if any commit within specified has a link to OSS-Fuzz bug tracker. If they have the link, then that commit is considered a fix for the bug referred by OSS-Fuzz bug tracker. Typical example of commit for such bug fix is shown below:



avcodec/lagarith: Fix runtime error: left shift of 1 by 31 places can…

```
…not be represented in type 'int'

Fixes: 1764/clusterfuzz-testcase-minimized-5394243164045312

Found-by: continuous fuzzing process https://github.com/google/oss-fuzz/tree/master/projects/ffmpeg
Signed-off-by: Michael Niedermayer <michael@niedermayer.cc>
```

Michael Niedermayer committed on May 23, 2017 — 1 parent 620b452 — commit 948b54763b6c851b2a0bec6702cd9c419065fdb6

Showing 1 changed file with 1 addition and 1 deletion.

2 libavcodec/lagarith.c

```
@@ -91,7 +91,7 @@ static uint32_t softfloat_mul(uint32_t x, uint64_t mantissa)
      uint64_t h = x * (mantissa >> 32);
      h += l >> 32;
      l &= 0xffffffff;
-     l += 1 << av_log2(h >> 21);
+     l += 1LL << av_log2(h >> 21);
      h += l >> 32;
      return h >> 20;
  }
```

0 comments on commit 948b547

Figure 4: An example of bug fix in Github linked to OSS-Fuzz bug report

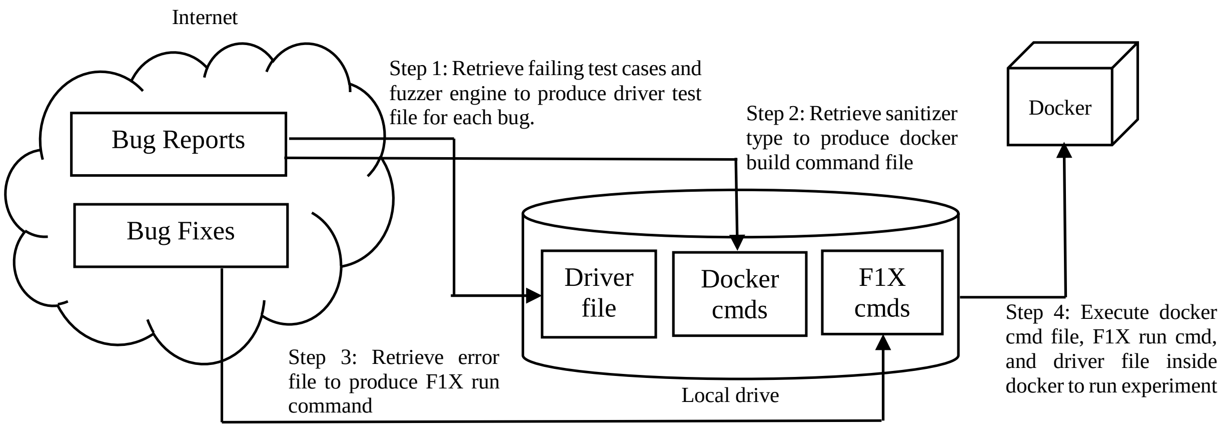
From commit page that contains bug fix, we are able to analyse their fix patterns, both for one-line fixes and multiple-lines fixes, for each category of security bugs, such as integer overflow, buffer overflow, and so on. The analysis of fix patterns will be useful reference for repair tools' developers who intend to optimize patch quality of their tools. Bug fix page also contains source file(s) and also error line(s). This information will be used for construct ***F1X run command file*** as it needs to specify source file to be fixed as one of its argument.

After constructing driver test file, docker command file, and F1X run command file, we install buggy version of target project inside docker container and use those files to run F1X to see if it is able to fix bugs in our dataset. After finish running, F1X will give statistical figures as following:

- Number of patch candidate evaluated

- Number of executed test cases
- Execution speeds (no. of execution per seconds)
- Number of plausible patch locations
- Number of generated patches.

The entire workflow process of our experiment benchmark can be seen in a diagram below:



Internet
Bug Reports
Bug Fixes
Step 1: Retrieve failing test cases and fuzzer engine to produce driver test file for each bug.
Step 2: Retrieve sanitizer type to produce docker build command file
Docker
Driver file
Docker cmds
F1X cmds
Local drive
Step 3: Retrieve error file to produce F1X run command
Step 4: Execute docker cmd file, F1X run cmd, and driver file inside docker to run experiment

Figure 5: A workflow on our repair experiment using OSS-Fuzz and F1X

Our contribution for this framework setup is given as followings:

1) We developed three web crawlers to obtain failing test case, sanitizer type, and error file name to produce text files that listed all those things for all reproducible bugs in our target subjects.
2) We configured compilation and helper scripts in OSS-Fuzz which initialize Docker environment and install target subjects inside Docker container. Many compilation scripts need to be configured to compile past versions of target subjects which contain bugs.
3) We created scripts which created driver files, docker command files and F1X run command files automatically for all reproducible bugs in all target subjects.

4. Experiment Dataset

In this section, we will elaborate and analyse statistical findings on security vulnerabilities we obtained from OSS-Fuzz together with their manual fix patterns.

4.1. Selection Process

The selections of these target subjects are made based on the number of bugs found by OSS-Fuzz and their reproducibility. Initially, we aimed at the projects which contain largest number of bugs and then we tried to install those projects in our Docker environment. However, we also found that a lot of security bugs are not reproducible even after executing it with correct code version (before patching) and failing test case obtained from OSS-Fuzz bug trackers. If majority of bugs are not reproducible, we skipped such target subjects.

Another factor in consideration is type of programming language. F1X is only able to fix C projects and hence we only took target subjects which majority are written in C. For some projects, we also encountered difficulties during their setup because their build script are modified multiple times across timeline of the bugs. On each build script, it includes different library dependencies and different version of same libraries. The latest build script might not be able to compile past version code of target subject and vice versa. We tried to troubleshoot this issue initially but if we found that it is too time consuming, we moved to another target subjects.

Lastly, there is a compatibility issue related to F1X. F1X used LLVM and Clang compiler version 3.8.1. Some target subject generated error during code instrumentation with F1X because they did not recognize some Clang compiler options. Hence, even if the bugs are reproducible, we skipped them because they cannot be compiled with F1X and hence F1X is unable to generate the fixes.

4.2. Threat to Validity

We realized that with constraints to find good target subjects we encountered in our experiment, there will be inherent limitation of our study. Most obvious limitation is related to bug type we encountered in our dataset. We can only analyse fix patterns from human programmers and F1X for bug types that are available in sufficient number. In our case, we have integer overflow and buffer overflow as dominant bug type in our dataset. Hence, we will only discuss their fix pattern. We will not discuss other bug types since we may draw wrong conclusion from only few available bugs.

Nevertheless, we acknowledged that with another dataset we may obtain different dominant bug types. However, we do not insist that integer overflow and buffer overflow is the most common bugs in all circumstances. If we obtain another bug types with different dataset, we can still apply same data collection and experiment methodology to analyse fix patterns for those bug types. Only if we used wrong process then this case study will not be valid.

Another limitation is related with repair tool we used in our experiment. F1X is categorized as heuristics-based repair and our finding may not be generalized to repair tools with other type of patch generation technique. We plan to include more repair tools for better comparison as our future works.

4.3. Bug Statistics

After being able to reproduce bugs from OSS-Fuzz, we are able to categorize it according to the content of their sanitizer error message and stack trace. We collected total of 240 bugs from five C open-source projects in OSS-Fuzz: FFMPEG, Wireshark, PROJ4, OpenJPEG and Libarchive.

We classified these security bugs according to its manual fix pattern. Each closed bug is linked to their fix in the online versioning software and hence we are able to examine them manually. From the point of view of repair tools, fix with more lines are more difficult to generate.

In fact, it is a well-known that most repair tools are only able to generate single line of fix. Hence, we classified all bug fixes into three types according to the number of inserted and deleted lines:

- **One-line fix**. This fix type is able to be generated by repair tools and the fix pattern is also easy to be recognized by humans.
- **Small multiple-line fix**. The fixes in this category have no more than 10 modified lines (both insertion and deletion combined). The fix in this category cannot be generated by repair tools. However, the fix is still simple enough for human to recognize fix pattern and do some analysis.
- **Large multiple-line fix**. The fixes in this category include fixes that have more than 10 modified lines or fixes with multiple-files. The complexity make it infeasible for repair tools to generate and for humans to recognize fix patterns.

The breakdown of the bugs and fix patterns by each category are displayed in charts below. As shown from the chart, the majority of bugs in total summary consists of two security bug types:

integer overflow and buffer overflow. Both bug types represent 144 cases out of 240 (60% of all bugs).

Bug Types	**Number of Bugs**
Integer Overflow	110
Buffer Overflow	34
Memory Leak	22
Segmentation Fault	22
Division by Zero	14
Out of Memory	11
Deadly Signal	10
Timeout	7
Array Out-of-Bound	5
Stack Use After Return	1
Heap Use After Free	1
Hang	1
No Category	2
Total	**240**

Table 1: Security bugs categorized by their type

Similarly for most individual projects, the most prevalent bugs are either integer overflow (in FFMPEG, Wireshark and OpenJPEG) or buffer overflow (in Libarchive). Only for PROJ4, the most prevalent bug is memory leak.

Bug Types	**FFMPEG**	**Wireshark**	**PROJ.4**	**OpenJPEG**	**Libarchive**
Int Overflow	81	9	4	9	7
Buf Overflow	15	7	2	2	8
Memory Leak	4	4	12	0	2
Seg Fault	5	4	10	2	1
Div By Zero	1	1	12	0	0
Out of Memory	8	2	0	0	1
Deadly Signal	4	3	0	6	0
Timeout	1	4	1	0	1
Out-of-Bound	5	5	0	0	0
Stack Aft-Ret	0	1	0	0	0
Heap Aft-Free	0	0	1	0	0
Hang	0	0	0	0	1
No Category	0	0	0	1	1
Total	**124**	**32**	**42**	**20**	**22**

Table 2: Security bugs breakdown in each target projects

From the Table 3, it can be seen that bug type with most common one-line fix is also integer and buffer overflow (63 out of 76 fixes). Fix-rate wise, heap use after free (100%), buffer overflow (47%) and integer overflow (43%) are most common bugs with one-line fix. However, heap use after free only consists of 1 bug and we do not have sufficient data for meaningful analysis. Only integer and buffer overflow will be included in our analysis.

Bug Types	**One-line Fix**	**Multilines Small Fix**	**Multilines Large Fix**
Int Overflow	47	39	24
Buf Overflow	16	8	10
Memory Leak	4	7	11
Seg Fault	4	10	8
Div By Zero	1	6	7
Out of Memory	1	7	3
Deadly Signal	1	2	7
Timeout	0	4	3
Out-of-Bound	1	2	2
Stack Aft-Ret	0	1	0
Heap Aft-Free	1	0	0
Hang	0	1	0
No Category	0	1	1
Total	**76**	**88**	**76**

Table 3: Security bugs breakdown based on their manual fix types

Out of memory (63.6%), timeout (57.8%) and segmentation fault (45.4%) are bugs with most common small multiple-lines fix. In contrast, deadly signal (70%), memory leak (50%) and division by zero (50%) are bugs with largest percentage for large multiple-lines fix.

From these statistics, we conclude that integer overflow and buffer overflow are the bug types with most potential to be automatically fixed with repair tools since they have largest percentage of one-line fix.

4.4. Fix Pattern Analysis

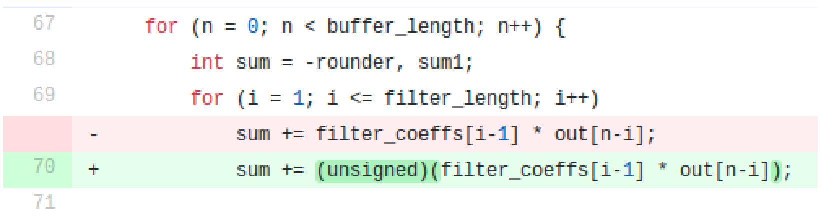
In this section, we will focus on analysing for one-line fix and small multiple-lines fix. One-line fix is analysed because most of repair tools are only able to fix bug in this category. We analysed small multiple-lines fix to deduce fix pattern by manual observation. For the sake of simplicity, we will not analyse large multiple-lines fixes since it is computationally too costly for repair tools to generate and validate and it is also infeasible for us to analyse the pattern without any domain knowledge for individual projects. Specifically, we will analyse one-line fixes for integer overflow and buffer overflow since they have the largest proportion of one-line fixes compared to other bug types.

4.4.1. One-Line Fixes

For integer overflow cases, there are three common fix patterns in one-line fix category:

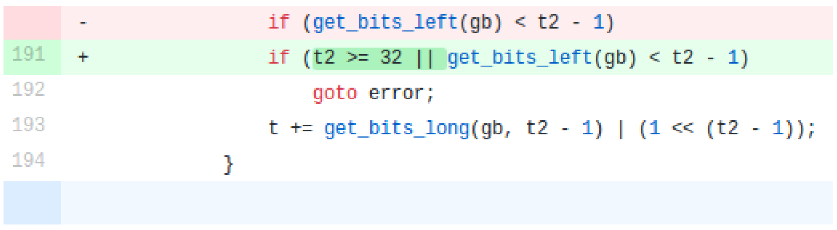
- Explicit casting on variable or constant (53.1%)
- Changing if-check (12.7%)
- Changing type declaration (12.7%)
- Others (e.g. changing function parameter, macro, assignment)

In explicit casting and changing type declaration, most common pattern is to convert signed integer to unsigned integer. From their sanitizer error message and stack trace, we observed that most values of involved variables are large positive and hence they are casted to data type which has larger positive range. For if-check addition, the most common condition is checking a boundary value for an integer that is involved in integer overflow error and return from function after optional function call for logging purpose.



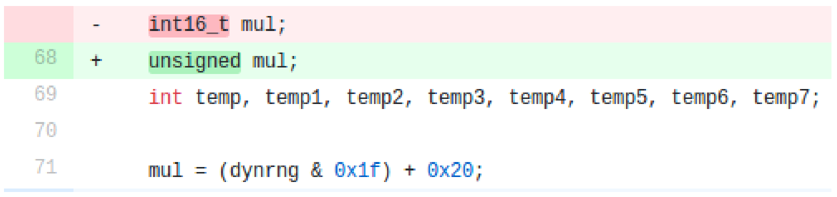
```
      for (n = 0; n < buffer_length; n++) {
          int sum = -rounder, sum1;
          for (i = 1; i <= filter_length; i++)
-             sum += filter_coeffs[i-1] * out[n-i];
+             sum += (unsigned)(filter_coeffs[i-1] * out[n-i]);
```

Figure 6: One-line fix for integer overflow by explicit casting



```
-            if (get_bits_left(gb) < t2 - 1)
+            if (t2 >= 32 || get_bits_left(gb) < t2 - 1)
                 goto error;
             t += get_bits_long(gb, t2 - 1) | (1 << (t2 - 1));
         }
```

Figure 7: One-line fix for integer overflow by changing if-condition

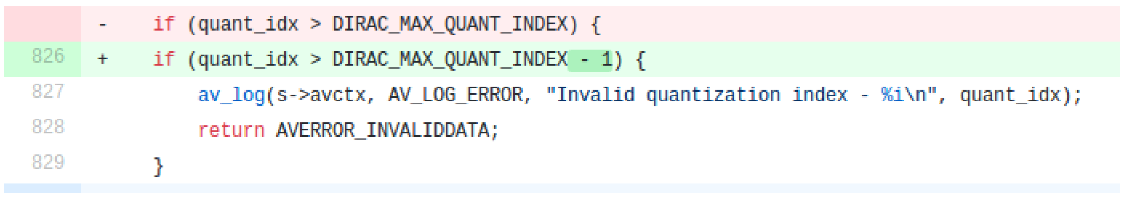


```
-    int16_t mul;
+    unsigned mul;
     int temp, temp1, temp2, temp3, temp4, temp5, temp6, temp7;

     mul = (dynrng & 0x1f) + 0x20;
```

Figure 8: One-line fix for integer overflow by changing type declaration

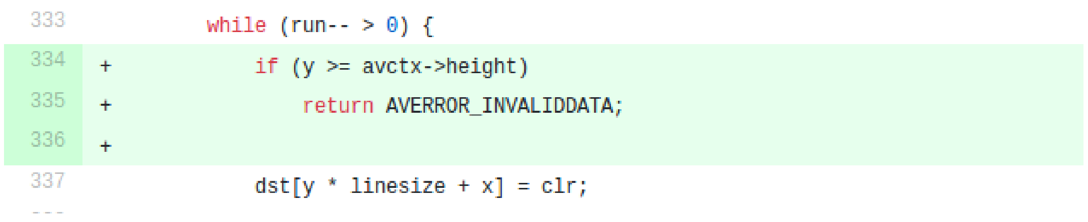
For buffer overflow, their root causes of the error can be largely divided out-of-bound array access and incorrect parameter of memory-related C functions, such as memcpy, malloc and sprintf. We observe that there are two common fix types:

- Changing if-condition that will check for erroneous value and return before reaching error line (25% of fixes). Authors either change operators with similar range (e.g. from x >y to x ⩾ y) or their operands slightly (e.g. x ⩾ y to x ⩾ y+1). A lot of times, one of the operands is an array or pointer index that is used in the error line.
- Adding if-check that will check for erroneous value and return before reaching error line (25% of fixes). Same as first type, one of the operands is an array or pointer index that is used in the error line.



```
-    if (quant_idx > DIRAC_MAX_QUANT_INDEX) {
+    if (quant_idx > DIRAC_MAX_QUANT_INDEX - 1) {
         av_log(s->avctx, AV_LOG_ERROR, "Invalid quantization index - %i\n", quant_idx);
         return AVERROR_INVALIDDATA;
     }
```

Figure 9: One-line fix for buffer overflow by changing operand value by 1



```
        while (run-- > 0) {
+           if (y >= avctx->height)
+               return AVERROR_INVALIDDATA;
+
            dst[y * linesize + x] = clr;
```

Figure 10: Multiple-lines fix for buffer overflow by adding if-condition

However, the fix pattern for buffer overflow is not as strong as integer overflow. A significant proportion of the fixes (around 50%, compared to only 20% for integer overflow) cannot be summarized into simple, visible patterns. Most of these fixes include adding or changing new function before error line. Although they require only one-line but we considered this as complex fix because a function is essentially a sequence of statements that are summarized in function name. Hence, it will require deeper analysis and effort before it can be integrated into automated repair tools.

4.4.2. Multiple Line Fixes

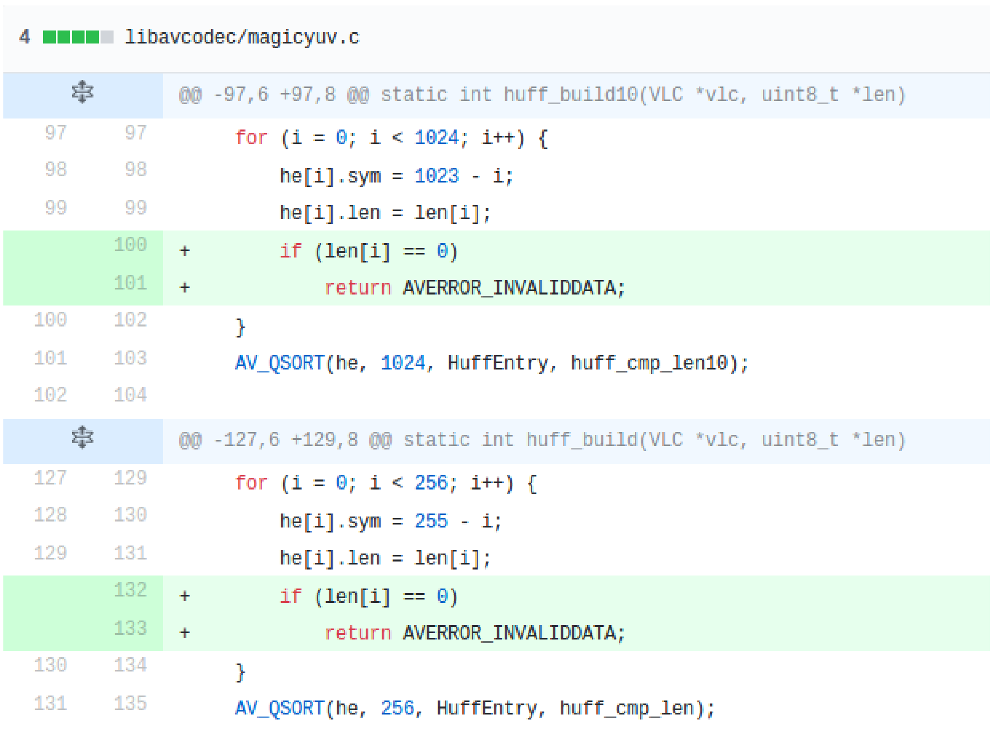
The root causes for integer overflow bugs which has multiple-lines fixes are the same as those that have one-line fix. The difference is that in many cases, the small multiple lines fix is the multiple repetitive patterns of single-line fix. We called this fix pattern as ***parallel fix***. Their fixes are basically classified into the following categories:

- Parallel addition/changing/removal of if-statement (12.8%)
- Parallel explicit casting and/or change of type declaration (15.3%)
- Parallel change of operator/statements/functions (10.2%)
- Single addition/changing/removal of if-statement (12.8%)
- Others (Combination of changing if-check, explicit casting and change of type declaration)

For fix of type 4 and 5, they are self-explanatory. However, we will give an example for each parallel fix types with their rationale why they are used to fix bugs.

1) Parallel addition/changing/removal of if-statement

We have a fix of parallel addition of if-statement for FFMPEG integer overflow in figure below:



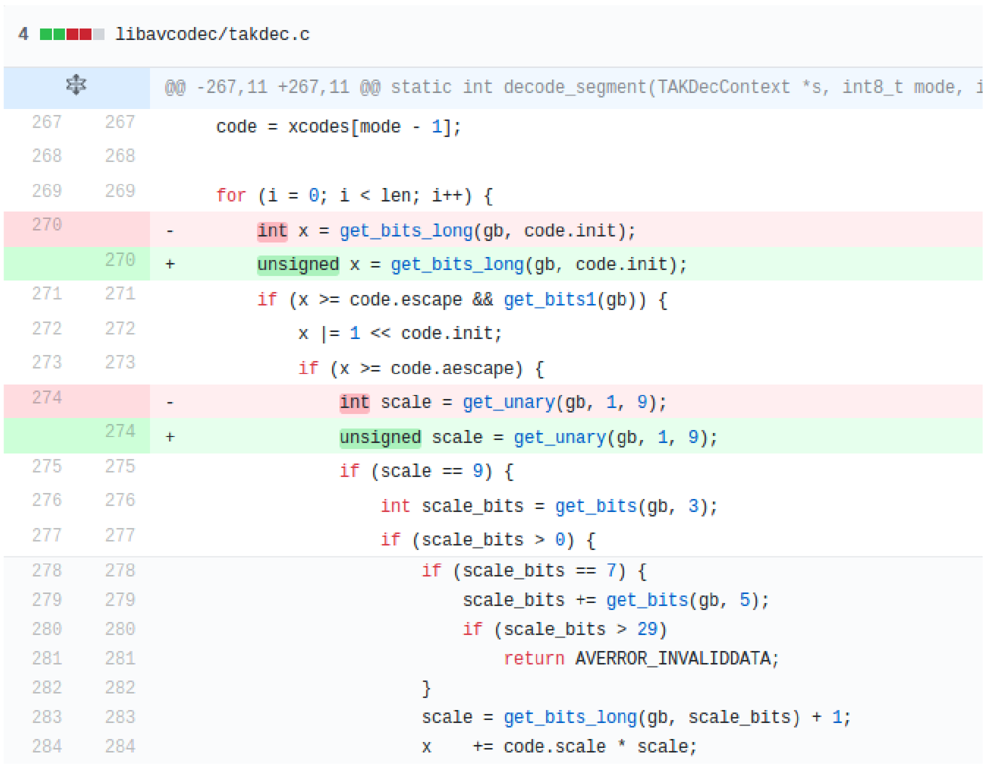
```
4 libavcodec/magicyuv.c
@@ -97,6 +97,8 @@ static int huff_build10(VLC *vlc, uint8_t *len)
     for (i = 0; i < 1024; i++) {
         he[i].sym = 1023 - i;
         he[i].len = len[i];
+        if (len[i] == 0)
+            return AVERROR_INVALIDDATA;
     }
     AV_QSORT(he, 1024, HuffEntry, huff_cmp_len10);

@@ -127,6 +129,8 @@ static int huff_build(VLC *vlc, uint8_t *len)
     for (i = 0; i < 256; i++) {
         he[i].sym = 255 - i;
         he[i].len = len[i];
+        if (len[i] == 0)
+            return AVERROR_INVALIDDATA;
     }
     AV_QSORT(he, 256, HuffEntry, huff_cmp_len);
```

Figure 11: Parallel fix for integer overflow by multiple addition of if-statement

We are able to verify that function huff_build and huff_build10 have almost the same structure (same variables, operations, function calls, etc) except in just their constant indices. These functions are located in the same file and integer overflow bug was identified in one of these function (huff_build) and addition of if-check is added before error line. Therefore, it is necessary that the same repair is also applied to its twin function.

2) Parallel change of type declaration

When error happened, often there are more than one operator involved. In such case, programmers often change type declaration of all operators that were involved in that arithmetic operation. In example below, the error happened at line 284 where both variables x and scale are involved. The developer made fix by changing data type of both x and scale.

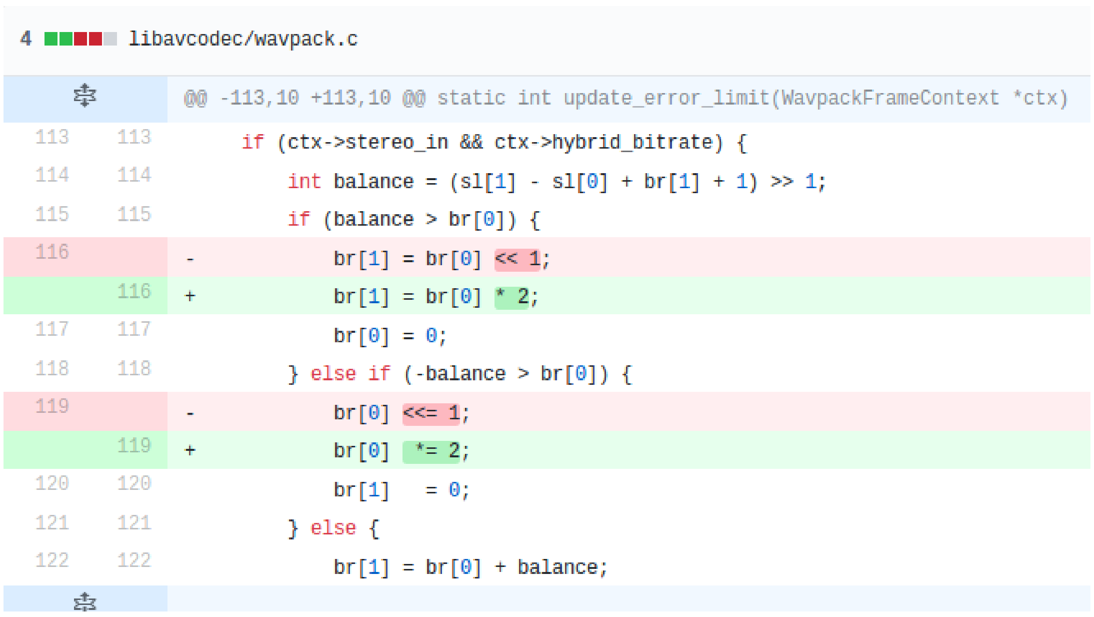


```
4 libavcodec/takdec.c

     @@ -267,11 +267,11 @@ static int decode_segment(TAKDecContext *s, int8_t mode, i
267 267       code = xcodes[mode - 1];
268 268
269 269       for (i = 0; i < len; i++) {
270     -         int x = get_bits_long(gb, code.init);
    270 +         unsigned x = get_bits_long(gb, code.init);
271 271           if (x >= code.escape && get_bits1(gb)) {
272 272               x |= 1 << code.init;
273 273               if (x >= code.aescape) {
274     -                 int scale = get_unary(gb, 1, 9);
    274 +                 unsigned scale = get_unary(gb, 1, 9);
275 275                   if (scale == 9) {
276 276                       int scale_bits = get_bits(gb, 3);
277 277                       if (scale_bits > 0) {
278 278                           if (scale_bits == 7) {
279 279                               scale_bits += get_bits(gb, 5);
280 280                               if (scale_bits > 29)
281 281                                   return AVERROR_INVALIDDATA;
282 282                           }
283 283                           scale = get_bits_long(gb, scale_bits) + 1;
284 284                           x    += code.scale * scale;
```

Figure 12: Parallel fix of integer overflow by multiple changing type declaration

3) Parallel change of statements/operators/functions

We also have fixes that change statement in parallel. In this illustration, the arithmetic operation which generated error message present inside if-else condition. However, the similar arithmetic operation also present at the another branch and hence the same fix applies to that branch (See figure below).



```
4 libavcodec/wavpack.c
@@ -113,10 +113,10 @@ static int update_error_limit(WavpackFrameContext *ctx)
        if (ctx->stereo_in && ctx->hybrid_bitrate) {
            int balance = (sl[1] - sl[0] + br[1] + 1) >> 1;
            if (balance > br[0]) {
-               br[1] = br[0] << 1;
+               br[1] = br[0] * 2;
                br[0] = 0;
            } else if (-balance > br[0]) {
-               br[0] <<= 1;
+               br[0]  *= 2;
                br[1]   = 0;
            } else {
                br[1] = br[0] + balance;
```

Figure 13: Parallel fix of integer overflow by multiple statement changes

5. Experiment Result

In this section, we will elaborate on the result of running F1X repair tool to fix integer overflow bug and analyse the finding. In our case study, we deliberately focus on integer overflow as they comprise majority of security bugs in our dataset. However, we also plan to expand our investigation on other security bug types as well.

5.1. Fix Statistics

Out of 240 bugs, only in 115 cases (47.9%) F1X successfully generated fixes. If we classified those 115 cases based on their manual fix category, small multiple-lines fix obtained highest fix rate at 68.2% (60 out of 88 bugs), followed by large multiple-lines fix at 50% (38 out of 76 bugs). Surprisingly, although F1X is supposed to be the best for generating one-line fix, it performed worst with those bugs with one-line manual fix at 22.3% (only 17 out of 76 bugs). We investigated root cause of the difficulty of fix generation in Section 5.2 (Finding Analysis).

The breakdown of bugs with successful fix generation by F1X during our experiment is shown in the table below:

Bug Types	**One-line Fix**	**Multilines Small Fix**	**Multilines Large Fix**
Int Overflow	9	28	18
Buf Overflow	8	7	5
Memory Leak	0	2	1
Seg Fault	0	7	2
Div By Zero	0	4	3
Out of Memory	0	5	1
Deadly Signal	0	3	4
Timeout	0	2	1
Out-of-Bound	0	1	2
Stack Aft-Ret	0	0	0
Heap Aft-Free	0	0	0
Hang	0	1	0
No Category	0	0	1
Total	**17**	**60**	**38**
Fix Percentage	**22.3%**	**68.2%**	**50%**

Table 4: Statistical result of fixed bugs by F1X categorized by their manual fix types

For other 125 bugs that cannot be fixed by F1X, 51 of them (40.8%) are bugs with manual one-line fix with large majority of them are belongs to integer overflow category. Multiple-lines fixes comprise of 29 bugs (23.2%) with integer overflow as also most common bug category.

Bug Types	**One-line Fix**	**Multilines Small Fix**	**Multilines Large Fix**
Int Overflow	35	6	5
Buf Overflow	5	2	1
Memory Leak	4	4	1
Seg Fault	2	3	2
Div By Zero	1	1	1
Out of Memory	1	0	1
Deadly Signal	1	0	1
Timeout	0	0	0
Out-of-Bound	1	1	0
Stack Aft-Ret	0	0	0
Heap Aft-Free	1	0	0
Hang	0	0	0
No Category	0	0	0
Total	**51**	**17**	**12**
Unfix Percentage	**40.8%**	**13.6%**	**9.6%**

Table 5: Statistical result of unfixed bugs by F1X categorized by manual fix types

Another 26 bugs (20.8%) cannot be fixed because various issues, such as compilation failures, project dependencies issues, or F1X inability to instrument header file. And the rests, their manual fixes are located in multiple files (15.2%).

5.2. Finding Analysis

In this section, we report some findings from our fix statistics and challenges we encountered during our experiment and derive conclusion based on them.

Finding 1 – Technical Challenge. Integer overflow bugs that have manual one-line fix comprise majority of unfixed bugs. After some discussion and analysis, we realized that the root cause is because the casting operator and changing type declaration are not within F1X operator template. Hence, F1X failed to generate fixes for these bugs because it cannot synthesize them. We also found that security-related bugs are only reproducible if the target projects are compiled with appropriate sanitizers. This will cause issue with generate-and-validate repair tools if they are not designed to include sanitizers and hence, unable to fix those bugs.

Finding 2 – Performance Challenge. After we implemented casting operator into F1X and testing them for a small C program, the repair tool still failed to generate fix for integer overflow bug. We investigated and found that F1X has an optimization algorithm of test-equivalence in which possible patches are being classified into classes during program execution if those patches have same execution traces and result for same test cases. However, for integer overflow, it is possible that the patches are not the same even if they have same execution traces. One patch could cause the overflow for sign-bit and another one does not. In such scenario, F1X test-equivalence algorithm could make itself failed to find appropriate patching solution. To resolve this issue, the test-equivalence algorithm need to be deactivated and this cause performance overhead during repair process. This finding tell us that optimization algorithm may compromise ability to generate correct patches.

Finding 3 – Patch Quality Challenge. The use of sanitizers during compilation of target projects broke down test suite of our target projects. Many executable always return sanitizer error message even when we execute them with default options using default test files. This error shows that currently existing open-source projects are still not developed with proper security in mind. An availability of test suite is a mean by which generate-and-validate repair tools can assess the quality of generated patches. Since the test suite broke down upon being compiled by sanitizers, we suggest another mean by which repair tools are able to assess patch quality. Good example would be to use previous manual fix patterns, as available from OSS-Fuzz.

5.3. Impact Analysis

We have conducted a case study by examining large number of security-related bugs. Based on our finding, it will bring the following advantages for research community and developers of repair tools:

- **Benefit 1 – Distribution of security-related bugs**. The statistics from this case study will provide a layout on most common security bugs on real-world programs written in C. Together with manual fix statistics, it also provides an overview of bug classes with most difficult and easiest fixes (i.e. bugs with most one-line fixes and large multiple-line fixes). This will be helpful for research community to focus on specific bug classes with more opportunity for successful repair.
- **Benefit 2 – Manual fix patterns of security bugs**. We also conducted comprehensive manual study for fix pattern for security bugs and identified most common fix patterns for integer overflow and buffer overflow, both for one-line fixes and multiple-line fixes. This finding will be useful for developers of general-purpose repair tools who want to customize their repair operator and search space in patch generation process for handling specific security bugs.
- **Benefit 3 – Findings on general-purpose tool to fix security bugs**. In previous section, we have identified findings and challenges we encountered during our experiment to run general purpose tools, F1X. These findings can be used in research community to design new repair tools which will be able to address these challenges and perform better to fix security related bugs.

6. Related Works

In this section, we will present literature review on related works: existing latest general purpose automated repair tools, case studies on analysing bugs and its fixes for program repair, and miscellaneous works which inspire our case study.

6.1. Automated Repair Tools

Automated repair tools have been successfully applied for providing debugging hints [26], automatically grading assignments [27] [28] and patching security vulnerabilities [23]. Some of the automatic repair tools are general purpose, such as GenProg [11], SemFix [12], PAR [13], Prophet [20], Angelix [23], and F1X [25] intended to repair general software bugs, while others are used to repair specific bugs such as integer overflow or buffer overflow [16] [29] [17] [30].

GenProg [11] used genetic programming technique to generate fixes. It implements simple fault localization strategy by favouring locations that are visited by negative test cases. It further limited search space to those fixes that are similar to other parts of the program. It stops when it finds repair candidate that pass all test cases in test suite. This technique relies on the fact that similar approach is also used in human programming (i.e. Programmers often copy code from one part of the program and paste it to another location to fix errors). GenProg was evaluated on sixteen programs in C and it showed an ability to fix various type of errors.

SemFix [12] is a repair tool which identified list of potential erroneous program statements by using Tarantula fault localization technique. It basically ranked suspicious program statements based on their frequency of execution for failing test cases in given test suite. After suspicious statements are identified, SemFix used symbolic execution and derived a path condition required to reach those statements for each failing test case. After that, it used constraint solving and program synthesis technique to generate patches that satisfy that path conditions. SemFix reported that it can fix more bugs compared to repair using genetic programming (GenProg).

The article [13] proposes PAR an approach to automated repair based on the observation that genetic programming-based patch generation technique GenProg [31] [11] can generate nonsensical patches due to the randomness of the application of mutation operators. GenProg works by generating variants of the program by using crossover operators and mutation operators. Then it runs test cases to validate the variants and iterates the steps until one of the variants passes all test cases. The limitation of GenProg is that since the mutations are random, it is possible to generate nonsensical patches. In PAR, instead of random mutation, the authors carefully inspected 62,656 human-written patches in open source projects and manually came up with 10 fix templates, which are program transformation scripts based on the fix patterns. Patches generated by their approach are much more acceptable by human subjects than the patches generated by GenProg. The generated patches are also more successful, where it fixes 26 out of 119 bugs.

The article [20] introduces Prophet, which uses an approach akin to machine learning to learn the features of correct code to select plausible automatically-generated patches. It is based on the hypothesis that, even across applications, correct code shares properties that can be learned and exploited to generate correct patches for incorrect applications. For real-world defects drawn from eight large open source applications, its fixes outperform previous generate-and validate path generation systems, including SPR [14], Kali, GenProg [11], and AE.

Angelix [23] is a semantic repair tool which use symbolic execution and constraint solving to generate patches. Angelix offered scalability compared to previous semantic based tools, such as SemFix [12], because it extracts repair constraint to a structure called angelic forest whose size is independent of the repaired program. Additionally, by using information in angelic forest, Angelix is able to generate multi-line fixes for large real-world programs.

The tool F1X implements the test-equivalence-based program repair presented in [25]. In program repair, the set of plausible repairs are tested to refine the candidate repairs in the set. Here, there is a large number of tests that need to be performed. This approach of F1X applies test equivalence to reduce the tests to just the representative of the equivalence classes. Our work improves this approach further by refining the repairs produced by the tool into those that more closely resemble the repairs done by human programmers.

The article [31] proposes static analysis-based approach to add input filters to programs to prevent integer overflow errors. It is based on computing (an approximation of) weakest precondition from the critical expression from memory allocation and block copy sites. The main efficiency measure is the ignoring of the conditions of the conditional expressions when computing the weakest precondition. The filters are sound: if an input passes the filter, it will not trigger an integer overflow error at any analysed site.

The article [19] is using Q&A sites such as Stack Overflow to find a bug fix to automatically patch bugs. The system performs web query using strings from the Java stack trace and perform fuzzy program analysis techniques to extract the patch from the Q&A page.

6.2. Study on Bugs and Fixes

Di Franco et al [33] conducted a survey on 269 numerical floating point bugs and classified them into four groups according to their root causes. A study by Zhong and Su [34] analysed more than 9,000 collected real-world bugs in Java and derived several insights from these bugs such as fault distribution, fault complexity, and bug fix patterns (e.g. use of mutation operators, APIs, file types for bug fixes). Ye et al [35] run three repair tools which use static analysis technique (HP Fortify, Checkmarx and Splint) to fix over given 100 buffer overflow bugs. They analysed root causes of false positives and false negatives for these bugs and also summarized fix patterns for guiding repair tools for buffer overflow. Campos and Maia [36] studied two distinct datasets of 4 million bug-fix commits from 101,471 projects and 369 bug fixes from five open-source projects. From these datasets, they identified five most common bug fix patterns.

The article [37] presents experimental results on generate-and validate automated repair tools SPR and Prophet. The article distinguishes plausible patches (passing tests) from correct patches (do not have latent defects and do not introduce new defects or vulnerabilities).

It pointed out that impoverished search space containing very few correct patches is one of the reasons for the poor performance of some prominent previous patch generation systems. Correct patches occur only sparsely within the search space. However, plausible but incorrect patches are relatively abundant (in comparison with correct patches). SPR and Prophet’s patch prioritization mechanisms are both effective at isolating correct patches. The article concludes that generate-and-validate systems must exploit information beyond test suite, e.g., using specification mining: the specification can be used to filter incorrect patches

6.3. Miscellaneous

The article [38] presents recommendations for researches in program repair. The author emphasizes the importance to identify the defect classes that are repaired. This enables the community to answer the question on what are repairable defect classes and why a defect class is easy or hard to repair. The defect categorization is also essential for evaluation: A dataset which contains more defects where a particular tool is effective, will favor the tool.

The article [39] presents automatic exploit generation (AEG), a complete (end-to-end) system to automatically generate control flow exploits, which are exploits where the system being attacked executes arbitrary code provided by the attacker. The system is based on a combination of source-code analysis (LLVM IR) using static symbolic execution and dynamic analysis. The symbolic execution is preconditioned, which means that paths are explored selectively to focus only on parts of the code with potential control-flow vulnerability. This improves scalability.

Urli et al [40] presents Repairnator, an autonomous agent that constantly monitors test failures on continuous integration (CI) system, reproduces bugs, and runs program repair tool, and suggest the repair to developers. The approach taken seems ineffective: among 11,523 test failures, only patches for 15 different bugs were generated, all of which are overfit.

7. Future Works

We have done a case study using a general-purpose repair tool, F1X, to fix security related bugs and derived three findings from integer overflow security bugs. We noted that this study has not analysed other security bug types. In the future, we plan to expand our case study to other bug types with more repair tools.

To address patch quality challenge caused by unavailability of test suite, we will compare patch prioritization strategies using past fixes. Prophet, one of the latest repair tools, use machine learning to mine previous successful human patches and rank generated patches according to their similarity with previous patches. However, such technique is designed to handle general and not security-related bugs. OSS-Fuzz contains past successful fixes for security-related bugs and it will be interesting to compare performance of Prophet and F1X by learning from past general fixes versus past security-related fixes in OSS-Fuzz.

We also noted that there have been many tools released to fix security-related bugs, such as integer overflow and buffer overflow. We plan to compare some of these tools against general purpose tools such as F1X to fix specific bugs.

8. Conclusion

In this work, we have investigated 240 security-related bugs and analysed both manual fix patterns and fixes generated by general-purpose repair tool, F1X. We have found that integer overflow and buffer overflow are the most common bugs encountered security bug type but it is also easiest to be fixed as most of their manual fixes are single-line fix.

The single-line fix for integer overflow can be summarized in three patterns: explicit casting, changing type declaration and changing if-condition. The multiple-line fixes can be summarized into three types: parallel fix, combination of single-line fix, and addition of if-condition. For buffer overflow, their most common single-line fix is changing of if-condition and addition of if-condition for multiple-line fix.

We also identified three challenges (technical, performance and patch quality) after running repair experiment with F1X and plan to address these challenges in our future work to create more comprehensive case study. These findings will be useful for developers who intend to optimize their repair tools.

References

[1] D. Molnar, X. C. Li and D. A. Wagner, “Dynamic Test Generation To Find Integer Bugs in x86 Binary Linux,” in *USENIX Security*, 2009.

[2] L. T. Wang, T. Wei, Z. Q. Lin and W. Zou, “IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary,” in *Network and Distributed System Security (NDSS)*, 2009.

[3] X. Wang, H. G. Chen, Z. H. Jia, N. Zeldovich and M. F. Kaashoek, “Improving integer security for systems with KINT,” in *USENIX conference on Operating Systems Design and Implementation*, 2012.

[4] M. Abadi, M. Budiu, U. Erlingsson and J. Ligatti, “Control-Flow Integrity,” in *Computers and Communication Security (CCS)*, 2005.

[5] “AFL,” 2018. [Online]. Available: http://lcamtuf.coredump.cx/afl/. [Accessed 3 April 2018].

[6] “Libfuzzer,” 2018. [Online]. Available: http://llvm.org/docs/LibFuzzer.html. [Accessed 3 April 2018].

[7] M. Boehme, V. T. Pham and A. Roychoudhury, “Coverage-based Greybox Fuzzing as Markov Chain,” in *Conference on Computer and Communications Security (CCS)*, 2016.

[8] “OSS-Fuzz: Continuous Fuzzing of Open-Source Software,” Google, [Online]. Available: https://github.com/google/oss-fuzz. [Accessed 3 April 2018].

[9] M. Boehme, V. T. Pham, M. D. Nguyen and A. Roychoudhury, “Directed Greybox Fuzzing,” in *Conference on Computer and Communications Security (CCS)*, 2017.

[10] M. Monperrus, “Automatic Software Repair: a Bibliography,” *ACM Computing Surveys,* pp. 1-24, 2017.

[11] W. Weimer, T. V. Nguyen, C. Le Goues and S. Forrest, “Automatically finding patches using genetic programming,” in *International Conference on Software Engineering (ICSE)*, 2009.

[12] H. D. T. Nguyen, D. W. Qi, A. Roychoudhury and S. Chandra, “SemFix: program repair via semantic analysis,” in *International Conference on Software Engineering (ICSE)*, 2013.

[13] D. S. Kim, J. C. Nam, J. W. Song and S. H. Kim, “Automatic Patch Generation Learned from Human-Written Patches,” in *International Conference on Software Engineering (ICSE)*, 2013.

[14] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in *Foundations of Software Engineering (FSE)*, 2015.

[15] W. Weimer, “Patches as better bug reports,” in *Generative Programming and Component Engineering (GPCE)*, 2006.

[16] X. Cheng, M. Zhou, X. Song, M. Gu and J. G. Sun, “IntPTI: Automatic Integer Error Repair with Proper-Type Inference,” in *Automated Software Engineering (ASE)*, 2017.

[17] F. J. Gao, L. Z. Wang and X. D. Li, “BovInspector: Automatic inspection and repair of buffer oveflow vulnerabilities,” in *Automated Software Engineering (ASE)*, 2016.

[18] Q. Gao, Y. F. Xiong, Y. Q. Mi, L. Zhang, W. K. Yang, Z. P. Zhou, B. Xie and H. Mei, “Safe Memory-Leak Fixing for C Programs,” in *International Conference on Software Engineering (ICSE),* 2015.

[19] Q. Gao, H. S. Zhang, J. Wang, Y. F. Xiong, L. Zhang and H. Mei, “Fixing Recurring Crash Bugs via Analyzing Q&A Sites (T),” in *Automated Software Engineering (ASE),* 2015.

[20] F. Long and M. Rinard, “Automatic patch generation by learning correct code,” in *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL),* 2016.

[21] X. B. Le, D. H. Chu, D. Lo, C. L. Goues and W. Visser, “S3: Syntax and Semantic-Guided Repair Synthesis via Programming by Examples,” in *Foundations of Software Engineering (FSE),* 2017.

[22] L. D'Antoni, R. Samanta and R. Singh, “Qlose: Program Repair with Quantitative Objectives,” in *International Conference on Computer Aided Verification*, 2016.

[23] S. Mechtaev, J. Y. Yi and A. Roychoudhury, “Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis,” in *International Conference on Software Engineering,* 2016.

[24] “Abstract Syntax Tree,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Abstract_syntax_tree. [Accessed 09 April 2018].

[25] S. Mechtaev, S. H. Tan, G. Xiang and A. Roychoudhury, “Test-Equivalence Analysis for Automatic Patch Generation,” in *(Under Submission),* 2018.

[26] Y. D. Tao, J. D. Kim, S. H. Kim and C. Xu, "Automatically generated patches as debugging aids: a human study," in *Foundations of Software Engineering (FSE)*, 2014.

[27] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki and B. Hartmann, "Learning syntactic program transformations from examples," in *International Conference on Software Engineering (ICSE)*, 2017.

[28] J. Y. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," in *Foundations of Software Engineering (FSE)*, 2017.

[29] S. Ding, H. B. K. Tan and H. Y. Zhang, "ABOR: An Automatic Framework for Buffer Overflow Removal in C/C++Programs," in *International Conference on Enterprise Information Systems (ICEIS)*, 2014.

[30] A. Shaw, D. Doggett and M. Hafiz, "Automatically Fixing C Buffer Overflows Using Program Transformations," in *Dependable Systems and Networks (DSN)*, 2014.

[31] 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 *International Conference on Software Engineering (ICSE)*, 2012.

[32] F. Long, S. Sidiroglou-Douskos, D. K. Kim and M. Rinard, "Sound input filter generation for integer overflow errors," in *Symposium on Principles of Programming Languages (POPL)*, 2014.

[33] A. Di Franco, H. Guo and C. Rubio-Gonzalez, "A Comprehensive Study of Real-World Numerical Bug Characteristics," in *Automated Software Engineering (ASE)*, 2017.

[34] H. Zhong and Z. D. Su, “An Empirical Study on Real Bug Fixes,” in *International Conference on Software Engineering (ICSE)*, 2015.

[35] Y. Tao, L. M. Zhang, L. Z. Wang and X. D. Li, “An Empirical Study on Detecting and Fixing Buffer Overflow Bugs,” in *IEEE International Conference on Software Testing, Verification and Validation,* 2016.

[36] E. Campos and M. Maia, “Common Bug-Fix Patterns: A Large-Scale Observational Study,” in *International Symposium on Empirical Software Engineering and Measurement,* 2017.

[37] F. Long and M. Rinard, “An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems,” in *International Conference on Software Engineering (ICSE),* 2016.

[38] 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 *International Conference on Software Engineering (ICSE),* 2014.

[39] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo and D. Brumley, “Automatic exploit generation,” in *Communcations of the ACM,* 2014.

[40] S. Urli, Z. X. Yu, L. Seinturier and M. Monperrus, “How to Design a Program Repair Bot? Insights from the Repairnator Project,” in *International Conference on Software Engineering (ICSE),* 2018.

[41] Z. C. Qi, F. Long, S. Achour and M. Rinard, “An Analysis of Patch Plausibility and Correctness for Generate-And-Validate Patch Generation Systems,” in *International Symposium on Software Testing and Analysis (ISSTA)*, 2015.