Use of General Repair Tool for Fixing Security Vulnerabilities

Edwin Lesmana Tjiong
Informatics Department
Kalbis Institute
Jakarta, Indonesia
edwin.tjiong@kalbis.ac.id

Sergey Mechtaev
Department of Computer Science
University College London
London, United Kingdom
s.mechtaev@ucl.ac.uk

Harya Bima Dirgantara
Informatics Department
Kalbis Institute
Jakarta, Indonesia
harya.dirgantara@kalbis.ac.id

***Abstract*—Automated patch generation approaches have been shown to address defects in real-world programs, including security vulnerabilities. On the one hand, general repair tools are designed to fix common bugs. On the other hand, specific repair tools targeted security-related vulnerabilities, such as integer or buffer overflow. However, fewer works focus on assessing general repair tools' capabilities to fix security vulnerabilities. The assessment will be helpful to find out if general repair tools can fix security vulnerabilities without knowing specific characterizing patterns of such vulnerabilities in advance, thus not being subjected to overfitting.**

In this paper, we present a detailed analysis of a case study using the semantic general repair tool, F1X, to fix security-related vulnerabilities found by the OSS-Fuzz framework. OSS-Fuzz framework is an automated continuous testing platform run on Google's cloud infrastructure, which can pinpoint source code containing security-related vulnerability and generate its failing test case. Using a dataset of 240 security vulnerabilities found in five open source programs from OSS-Fuzz, we compared and analyzed fix patterns generated by F1X with fixing patterns in OSS-Fuzz Github repositories. We believe that the result of this case study will be insightful for developers to strengthen and optimize their repair tools and security analysts to consider integrating automated repair tools into production systems.

***Index Terms*—Program Repair, Automated Software Testing, Security**

I. Introduction

The complexity of computer software is continuously increasing with tremendous speed. It is common for large-scale software to have a code base of millions line-of-code (LOC). 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 patch bugs before malicious parties can exploit them. Therefore, two technologies have been devised to assist maintenance: automated testing (fuzzing) and automated program repair.

Fuzzing is a technique that uses a program called fuzzer, containing an initial input test case, to execute the program under test. The program under test is run, and at the end of each execution, the input file is mutated, and then the program is re-executed with a regenerated input file. This process is repeated until bugs are found or the user terminates the fuzzing program. The fuzzer is used to automate the bug discovery process.

By using the fuzzer, many security vulnerabilities have been found. However, the overwhelming number of bugs found may not be balanced by the quick response of programmers assigned to fix the bugs. Therefore, we use automated program repair to automate the bug fixing process. An excellent automated repair tool can provide a temporary fix for the system. At the same time, the programmer analyzes and finds the root cause of the security bug before applying a more proper patch accordingly. Afterward, a series of successful human fixes can be used to assess and improve the quality of fixes generated by automated repair tools.

II. Related Works

In this section, we will review past works in fields related to our work: automated testing, automated repair, and the study of bugs and their fixes.

A. Automated Testing

For automated testing, there are many fuzzers developed specialized in discovering security-related bugs, such as integer overflow [1] [2] [3] and buffer overflow [4]. More advanced fuzzing, called *grey box fuzzing*, uses code instrumentation to guide the mutated test input towards exploring more execution paths. The grey box fuzzing approach is able to discover more bugs faster [5] [6] [7]. Using this grey box fuzzing technique, OSS-Fuzz [8] framework has been developed. This framework combined grey box fuzzers, such as AFL [5], libFuzzer [6], and AFLGO [9]. OSS-Fuzz executed these fuzzers continuously against open-source libraries and Chrome components and discovered hundreds of security vulnerabilities and bugs.

B. Automated Repair

In the field of program repair, tools have been developed for providing debugging hints [24], automatically grading assignments [25] [26], and patching security vulnerabilities [22]. Some of the automatic repair tools are general-purpose, such as GenProg [10], SemFix [11], PAR [12], Prophet [19], Angelix [22], F1X [23], designed to repair general bugs. Other

tools are used to repair specific bugs such as integer overflow or buffer overflow [15] [27] [16] [28].

GenProg [10] used genetic programming technique to generate fixes. It implements a simple fault localization strategy by favoring locations visited by negative test cases. It further limited search space to fixes similar to other parts of the program. It stops when it finds a repair candidate that passes all test cases in the test suite. GenProg was evaluated on sixteen programs in C, which showed an ability to fix various types of errors.

SemFix [11] is a repair tool that identifies a list of potential erroneous program statements and ranks those based on their execution frequency for failing test cases. It used symbolic execution and derived path conditions required to reach those statements for each failing test case. After that, it used constraint solving to generate patches that satisfy those path conditions.

Prophet [19] used a machine learning approach to learn the features of correct code to select plausible generated patches. It is based on the hypothesis that valid code shares properties that can be discovered and exploited to create proper patches for incorrect applications.

F1X [23] implements the test-equivalence-based program repair. Generated repair candidates are reduced into their respective equivalence classes. This equivalence-class technique enables F1X to do faster repairs compared to other tools.

Fix2Fit [33] is a repair tool which prioritizes patches based on patch partition which are differentiated by new test cases generated by its fuzzer. Fix2Fit uses sanitizer to get new test cases as an oracle on the newly repaired program.

C. Study on Bugs and Fixes

Di Franco et al. [29] 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 [30] analyzed more than 9,000 collected real-world bugs in Java and derived several insights from these bugs, such as fault distribution, fault complexity, and significant fix patterns (e.g., use of mutation operators, APIs, file types for bug fixes). Ye et al. [31] run three repair tools that use static analysis techniques (HP Fortify, Checkmarx, and Splint) to fix over given 100 buffer overflow bugs. They analyzed the root causes of false positives and negatives for these bugs and summarized fix patterns for guiding repair tools for buffer overflow. Campos and Maia [32] studied two distinct datasets of four-millions bug-fix commits from 101,471 projects and 369 bug fixes from five open-source projects. From these datasets, they identified the five most common bug fix patterns.

III. Experiment Design

To collect data for our experiment, we looked first into the 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 the necessary configurations needed to reproduce each bug, as described in the figure 1.

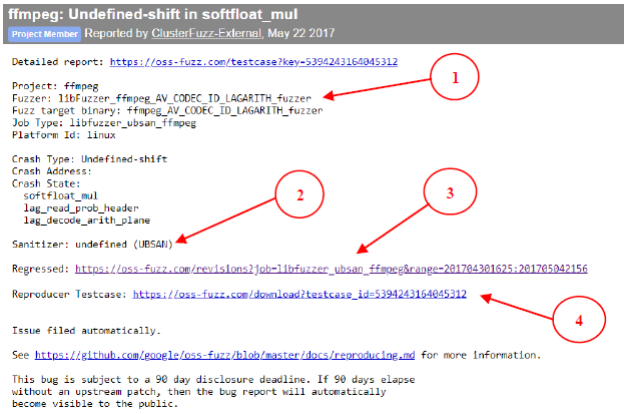

Fig. 1: OSS-Fuzz Bug Report Page

1) Appropriate fuzzer engine to use (shown as number 1 in the figure). Users can only use each fuzzer engine to reproduce a specific type of bug.
2) Appropriate sanitizer to use (shown as number 2 in the figure). A sanitizer is a program that checks certain specified operations inside the source code and triggers an error message when the error happens during runtime execution for that program under test. OSS-Fuzz uses three types of sanitizers: address sanitizer (asan), memory sanitizer (msan), and undefined sanitizer (ubsan). Each of them will produce a specific error message for certain bug types.
3) This is a link to a versioning software site from which we can find a fix for the specific bug (shown as number 3 in the figure). The commit will contain a message like 'Found by OSS-Fuzz'.
4) A failing test case (shown as number 4 in the figure). This test case will be used as an input to the F1X repair tool to validate generated patches.

Using item number 1 and 4, we can construct **driver test file** for each bug. The driver test file is the file that contains the execution command that triggers the target subject, compiled with a specified fuzzer (item 1), with the parameter of failing test case (item 4). Upon execution, the specified bug will appear immediately, as shown in the figure 2 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)
```

Fig. 2: Error trace obtained by giving failing test case to program under test

Next, we also developed another web crawler to extract

sanitizer type from the OSS-Fuzz bug tracker to create **docker command file**. This file is used to build a docker container for reproducing each security bug. Analysts can only correctly produce the error trace for each bug by specifying the correct sanitizer type.

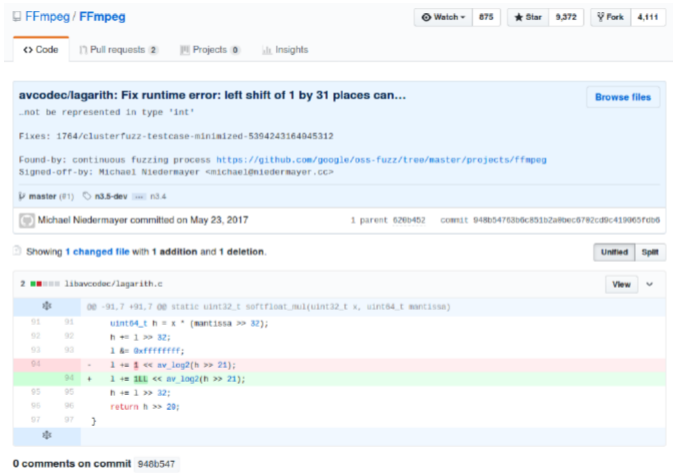

Fig. 3: A bugfix in Github account for FFMPEG linked to the OSS-Fuzz bug report

Lastly, we created another web crawler to check the link of item 3 (containing a range of commits in their Github account) if any commit has a link to the OSS-Fuzz bug tracker. If they have the link, the commit is considered a fix for the bug referred by the OSS-Fuzz bug tracker. A typical example of a commit for such a bug fix is shown in the figure 3.

From the commit page containing the bug fix, we can analyze their fix patterns, both for one-line fixes and multiple-line fixes, for each category of security bugs (such as integer overflow, buffer overflow, and so on). The bug fix page also contains source file(s) with its error line(s). Analysts will use this information for constructing **F1X run command file** to run F1X with the input of the error line.

After constructing the driver test and docker command file, we install both the F1X repair tool [23] and the buggy version of the target program inside a docker container. We run F1X to see if it can fix bugs in our dataset. After finish running, F1X will give statistical figures like number of evaluated patch candidates and executed test cases, number of execution per second, number of plausible patch locations and number of generated patches.

The entire workflow process of our experiment benchmark can be seen in a workflow in figure 4.

IV. Experiment Dataset

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

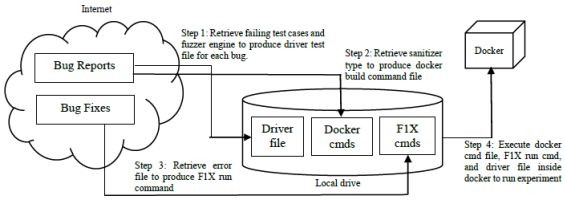

Fig. 4: Workflow of our repair experiment using OSS-Fuzz and F1X

A. Selection Process

These target subjects are selected based on the number of bugs found by OSS-Fuzz and their reproducibility. Initially, we aimed at the projects with the most significant number of bugs and then tried to install those projects in our Docker environment. However, we also found that many security bugs are not reproducible even after executing it with the correct code version and failing test cases obtained from OSS-Fuzz bug trackers. If most bugs are not reproducible, we skipped such target subjects.

Another factor in consideration is the type of programming language. F1X can only fix C projects; hence, we only took target subjects whose majority of codes are written in C. For some projects, we also encountered difficulties during their setup because their build script was modified multiple times across the timeline of the bugs. Each build script includes different library dependencies and different versions of the same libraries. The latest build script might not be able to compile past version code of the target subject and vice versa.

Lastly, there is a compatibility issue related to F1X. Some target subjects generated errors during code instrumentation with F1X because they did not recognize its 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.

B. Bug Statistics

After reproducing bugs from OSS-Fuzz, we can categorize them according to their sanitizer error message and stack trace content. We collected 240 bugs from five C open-source projects in OSS-Fuzz: FFMPEG, Wireshark, PROJ4, OpenJPEG, and Libarchive.

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

It is well-known that most repair tools can only generate a 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 type can be generated by repair tools, and the fix pattern is also easily recognized by humans.

- **Small multiple-line fix**. The fixes in this category have no more than ten modified lines (both insertion and deletion combined). Repair tools cannot generate the fix in this category. However, the fix is still simple enough for a human to recognize its fix pattern and to do some analysis.
- **Large multiple-line fix**. This category's fixes include fixes with more than ten modified lines or fixes with multiple files. The complexity makes it infeasible for repair tools to generate and for humans to recognize fix patterns.

TABLE I: Security bugs categorized by their types

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**

The breakdown of the bugs and fix patterns by each category are displayed in the Table I. As shown from the chart, most bugs consist of two security bug types: integer overflow and buffer overflow. Both bug types represent 144 cases out of 240 (60% of all bugs).

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 pervasive bug is a memory leak. The breakdown can be seen in Table II.

Table III shows that the bug type with the most common one-line fix is also an 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 the most common bugs with a 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 study.

TABLE II: Security bugs breakdown in each target project

Bug Types	FF	WI	PR	OJ	LI
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	0	0	6	0
Timeout	1	4	1	0	1
Out of Bound	5	0	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**

Note: **FF** - FFMPEG, **WI** - Wireshark, **PR** - PROJ.4, **OJ** - OpenJPEG, **LI** - Libarchive

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

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

TABLE III: Security bugs based on their manual fix types

Bug Types	One-line Fix	Multi-line Small Fix	Multi-line 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**

C. Fix Pattern Analysis

In this section, we will focus on analyzing one-line fix and small multiple-lines fix. The one-line fix is investigated because most repair tools can only fix bugs in this category. We analyzed small multiple-line fixes to deduce fix patterns by manual observation. For the sake of simplicity, we will not analyze large multiple-line fixes since it is computationally too costly for repair tools to generate and validate. It is also infeasible to explore the pattern without domain knowledge for individual projects. Specifically, we will analyze one-line fixes for integer overflow and buffer overflow since they have the most significant proportion of one-line fix compared to other bug types.

1) One-Line Fixes: For integer overflow cases, there are three common fix patterns in the 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., edit function parameter, macro, assignment)

In explicit casting and changing type declaration, the most common pattern is to convert signed integer to unsigned integer. From the sanitizer error message and stack trace, we observed that most values of involved variables are positive numbers, which are cast to a data type with a wider positive

range. For if-check addition, the most common condition is checking a boundary value for an integer involved in integer overflow error and returning from operation after an optional function call for logging.

For buffer overflow, their error root causes can be divided into 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 will check for erroneous values and return before reaching the error line (25% of fixes). Either the authors change operators with similar range (e.g., from $x > y$ to $x \geq y$) or their operands slightly (e.g., $x \geq y$ to $x \geq y + 1$). A lot of times, one of the operands is an array or pointer index that is used in the error line.
- Adding if-check will check for erroneous value and return before reaching the error line (25% of fixes). Like the first type, one of the operands is an array or pointer index used in the error line.

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

2) Multiple-Line Fixes: The root causes for integer overflow bugs with multiple-line fixes are the same as those with one-line fixes. The difference is that in many cases, the small multiple-line fix is the multiple repetitive patterns of a single-line fix. We called this fix pattern as **parallel fix**. Their fixes are classified into the following categories:

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

D. Experiment Result

In this section, we discuss the result of running the F1X repair tool to fix the bugs and analyze the finding.

1) Fix Statistics: Out of 240 bugs, only in 115 cases (47.9%) did F1X successfully generate fixes. If we classified those 115 cases based on their manual fix category, small multiple-lines fix obtained the 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 a one-line fix, it performed worst with those bugs with a one-line manual fix at 22.3% (only 17 out of 76 bugs).

Of the other 125 bugs that F1X cannot fix, 51 (40.8%) are bugs with a manual one-line fix; a large majority belong to the integer overflow category. Multiple-line fixes comprise 29 bugs (23.2%), with integer overflow as the most common bug category.

Another 26 bugs (20.8%) not listed inside the table cannot be fixed because various issues, such as compilation failures, project dependencies issues, or F1X's inability to instrument header file. And the rests (19 bugs - 15.2%), their manual fixes are located in multiple files.

TABLE IV: Statistics of fixed and unfixed bugs by F1X

Bug Types	**One-line Fix**	**Multi-line Small Fix**	**Multi-line Large Fix**
Int Overflow	9 (35)	28 (6)	18 (5)
Buf Overflow	8 (5)	7 (2)	5 (1)
Memory Leak	0 (4)	2 (4)	1 (1)
Seg Fault	0 (2)	7 (3)	2 (2)
Div By Zero	0 (1)	4 (1)	3 (1)
Out of Memory	0 (1)	5 (0)	1 (1)
Deadly Signal	0 (1)	3 (0)	4 (1)
Timeout	0 (0)	2 (0)	1 (0)
Out of Bound	0 (1)	1 (1)	2 (0)
Stack Aft-Ret	0 (0)	0 (0)	0 (0)
Heap Aft-Free	0 (1)	0 (0)	0 (0)
Hang	0 (0)	1 (0)	0 (0)
No Category	0 (0)	0 (0)	1 (0)
Total	**17 (51)**	**60 (17)**	**38 (12)**
Fix Percentage	**22.3%**	**68.2%**	**50%**
Unfix Percentage	**40.8%**	**13.6%**	**9.6%**

The breakdown of bugs with successful and unsuccessful fix generation by F1X is shown in Table IV. Numbers outside and inside bracket in the table is the number of fixed and unfixed bugs for specified type of fix, respectively.

2) Finding Analysis: In this section, we report some findings from our fix statistics and challenges we encountered during our experiment and derive a conclusion based on them.

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

Performance Challenge. After implementing the casting operator into F1X and testing them for a small C program, the repair tool still failed to generate a fix for the integer overflow bug. We found that F1X has an optimization algorithm of test-equivalence in which possible patches are classified during program execution if those patches have the same execution traces and result for the same test cases. However, for integer overflow, it is possible that the patches are not the same even if they have the same execution traces. One patch could cause sign-bit overflow, and another does not. In such a scenario,

the F1X test-equivalence algorithm could make itself failed to find an appropriate patching solution. To resolve this issue, the test-equivalence algorithm needs to be deactivated, and this causes performance overhead during the repair process.

Patch Quality Challenge. Using sanitizers during the compilation of target projects broke down the test suite of our target projects. Many executables always return a sanitizer error message even when we execute them with default options using default test files. This error shows that existing open-source projects are still not developed with proper security in mind. The availability of a test suite is a means 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 way repair tools can assess patch quality. A good example would be to use previous manual fix patterns, as available from OSS-Fuzz.

E. Conclusion

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

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, a combination of single-line fix, and the addition of if-condition. For buffer overflow, the most common single-line fix is changing if-condition and adding if-condition for a multiple-line fix.

We also identified three challenges (technical, performance, and patch quality) after running the repair experiment with F1X. These findings will be helpful 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 (OSDI), 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://lcamcuf.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] W. Weiner, T.V. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in International Conference on Software Engineering (ICSE), 2009.

[11] 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.

[12] 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.

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

[14] W. Weimer, "Patches as better bug reports," in Generative Programming and Component Engineering (GPCE), 2006.

[15] 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.

[16] F.J. Gao, L.Z. Wang, and X.D. Li, "BovInspector: automatic inspection and repair of buffer overflow vulnerabilities," in Automated Software Engineering (ASE), 2016.

[17] 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.

[18] Q. Gao, H.S. Zhang, J. Wang, Y.F. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing QA sites (T)," in Automated Software Engineering (ASE), 2015.

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

[20] 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.

[21] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: program repair with quantitative objectives," in International Conference on Computer Aided Verification, 2016.

[22] S. Mechtaev, J.Y. Yi, and A. Roychoudhury, "Angelix: scalable multiline program patch synthesis via symbolic analysis," in International Conference on Software Engineering, 2016.

[23] S. Mechtaev, S.H. Tan, X. Gao, and A. Roychoudhury, "Test-equivalence analysis for automatic patch generation," in ACM Transactions on Software Engineering and Methodology (TOSEM), 2018.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] A. Shaw, D. Doggett, and M. Hafiz, "Automatically fixing C buffer overflows using program transformations," in Dependable Systems and Networks (DSN), 2014.

[29] A. Di Franco, H. Guo, and C. Rubio-Gonzalez, "A comprehensive study of real-world numerical bug characteristics," in Automated Software Engineering (ASE), 2017.

[30] H. Zhong and Z.D. Su, "An empirical study on real bug fixes," in International Conference on Software Engineering (ICSE), 2015.

[31] 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.

[32] E. Campos and M. Maia, "Common bug-fix patterns: a large-scale observational study," in International Symposium on Empirical Software Engineering and Measurement, 2017.

[33] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019.