Discovering Software Vulnerabilities Based on Fuzz Testing

Yu-Ming Chung, Chihli Hung*
Chung Yuan Christian University, No. 200 Jongpei Rd., Jongli Dist., Taoyuan City, 32023, Taiwan.

*Corresponding author: Tel.: +88632655407; email: chihli@cycu.edu.tw
Manuscript submitted December 25, 2018; February 20, 2019.
doi: 10.17706/jcp.14.2.111-118

Abstract: In the era of the Internet, information security issues are of paramount importance. Software packages invariably have security vulnerabilities. If exploited by malicious hackers, vulnerabilities can cause substantial losses to software corporations and end users. Due to the increase in Advanced Persistent Threat (APT) attacks, vulnerabilities have to be discovered as rapidly as possible. This research focuses on Microsoft Office Word software and proposes the fuzzing vulnerability digging model. In the field of fuzz testing, the traditional approaches consume considerable time and system resources without analyzing file formats. Therefore, the fuzzing vulnerability digging model proposed in this research examines the file format to identify any possible weaknesses. According to the experiments, our proposed model outperforms two benchmarking models, i.e. the FileFuzz tester and MiniFuzz tester, for a fixed period of time. Finally, we present an example which imitates a Shellcode attack carried out via the weaknesses discovered by the proposed model. According to the comparison results, the proposed model has the potential to identify weaknesses in MS Office Word software more effectively and efficiently.

Key words: Fuzz testing, software security, software testing, vulnerability exploiting.

1. Introduction

With the rapid advancement of network technology, there is an increase in network attacks by way of Advanced Persistent Threat (APT). According to research by Trend Micro Inc. [1], 90% of APT is launched by phishing files containing either malicious files or links to malicious sites. Once they are opened, malicious files invade the host through the software vulnerability, and the malware contained within the malicious files is executed on the host effortlessly. The attacks are mainly aimed at the vulnerabilities within the application software executed on Microsoft's platform, such as OFFICE, PDF, IE browser, etc. which are extremely popular. Once the vulnerability is detected by hackers, countless users are affected and can suffer considerable loss. As undiscovered defects are inevitable in software, manufacturers have to locate the vulnerability in the early stages of software development and test the security of the software in order to locate the vulnerability as soon as possible.

There are many types of vulnerability, such as the buffer overflow, SQL injection, Trojan Horse, etc. and a wide array of digging techniques, such as static testing, dynamic testing, binary comparison, fuzz testing, etc. In this research, we focus on the fuzzing vulnerability digging system designed for Microsoft Office Word programs. In general, the error condition in a program may lead to exploitable vulnerabilities. Fuzz testing, proposed by Miller *et al.* [2], is a process whereby invalid data is sent to a target in order to trigger an error condition. This is an effective and widely used technique for finding vulnerabilities in software [3]. In the

field of fuzz testing, most existing models such as FileFuzz [4], MiniFuzz [5], are not designed to analyze file formats, and thus may generate duplicates and produce too many test samples. However, this research processes MS Office Word files without accessing the source codes, in order to identify the vulnerabilities that were not detected in the development process,

The rest of the paper is organized as follows. Section 2 discusses related works including static analysis, dynamic analysis, binary comparison and fuzz testing. Section 3 proposes the fuzzing vulnerability digging model. The experimental results are shown in Section 4. A conclusion and some possible areas for further work are briefly presented in Section 5.

2. Related Work

To strengthen software security and to upgrade information safety, it is crucial and essential to identify and exploit vulnerabilities of software. Popular vulnerability exploiting techniques include static analysis, dynamic analysis, binary comparison, and fuzz testing [6]. Static analysis is a technique in which software testing is performed without executing the targeted program. This method finds potential security vulnerabilities primarily through the analysis of grammar, syntax and semantic usages of the targeted program [7], [8]. Static analysis is of particular use in the software development stage. However it cannot address potential vulnerabilities during the execution stage. Dynamic analysis is a software testing technique, which monitors the dynamic execution state of a running targeted program. This state includes symbolic expressions, the executed path, and taint information [9], [10]. In comparison with static analysis, the results of dynamic analysis are more accurate, as the exploited points are usually caused by weaknesses in the targeted program. However, greater professional knowledge is required in order to trace and explain the cause of the error.

Binary comparison, also known as patch comparison, is designed to dig out the existing vulnerabilities. Software patches are used as software security vulnerability remedies. Software manufacturers do not generally indicate the exact locations and possible causes of vulnerabilities in their security announcements while releasing a patch file. By comparing the differences between the post-patched file and its pre-patched version, it is not difficult to locate the vulnerabilities [11], [12]. However, the technique of binary comparison is not suitable for finding undiscovered vulnerabilities.

Fuzz testing is a software testing technique that continuously sends invalid data to a targeted program so as to cause an error condition, which could lead to exploitable vulnerabilities. The fuzz testing technique tests the targeted program when it is running and does not require access to the source code. The first fuzz testing tool was developed by Miller *et al.* [2] for testing the reliability of UNIX tools. It has been proven useful and successful for discovering vulnerabilities in software which may not be found by other techniques [13], [14]. There are no accepted guidelines in the literature for the fuzz testing technique. Thus there may be an infinite number of input values for testing the targeted software and it is generally measured solely by the results of the specific fuzz testing [13]. For example, in terms of test sample generation, Miller *et al.* [2] and Nethercote and Seward [15] generate test samples by randomly mutating some part of the input data. The technique of random mutation may not be efficient enough when dealing with a specific software whose format can be exploited in advance [16], [3]. Cadar *et al.* [17] use the technique of symbolic input initialization to expand the fuzzing coverage and automatically generate inputs that crash the targeted software. More complete and recent reviews on fuzz testing can be found in the work of Chen *et al.* [3].

3. Approach

The technique for traditional fuzz testing generates test samples randomly, which may not be efficient

enough when dealing with a specific software whose format can be exploited in advance [16], [3]. This research thus analyzes the file format to identify the file's possible weaknesses and then uses a fuzz testing technique to test the weaknesses using generated abnormal files. The proposed model consists of four modules, namely file format inspection and comparison, weakness locating, data generation, and data testing, which are shown in Fig. 1.

3.1. File Format Inspection and Comparison

Unlike traditional fuzz testing approaches such as [2], [4], [5], which do not focus on a specific file format, this research inspects a Microsoft Office Word file format [18]. Only the changed files are tested in order to reduce the time needed by the fuzz testing. We then focus on files that may contain a weakness, so as to improve the efficiency of the traditional fuzz testing system. The Word file format is saved using an iStorage interface, also known as an OLE object file. The iStorage, provided by Microsoft Windows, is designed to save OLE objects in a hierarchical structure comprising directory, subdirectory and file. The iStorage allows various types of data to be stored in different levels of directories. In the iStorage, every 512 bytes of a file are treated as a unit, which is known as a Big Block Data (BBD). All disk sectors are administered by the File Allocation Table (FAT) and the data stored in FAT is called a stream. Every disk sector accommodates 512 bytes of data. If a small volume of data is stored in a disk sector, that disk sector's space is wasted. Therefore, the iStorage stores small volumes of data separately. In the iStorage, the sequence of data, which is stored in the disk sector, is determined by the structure of FAT. A word file header indicates the number of disk sectors, which are allocated by FAT. Therefore, the combination of the data in the disk sectors enables the data stored in FAT to be located.

3.2. Weakness Locating

The main purpose of this module is to locate possible weaknesses in a Microsoft Office Word file. In order to minimize the length of the target Word file, the core Python program of the weakness locating module is designed as in Fig. 2. Once a possible weakness is located, the remainder of the Word file is filtered.

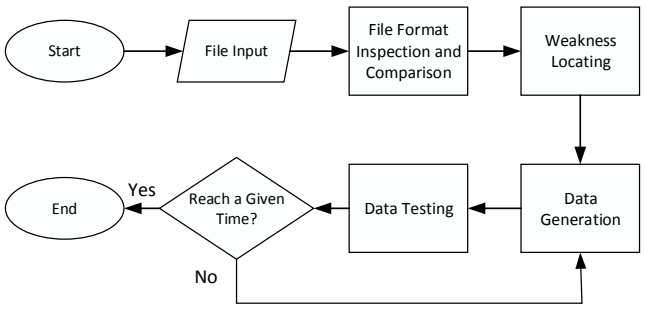


Fig. 1. Conceptual diagram of the proposed model.

```
def loadfat(self, header):
    sect = header[76:512]
    fat = []
    for i in rang(0, len(sect), 4):
        ix = i32(sect, i)
        if ix == -2 or ix == -1: # ix ==0xFFFFFFFEL or ix == 0xFFFFFFFFL
            break
        s = self.getsect(ix)
        fat = fat + map(lambda i, s=s: i32(s, i), range(0, len(s), 4))
    self.fat =fat
```

Fig. 2. The core Python program of the weakness locating module.

3.3. Data Generation

The data generation module inspects the Word file format and uses a mutation-based test data generation approach [19], [20]. It randomly generates test samples by bit flipping and data insertion in order to produce unexpected Word file formats. Two bytes are revised each time. If a version of test samples causes a system crash, that version is reserved. This version simply denotes a weakness and probably signifies a usable vulnerability. All duplicated test samples are filtered by this module as well.

3.4. Data Testing

This module contains two functions, which are sample testing and storing the test results. The sample testing phase uses Microsoft Word to read the test samples one by one and the Python program shown in Fig. 3 monitors the test results based on Windows Debug API. The result storing phase stores detected abnormalities and system information, shown in Fig. 4. This phase then stores test samples that cause errors, and further checks whether or not the abnormalities can be used for purposes of an attack.

4. Experiments

The proposed fuzzing vulnerability digging model is devised to analyze the file format, locate any possible weaknesses in the file, and test the file format using the fuzz testing technique in order to achieve an improvement in the vulnerability digging efficiency.

```
while not done:
    #If we've reached timeout, tell Windows to kill the process.
    if timeout != INFINITE and time.time() >= endtime:
        TerminateProcess(proc.hProcess, 0)

    #Wait for a debug envent. Generate an exception on timeout.
    try:
        e = WaitForDebugEvent(500)
    except WindowsError as err:
        continue

    #The default continue status
    cont_status = DBG_CONTINUE

    if e.dwDebugEventCode == EXCEPTION_DEBUG_EVENT:
        #Get the exception information
        exc = e.u.Exception
        #not handling any exceptions
        cont_status = DBG_EXCEPTION_NOT_HANDLED

        #Clear the exception information if this is the first chance.
        #The second exception should be a real crash.
        if exc.dwFisrtChance:
            exc = None
```

Fig. 3. The core Python program for the sample testing phase in the data testing module.

```
#Check if there is a crash
if result is not None:
    #save a log
    f = open(os.path.join(CONFIG["outcome_dir"], str(seed)+".txt"), "w")
    f.write("INPUT: %s\n" % inFile)
    f.write("Seed: %s\n" % seed)
    f.write("Last seed: %s\n" % lastSeed)
    f.close()

    #save output
    try:
        shutil.copy(outFile, os.path.join(CONFIG["outcome_dir"],os.path.basename(outFile)))
    except Exception as e:
        print "Failed to copy output file:", outFile

    outcome = True
```

Fig. 4. The core Python program for the storing test result phase in the data testing module.

4.1. Description of the Experiments

The experimental environment of this research is implemented with virtual hosting software, i.e. VMware Workstation. A Windows 7 64-bit operating system and Microsoft Office 2013 Word processing software are installed in the virtual host for an automatic software vulnerability digging test. The experiment is divided into two parts. In Part 1, the proposed fuzzing vulnerability digging model is compared with the FileFuzz tester [4] developed by iDefense Corporation and the MiniFuzz tester [5] provided by Microsoft. In Part 2, we show a real-world vulnerability implementation test based on the proposed fuzzing vulnerability digging model.

4.2. Results and Analysis

1) Part 1

Like the experiments in Ouyang *et al.* [21], which compare their model with other approaches for a fixed time period, we also compare the proposed fuzzing vulnerability digging model with the FileFuzz tester and MiniFuzz tester based on the digging efficiency for a fixed time, i.e. 24 hours. The test samples are from VirusTotal [22], which provides the services developed by Hispasec Sistemas Security Laboratory. In this research, we use three different Word formats, i.e. DOC, DOCX and RTF, and show the comparison results in Tables 1-3. An abnormal file is a test sample, which causes errors. An effective weakness is a real vulnerability. In most cases, vulnerabilities are produced by abnormal files. According to the results in Tables 1-3 when the same amount of time is given, the proposed fuzzing vulnerability digging model digs

out two, two, four weakness from 103, 142, 174 abnormal files for DOC, DOCX and RTF formats respectively and achieves much higher effective digging ratios than FileFuzz and MiniFuzz testers.

In terms of the FileFuzz tester, due to the lack of analyzing and filtering mechanisms from those abnormal test samples that have been generated, too many abnormal test samples are produced. As a result, the test staff have to spend a lot of time analyzing the large volume of abnormal test samples. On the other hand, the MiniFuzz tester is unable to analyze the correlation between the data and the program. As a result, MiniFuzz digs out slightly fewer effective weaknesses within 24 hours than our proposed model.

Table 1. Comparing the Results Obtained from DOC Format Test

Testing System	Effective Weakness	Abnormal Files	Ratio
MiniFuzz	1	573	0.17%
FileFuzz	1	627	0.16%
This research	2	103	1.97%

2) Part 2

In order to show the hazard imposed by the proposed fuzzing vulnerability digging model, a vulnerability implementation test is executed. According to the Microsoft RTF format specification, listoverridecount has to be 0, 1 or 9 as shown in Fig. 5. When the digit is modified to other values and the amount controlled by lfolevel exceeds the amount allowed by listoverridecount, a buffer overflow occurs.

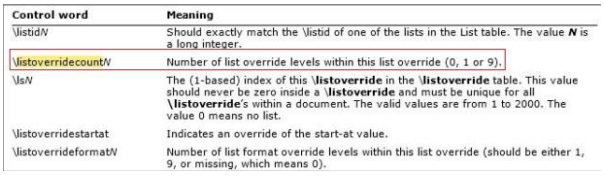
As indicated by the test results, when the value of listoverridecount is modified to 25, a buffer overflow occurs, precisely covering the location of memory to be used (Fig. 6).

Table 2. Comparing the Results Obtained from DOCX Format Test

Testing System	Effective Weakness	Abnormal Files	Ratio
MiniFuzz	1	751	0.13%
FileFuzz	0	892	0.00%
This research	2	142	1.41%

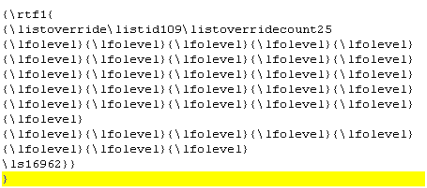
Table 3. Comparing the Results Obtained from RTF Format Test

Testing System	Effective Weakness	Abnormal Files	Ratio
MiniFuzz	2	669	0.30%
FileFuzz	0	992	0.00%
This research	4	174	2.30%



Control word	Meaning
\listidN	Should exactly match the \listid of one of the lists in the List table. The value ***N*** is a long integer.
\listoverridecountN	Number of list override levels within this list override (0, 1 or 9).
\lsN	The (1-based) index of this **\listoverride** in the **\listoverride** table. This value should never be zero inside a **\listoverride** and must be unique for all **\listoverride**'s within a document. The valid values are from 1 to 2000. The value 0 means no list.
\listoverridestartat	Indicates an override of the start-at value.
\listoverrideformatN	Number of list format override levels within this list override (should be either 1, 9, or missing, which means 0).

Fig. 5. The listoverride format.



```
{\rtf1{
{\listoverride\listid109\listoverridecount25
{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}
{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}
{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}
{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}
{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}
{\lfolevel}
{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}{\lfolevel}
{\lfolevel}{\lfolevel}{\lfolevel}
\ls16962}}
}
```

Fig. 6. Test file format.

We then use Ollydbg [23] to observe the operational state of the test file in the memory. When listoverridecount is modified to 25 and the program proceeds to the position 0x654918A2 of the memory, the value of (EBP-0x2E04) becomes 0x19. After that, the program proceeds to the position 0x6549BAFF of the memory and writes 0x19*8 amount of data in the stack (Fig. 7). If 0x19 is loaded as a value, the program proceeds to 0x654A31DF and an error occurs (Fig. 8). When the program proceeds to the position 0x654A3202 of the memory, a buffer overflow is caused by the increasing amount in lfolevel (Fig. 9). Normally, data 0x66477A00 and 0x66477A50 are stored in the position 0x02E617E0 of the memory (Fig.

10). As soon as the program triggers a buffer overflow, the data that is stored in the memory changes to 0x0000003C and 0x034f0150 (Fig. 11). As shown in Fig. 12, as soon as the data that is stored in the memory changes, the program reads the position 0x034f0154 of the memory and thus obtains 0x275A48E8, which is a starting point of Shellcode.

Fig. 7. State of address 0x654918A2.

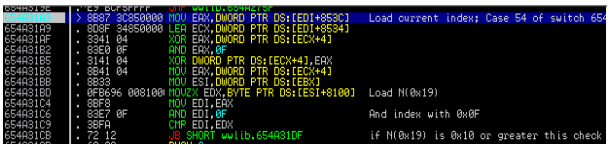


Fig. 8. Checking the value of listoverridecount.

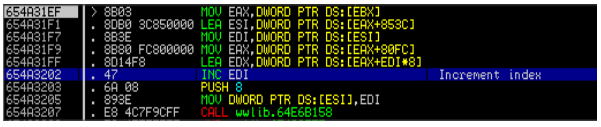


Fig. 9. State of address 0x654A3202.

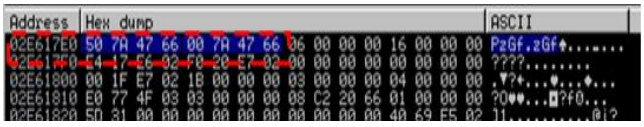


Fig. 10. The state before a buffer overflow at address 0x02E617E0.

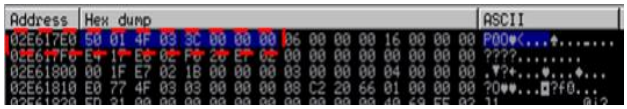


Fig. 11. The state after a buffer overflow at address 0x02E617E0.

Fig. 12. The state after a buffer overflow.

When a Word file that contains a vulnerability is executed, a buffer overflow occurs and a Shellcode, compiled by this research, is executed accordingly. The Shellcode is designed to release the backdoor program set up by a malicious file. As shown in Fig. 13, when the Shellcode is placed in the file that is likely to trigger a buffer overflow, a vulnerability will be triggered and a backdoor program will be executed successfully once the file is opened (In this research, a computer program, i.e. Calculator, is used instead of a backdoor program).

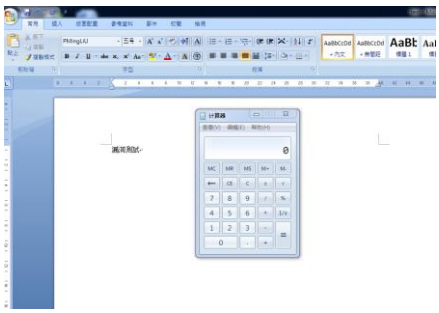


Fig. 13. Using a vulnerability.

5. Conclusion and Further Work

All security-dependent software has to undergo comprehensive testing to ensure it meets the required level of security. With the rapid development of mobile computing and big data techniques, system reliability, software quality and information security have become more important than ever. This research has examined vulnerability-related issues and developed an automatic software vulnerability digging

model. The fuzzing vulnerability digging model implemented by this research is compared with FileFuzz tester [4] and MiniFuzz tester [5]. According to the comparison results, the proposed model has the potential to identify MS Office Word's software weaknesses more effectively and efficiently. Finally, we present an example which imitates a Shellcode attack carried out through the weaknesses discovered by the proposed model.

As network attacks by Advanced Persistent Threat (APT) are gradually increasing, one important issue in the field of fuzz testing is to dig out as many unique weaknesses as possible within a reasonable given time period. An optimal test sample rather than a random test sample generation should be further studied. An optimal scheduling strategy for the test order instead of the predefined scheduling strategy is another area of possible further work. On the other hand, machine learning and deep learning have shown extraordinary results in various fields [24]. Combining machine learning and deep learning with fuzz testing techniques may be an another exciting direction for possible future work.

Acknowledgments

This work was supported in part by the Ministry of Science and Technology of Taiwan under Grant MOST 106-2410-H-033-014-MY2.

References

[1] TrendLabs. (2012). 91% of APT attacks start with a spear-phishing email. *Infosequrity Magazine,* Retrieved March 26, 2016, from http://www.infosecurity-magazine.com/news/91-of-apt-attacks-start-with-a-spear-phishing

[2] Miller, B. P., Fredriksen, L., & So, B. An empirical study of the reliability of UNIX utilities. *Communications of the ACM, 33(12),* 32-44.

[3] Chen, C., Cui, B., Ma, J., Wu, R., & Guo, J. (2018). A systematic review of fuzzing techniques. *Computers & Security, 75,* 118-137.

[4] FileFuzz. Retrieved July 25, 2017, from http://www.securiteam.com/tools/5PP051FGUE.html

[5] MiniFuzz. Retrieved July 25, 2017, from https://msdn.microsoft.com/en-us/gg675011

[6] Zhang, X., & Li, Z. J. (2016). Survey of fuzz testing technology. *Computer Science, 43(5),* 1-8.

[7] Kinder, J., Zuleger, F., & Veith, H. (2009). An abstract interpretation-based framework for control flow reconstruction from binaries. *Proceedings of International Workshop on Verification, Model Checking, and Abstract Interpretation* (pp. 214-228).

[8] Sparks, S., Embleton, S., Cunningham, R., & Zou, C. (2007). Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. *Proceedings of the 23rd Computer Security Applications Conference* (pp. 477-486).

[9] Cha, S. K., Avgerinos, T., Rebert A., & Brumley, D. (2012). Unleashing mayhem on binary code. *Proceedings of IEEE Symposium on Security and Privacy* (pp. 380-394).

[10] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., & Bos, H. (2017). VUzzer: Application-aware evolutionary fuzzing. *Proceedings of the Network and Distributed System Security Symposium.*

[11] Marinescu, P. D., & Cadar, C. (2013). KATCH: High-coverage testing of software patches. *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering* (pp. 235-245).

[12] Sun, H., Liu, S., Xiao, D., & Xiao, R. (2017). Applying binary patch comparison to Cisco IOS. *Proceedings of the 2017 VI International Conference on Network, Communication and Computing* (pp. 38-42).

[13] Sutton, M., Greene, A., & Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery.* New York: Pearson Education.

[14] Kang, J., & Park, J. H. (2017). A secure-coding and vulnerability check system based on smart-fuzzing

and exploit. *Neurocomputing, 256,* 23-34.

[15] Nethercote, N., & Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 89-100).

[16] Bastani, O., Sharma, R., Aiken, A., & Liang, P. (2017). Synthesizing program input grammars. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 95-110).

[17] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., & Engler, D. R. (2006). EXE: Automatically generating inputs of death. *Proceedings of the 13th ACM Conference on Computer and Communications Security* (pp. 322-335).

[18] Microsoft. understanding the word. *Binary File Format.* Retrieved July 1, 2017 from https://msdn.microsoft.com/en-us/library/office/gg615596(v=office.14).aspx

[19] Cohen, M. B., Snyder, J., & Rothermel, G. (2006). Testing across configurations: Implications for combinatorial testing. *ACM Sigsoft Software Engineering Notes, 31(6),* 1-9.

[20] Lanet, J., Bouder, H. L., Benattou, M., & Legay, A. (2017). When time meets test. *International Journal of Information Security,* 1-15.

[21] Ouyang, Y. J., Wei, Q., Wang, Q. X., & Yin, Z. X. (2015). Intelligent fuzzing based on exception distribution steering. *Journal of Electronics & Information Technology, 37(1),* 143-149.

[22] Virustotal. Retrieved July 25, 2017, from https://www.virustotal.com

[23] Ollydbg. Retrieved July 25, 2017, from http://www.ollydbg.de

[24] Zhang, Q., Yang, L. T., Chen, Z., & Li, P. (2018). A survey on deep learning for big data. *Information Fusion, 42,* 146-157.

Yu-Ming Chung obtained a master at Department of Information Management of Chung Yuan Christian University, Taiwan in 2016. He is an independent cyber security consultant in Taiwan. He has worked in the fields of reverse engineering and fuzzing (for Windows vulnerabilities) for more than 10 years. His current research interests are in malicious code and packets analysis.

Chihli Hung is a professor at the Department of Information Management of Chung Yuan Christian University, Taiwan. He obtained a Ph.D at School of Computing and Technology from the University of Sunderland, UK in 2004. His current research interests are in information security management, text mining, machine learning, data mining and intelligent systems.