File Fuzzing System using Field Information and Fault-Injection Rule

Dong Hyun Lee1), Su Yong Kim2), Dae Sik Choi3), Hyung Geun Oh4)

Abstract

File fuzzing(or file fuzz testing) is a software testing technique that checks the response of a target program against abnormal file inputs. It is simply random testing but powerful. Especially, it is worth as security testing. However, file fuzzing is inefficient in the sense that it takes too much time, nearly endless, and so on. For even one input file, it takes several seconds to execute. Besides, most input files that are generated randomly are invalid.

We propose the advanced file fuzzing system applying field information and fault-injection rule. For a file, field information represents the starting position, size, unique name, and valid data type of each field. And fault-injection rule is the formalized expression to describe generating and injecting a fault. These enable us to make effective input files and to distribute fuzzing works to several machines. In addition, our system provides the independent random fuzzing.

Keywords : File fuzzing, Software testing technique, Fault-Injection, Security Test

1. Introduction

Fuzzing is a software testing technique that provides random data to the inputs of a target program and finds bugs or vulnerabilities[1]. It has the characteristics of black-box testing (source codes of a target program are usually unknown), execution-based testing (a target program is run and monitored), and security testing (it checks boundary condition or outer input). It is also cost-effective. An implementation is simple, however results make us to find more defects than expected[2-3]. It has gained in popularity over the last few years so that vendors have adopted fuzzing as a part of their software developmen[4-5].

Received(October 04, 2008), Review request(October 05, 2008), Review Result(1st:October 25, 2008, 2nd:November 14, 2008) Accepted(December 31, 2008)

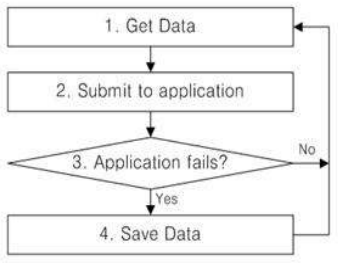
[1](Corresponding Author) Researcher, Attached Institute of ETRI, 138 Gajeongno, Yuseong-gu, Daejeon, 305-700, Korea.
email: donghyun2u@ensec.re.kr

[2]Researcher, Attached Institute of ETRI, 138 Gajeongno, Yuseong-gu, Daejeon, 305-700, Korea.
email: sweetlie@ensec.re.kr

[3]Researcher, Attached Institute of ETRI, 138 Gajeongno, Yuseong-gu, Daejeon, 305-700, Korea.
email: dschoi@ensec.re.kr

[4]Researcher, Attached Institute of ETRI, 138 Gajeongno, Yuseong-gu, Daejeon, 305-700, Korea.
email: hgoh@ensec.re.kr

Fuzzing methodology is based on fault-injection (see Figure 1,[2]). Iteration starts from data creation. The data should be semi-valid, it means that correct enough to keep parsers from immediately dismissing it, but at the same time broken enough to cause problems so that parsing done on this input will fail[2][6]. Then the data is submitted to a target application, and failing is monitored. If failing, such as crash, is detected, the fuzzing data will be saved. After that, next iteration starts again.



[Fig. 1] Fuzzing flowchart

Fuzzing can be classified with input data. In most systems, the majority of input comes from files, APIs, user interfaces, network interfaces, database entries, and command line arguments[2]. In this paper, we focus on file fuzzing which uses files as input data.

We first provide an overview of file fuzzing and present its point at issue in Section 2. Then, in Section 3, we propose our solution, the advance file fuzzing system using field information and fault-injection rule,and show how it works. Section 4 summarizes our improvements, and Section 5 comes to a conclusion.

2. File fuzzing

File fuzzing tests the response of a target program against abnormal input files, so it aims at file parser, input validation controller, and exception handler[7]. File fuzzing consists of three phases, file making, file executing, and exception monitoring.

The first phase is file making, which creates abnormal input files. There are two types of file making, either generation or mutation[2]. Generation is making new files based on the file format already known, and mutation is making fault-injected files from a valid sample file. In common, mutation is more useful and widely used because generation is quite a harder work and only used when the format of input filesis known. So in this paper, we will treat file making as mutation. The next phase is file executing, which executes a target program and opens input files one by one. The last phase is exception monitoring. Monitored objects are crash and hang in principal, but any abnormal actions can be. Through exception monitoring, we can find not only program bugs but also critical security vulnerabilities like buffer overflow.

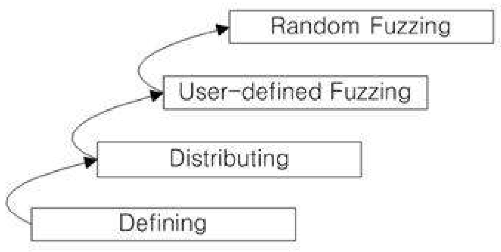
File making is the most important part in file fuzzing system. Whether we can find program's defects or not, it depends on input files. And invalid input files make waste of time. So how effective input files we make is the key issue.

Effective file making is creating semi-valid input files and reducing invalid test. In order to do this, it needs the fault set that induces failing, and also needs the means that injects a fault into an appropriate place of a file. However, the ideal fault set doesn't exist, so we should use various faults case by case.

In fact, there is no smart solution. FileFuzz, SPIKEfile, and notSPIKEfile[7] allow restrictive fault-injection. Holodeck[8] has random and find-and-replace fault-injection. And most other file fuzzing tools are not better[9-11].

3. Advanced file fuzzing system

We propose more elaborate and efficient file fuzzing system. It takes four steps (see Figure 2), which are 'defining methods of file making', 'distributing fuzzing works over usable computers', 'doing fuzzing that is defined at the first step', and 'doing random fuzzing to find unpredictable defects for a long time'. All steps are able to be fully-automated except that the first step can be semi-automated. More details are explained later in this section.



[Fig. 2] Four steps of advanced file fuzzing

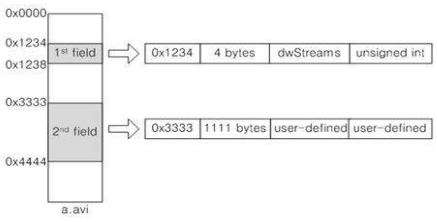
3.1. Field information

File is an array of binary and can be composed of fields that have a meaning. In common, fields are defined by file format. However, user-defined fields can be also possible. User-defined fields are useful in case of unknown file format especially. Fields can be overlapped or modified, and even whole file can become one field.

Field information represents the starting position, size, unique name, and valid data type of each field for a sample file. It is used for the file making. The set of field information for a sample file 's', FI(s), is defined below.

$$FI(s)=\{FI_i(s)\mid FI_i(s)=(pos_i(s),\ size_i(s),\ id_i(s),\ typ$$

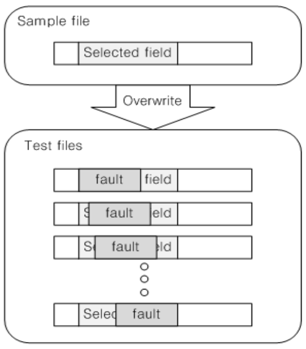
FIi(s) means field information of the i-th field for a sample file ‘s’.The posi(s), sizei(s), idi(s), and typei(s) represent the starting position, size, unique name, and valid data type of the i-th field for a sample file ‘s’respectively.



[Fig. 3] An example of field information

Here is one example (see Figure 3). It shows the FI(a.avi) that has two fields. The first field keeps the avi file format. The pos1(a.avi) is ‘0x1234’, size1(a.avi) is ‘4 bytes’ from 0x1238-0x1234, id1(a.avi) is ‘dwStreams’, that is used for the user readability, and type1(a.avi) is ‘4 bytes unsigned integer’ from avi file format. The second field is the user-defined field. The pos2(a.avi) is ‘0x3333’, size2(a.avi) is ‘1111 bytes’ from 0x4444-0x3333, id2(a.avi) is ‘user-defined’, and type2(a.avi) is ‘user-defined’. So FI(a.avi) is {(0x1234, 4 bytes, dwStreams, 4 bytes unsigned integer), (0x3333, 1111 bytes, user-defined id, user-defined type)}.

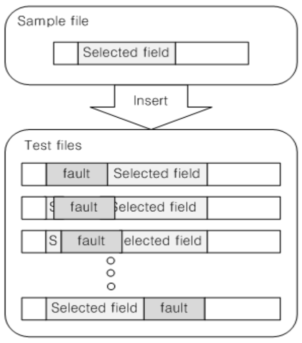
3.2. Fault-injection rule



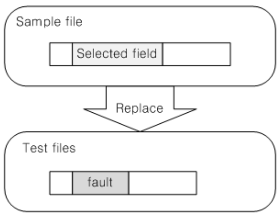
[Fig. 4] Injection-type : Overwrite

Fault-injection is an effective method to make semi-valid input files. It injects a fault into valid sample file and creates modified files. Fault-injection rule is the formalized expression to describe generating and injecting a fault. It has 4 attributes, injection-type, injection-option, fault-type, and fault-option.

Injection-type is how to inject a fault into the selected field. Injection-type is one among ‘overwrite’, ‘insert’, and ‘replace’. The ‘overwrite’/‘insert’is the rule that overwrites/inserts a fault on/into every possible position of the selected field by the slide (see Figure 4, Figure 5), and the ‘replace’ is the rule that replaces the selected field with a fault (see Figure 6). Injection-option has two parameters. One is the step-size of the slide for ‘overwrite’ or ‘insert’. The other is the swap-flag that is the on/off control about byte swapping of a fault.



[Fig. 5] Injection-type : Insert



[Fig. 6] Injection-type : Replace

Fault-type is one among ‘set’, ‘sequential’, and ‘random’. The ‘set’ means a set of faults, and the ‘sequential’/‘random’ means a series of faults changed sequentially/randomly. According to the fault-type, fault-option is different. Fault-option of the ‘set’ has the number of faults and the fault set. Fault-option of the both ‘sequential’ and ‘random’ has the minimum fault-value, maximum fault-value, step-size of fault-value, minimum fault-length, maximum fault-length, and step-size of fault-length. A fault is a binary array so it is decided by fault-value and fault-length. If fault-value is longer than fault-length, it will be truncated. If fault-value is shorter than fault-length, it will be repeated until it has the equal length. In addition, fault-option of the ‘random’ has one more parameter, the number of faults which will be created.

The set of fault-injection rule, FR, is defined below. FRi is the i-th fault-injection rule. IT means injection-type and IO meansinjection-option. FT means fault-type and FO means fault-option. The stepsizex is the step-size of x. The minx and maxx are the minimum of x and the maximum of x respectively.

$$FR = \{FR_i \mid FR_i = (a_i, b_i, c_i, d_i),\ a_i \in IT,\ b_i \in IO,\ c_i \in FT,\ d_i \in FO\}$$

$$IT = \{overwrite,\ insert,\ replace\}$$

$$IO = \{(x, y) \mid x = stepsize_{slide},\ y \in \{on, off\}\}$$

$$FT = \{set,\ sequential,\ random\}$$

$$FO = \begin{cases} \{(n, X) \mid n = \#\text{ of faults},\ X = \{fault_j \mid 1 \le j \le n\}\} & ,\ if\ set \\ \{(x, y) \mid x = \lambda_{fault\ value},\ y = \lambda_{fault\ length}\} & ,\ if\ sequential \\ \{(x, y, n) \mid x = \lambda_{fault\ value},\ y = \lambda_{fault\ length},\ n = \#\text{ of faults}\} & ,\ if\ random \end{cases}$$

$$\lambda_x = (min_x, max_x, stepsize_x)$$

Here are some examples.FR1 = (replace, (0, off), set, (4, {0x7FFFFFFF, 0x11111111, 0xFFFFFFFF, 0x00000000})) means the fault-injection that replace the selected field with 0x7FFFFFFF, 0x11111111, 0xFFFFFFFF, and 0x00000000, so totally 4 faulted files will be created.

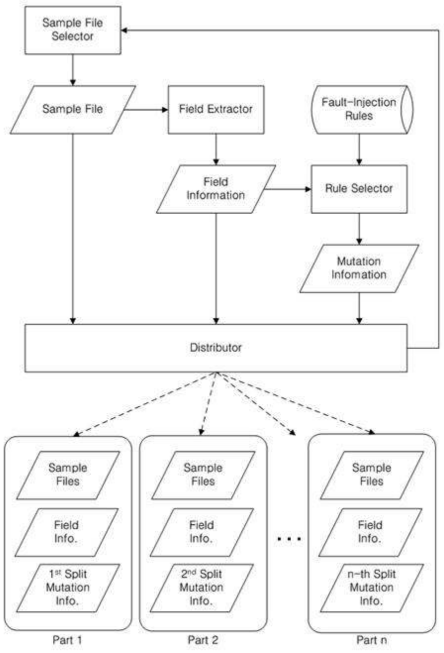
FR2 = (overwrite, (1 byte, off), sequential, ((0x4142, 0x4142, 0), (2 bytes, 8 bytes, 2 bytes))) means the fault-injection that overwrite the 0x4142, 0x41424142, 0x414241424142, and 0x4142414241424142 on every 1 byte sliding positionsof the selected field. If the length of the selected field is 10 bytes, totally 24(=9+7+5+3) faultedfiles will be created.

And FR3 = (insert, (1 byte, off), random, ((0x0000, 0xFFFF, 0xF), (2 bytes, 4 bytes, 2 bytes), 100)) means the fault-injection that insert faults randomly selected from {0x0000, 0x000F, 0x001E, 0x002D, ⋯ , 0xFFF0, 0xFFFF}∪{0x00000000, 0x000F000F, 0x001E001E, 0x002D002D, ⋯ , 0xFFF0FFF0, 0xFFFFFFFF} into every 1 byte sliding positions of the selected field. If the length of the selected field is 10 bytes, totally 1100(=100*11) faulted files will be created.

3.3. Defining &Distributing

The first step of the advanced file fuzzing system is defining of file making. We formulate this by means of field information and fault-injection rule.

It begins from sample file selector (see Figure 7), whose role is that select one sample file from pre-collected sample files one by one. Then field extractor creates field information from this selected sample file. If the sample file format is known, field extractor can be automated. And also user is able to modify and tocreate field information. Next, rule selector selects fault-injection rules from pre-defined rules and creates mutation information that represents which fault-injection rules are applied for each data types. Fault-injection rules and mutation information are also able to be modified or user-defined. The set of mutation information for a sample file ‘s’, MI(s), is defined below.



[Fig. 7] Preprocess for user-defined fuzzing

$$MI(s) = \{(type, R) \mid (pos, size, id, type) \in FI(s),\ R \subset FR\}$$

One example of mutation information is (4 bytes unsigned integer, {FR1, FR3}). It means that apply the 1st and 3rd fault-injection rules to the field when the valid data type of the field is 4 bytes unsigned integer. Selected fault-injection rules are should be agreed with valid data type. If FR1 is the fault set for 4 bytes unsigned integer overflow and FR3 is the sequential faults for 4 bytes unsigned integer, we can expect that fuzzing is very effective. This is what we will do.

After we get mutation information, the iteration fornext sample file begins again at sample file selector. If all sample files are done, next step is distributing fuzzing works.

The role of distributor is distributing sample files, field information, and split mutation information over usable computers (see Figure 7). Split mutationinformation is different with each other. Through these, each computer works a part of fuzzing. When k computers are usable, then the i-th split of MI(s), MI(i)(s), is same as MI(s) except fault-injection rules. Each fault-injection rule is transformed, so that faults are divided. We defined it as below.

$$f^{(i)} : MI(s) \rightarrow MI^{(i)}(s), \qquad (type, g^{(i)}(R)) = f^{(i)}((type, R))$$

$$g^{(i)} : FR \rightarrow FR^{(i)}, \qquad (z_1, z_2, z_3, z_4^{(i)}) = g^{(i)}((z_1, z_2, z_3, z_4))$$

$$\begin{aligned}
&case\ 1)\ if\ z_3 = set\ and\ z_4 = (n, X) = (n, \{x_1, x_2, x_3, \dots, x_n\}) \\
&\qquad then\ z_4^{(i)} = (\alpha_{n,k}^{(i)}, X^{(i)}) \\
&\qquad , X^{(i)} = \{x_p \mid (i-1)\alpha_{n,k}^{(1)} < p \le (i-1)\alpha_{n,k}^{(1)} + \alpha_{n,k}^{(i)}\}
\end{aligned}$$

$$\begin{aligned}
&case\ 2)\ if\ z_3 = sequential\ and\ z_4 = ((x_1, x_2, x_3), (y_1, y_2, y_3)), \\
&\qquad then\ z_4^{(i)} = either\ ((x_1^{(i)}, x_2^{(i)}, x_3), (y_1, y_2, y_3))\ or \\
&\qquad\qquad\qquad ((x_1, x_2, x_3), (y_1^{(i)}, y_2^{(i)}, y_3)) \\
&\qquad , x_1^{(i)} = [(i-1)\alpha_{m,k}^{(1)} / x_3 + \beta^{(i)}]x_3 + x_1 \\
&\qquad , x_2^{(i)} = \alpha_{m,k}^{(i)} + (i-1)\alpha_{m,k}^{(1)} + x_1 \\
&\qquad , y_1^{(i)} = [(i-1)\alpha_{n,k}^{(1)} / y_3 + \beta^{(i)}]y_3 + y_1 \\
&\qquad , y_2^{(i)} = \alpha_{n,k}^{(i)} + (i-1)\alpha_{n,k}^{(1)} + y_1 \\
&\qquad , m = x_2 - x_1 \\
&\qquad , n = y_2 - y_1
\end{aligned}$$

$$\begin{aligned}
&case\ 3)\ if\ z_3 = random\ and\ z_4 = (x, y, n), \\
&\qquad then\ z_4^{(i)} = (x, y, \alpha_{n,k}^{(i)})
\end{aligned}$$

$$\alpha_{n,k}^{(i)} = \begin{cases} [n/k] & , if\ i < k \\ n - [n/k](k-1) & , if\ i = k \end{cases} \qquad i, n, k \in \aleph$$

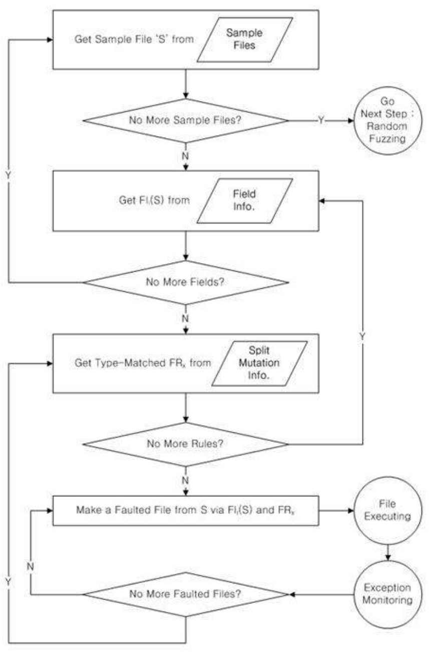
$$\beta^{(i)} = \begin{cases} 0 & , if\ i = 1 \\ 1 & , if\ i \neq 1 \end{cases} \qquad i \in \aleph$$

The f(i) and g(i) are the transformsfor splitting. MI(s) is transformed to MI(i)(s) by f(i), and each fault-injection rule is transformed by g(i). So the fault set having n elements is divided into k fault sets having n/k elements.The sequential rule whose range is n divided into k sequential rule whose range is n/k. And the random rule to generate n faults is divided into k random rule to generate n/k faults.

3.4. User-defined fuzzing

The mainidea of user-defined fuzzing is injecting the faults we want into the fields we want. For this, file making is defined by sample files, field information and mutation information.

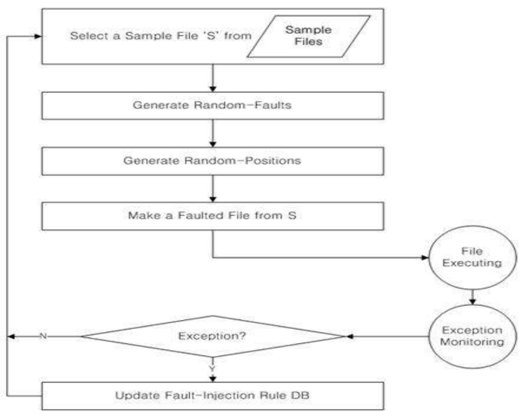
Figure 8 is the flowchart of user-defined fuzzing. File making is for that we only concern. We get a sample file ‘S’ from sample files, a FIi(S) from field information, and the type-matched FRx from split mutation information. FRxshould be matched with the invalid data type of FIi(S). Then we inject the fault defined by FRx into the field defined by FIi(S), so that faulted files are made one by one. For each faulted file, file executing and exception monitoring are performed.



[Fig. 8] Flowchart of user-defined fuzzing

3.5. Random fuzzing

Random fuzzing has the method of file making that injects random faults into random positions of sample files. It works for a long time.In spite of this, random fuzzing is important because it has the potential to find unpredictable defects. So user-defined fuzzing and random fuzzing complement mutually.


Select a Sample File 'S' from
Sample Files
Generate Random-Faults
Generate Random-Positions
Make a Faulted File from S
File Executing
Exception Monitoring
Exception?
N
Y
Update Fault-Injection Rule DB

[Fig. 9] Flowchart of random fuzzing

Figure 9 is the flowchart of random fuzzing. Iteration begins from selecting a sample file from sample files. For fault-injection, faults and positions are generated randomly at each cycle. The number of random faults, injected once, can be more than one. And the number of positions is determined by the number of random faults. Then we make a faulted file, and file executing and exception monitoring are performed. Iteration continues until user stops.

It has the feedback so that the random faults that raise exceptions add to fault-injection rule database. These are used for user-defined fuzzing next time.

4. Improvements

In this chapter, we enumerate improvements of the proposed file fuzzing system. The first is that we formalize field information and fault-injection rule. It enables us to manage file making and load distribution. So our system makes it possible to define file making at a user's disposition and to distribute fuzzing works to several machines.

The second is that our system provides effective and efficient file fuzzing. Whole systems areautomated except the file making. However, we extract fields automatically in case of known file formatand inject faults that based on the valid data type of each field. Type-matched fault-injection gives more chance to make semi-valid input files and reduce invalid tests. And we separate user-defined fuzzing and random fuzzing so that users focus on user-defined fuzzing. Also the feedbacks from random fuzzing help to update fault-injection database.

The third is that our system is general-purpose as file fuzzing frameworks. We allow users to edit field informationand fault-injection rule so that users are able to define suspicious fields and to test with user-defined faults. We provide the means that inject faults user want into fields user want. And we also provide the powerful random fuzzing.

The forth is that we has load distribution system. Because file fuzzing takes too much time and resources, load distribution is meaningful. Further more, through this, we are able to separate controlling works and fuzzing works. In common, the computer that is doing file fuzzing is under busy CPU and hard to handle via mouse or keyboard. Independent controlling helps exception monitoring and user convenience.

5. Conclusion

This paper has examined the characteristics of file fuzzing and proposed advanced file fuzzing system applying field information and fault-injection rule. It has described the definition of filed information and fault-injection rule. And it has explained the four steps of the proposed system, defining, distributing, user-defined fuzzing, and random fuzzing. We have implemented our system and used it in practical. Our system provides more effective and efficient file fuzzing environment. In the future, we plan to address automated exception analysis and code coverage analysis.

6. References

[1] Definition of fuzzing, http://en.wikipedia.org/wiki/Fuzzing.

[2] Peter Oehlert, "Violating Assumptions with Fuzzing", *IEEE Security & Privacy*, pp. 58-62, March/April 2005.

[3] Jared DeMott, “The Evolving Art of Fuzzing”, *DEFCON 14*, Las Vegas, 2006.

[4] Joe Moore, Mark Rowe, "Tools and techniques to automate the discovery of zero day vulnerabilities", *19th Annual FIRST Conference*, June 2007.

[5] Kenneth R. van Wyk, "Integrating Security Tools Into a Secure Software Development Process", *19th Annual FIRST Conference*, June 2007.

[6] Ilja van Sprundel, "Fuzzing: Breaking software in an automated fashion", http://events.ccc.de/congress/2005/fahrplan/attachments/582-paper_fuzzing.pdf, Dec. 2005.

[7] M. Sutten and A. Greene, "The Art of File Format Fuzzing", *BlackHat Conference*, Las Vegas, NV, July 2005.

[8] Holodeck, http://www.securityinnovation.com

[9] Gwar, http://gruba.blogspot.com/2006/11/file-fuzzer.html

[10] Mangle, http://www.digitaldwarf.be/

[11] FuzzyFiles, http://reedarvin.thearvins.com/

Authors

Author did not want to show his photo

Dong Hyun Lee

Author did not want to show his bio-data.

Author did not want to show his photo

Su Yong Kim

Author did not want to show his bio-data.

Author did not want to show his photo

Dae Sik Choi
Author did not want to show his bio-data.

Author did not want to show his photo

Hyung Geun Oh
Author did not want to show his bio-data.