Efficient Model-based Fuzz Testing Using Higher-order Attribute Grammars

Fan Pan, Ying Hou, Zheng Hong, Lifa Wu, Haiguang Lai
Institute of Command Automaton, PLA University of science and technology, Nanjing, Jiangsu, China
Email: dynamozhao@163.com, yinghou26@gmail.com, hongzhengjs@139.com, wulifa@vip.163.com, lite@263.net

***Abstract*—Format specifications of data input are critical to model-based fuzz testing. Present methods cannot describe the format accurately, which leads to high redundancy in testing practices. In order to improve testing efficiency, we propose a grammar-driven approach to fuzz testing. Firstly, we build a formal model of data format using higher-order attribute grammars, and construct syntax tree on the basis of data samples. Secondly, all nodes in the syntax tree are traversed and mutated to generate test cases according to the attribute rules. Experimental results show that the proposed approach can reduce invalid and redundant test cases, and discover potential vulnerabilities of software implementations effectively.**

***Index Terms*—Model-based fuzz testing, Higher-order attribute grammars, Syntax analysis tree, Test case generation**

I. INTRODUCTION

Fuzz testing or fuzzing is a kind of software testing technique, which involves providing random or malicious data as input to a software implementation [1]. Due to the high benefit-to-cost ratio, it has emerged as a key approach for discovering software vulnerabilities over the past few years.

The key issue of fuzz testing is to generate semi-valid test cases that can bypass checks and verifications [2]. The unexpected input is usually rejected by the target software at a very early stage. Failure to produce well-formed test cases will severely lower the chances to find any errors. Model-based fuzz testing is a form of fuzz testing, and a formal model of the input is used to generate well-formed inputs. Model-based fuzz testing can not only improve efficiency of fuzz testing, but also measure the comprehensiveness of testing and enhance the level of test automation.

There are many previous works about model-based fuzz testing [3-11]. Although their test practices have found a large number of vulnerabilities, these approaches have certain limitations: 1) the model of format specification cannot describe the context-sensitive constraints; 2) the shared fields in different type of format lead to high redundancy in test practices; 3) format specification is not always consistent with the implementation of target software, and the resulting test cases may be invalid.

Our work was motivated by these limitations. In this paper, we propose a grammar-driven approach to further improve the quality and efficiency of fuzz testing. The major idea is to build a formal model of input format using Higher-order Attribute Grammars (HAG), and apply it to automatically guide test case generation. The advantages of our approach are listed as follows: 1) We describe the context-sensitive constraints as attribute rules in HAG for reducing the production of invalid test cases; 2) By recording the tested model elements during testing process, we are able to lower the redundancy of test practices. To investigate the feasibility of our approach, we developed a prototype system based on the Peach platform [6], and tested it on a number of software implementations.

The rest of this paper is organized as follows. Related work is introduced in section 2. Section 3 analyzes the properties of input data and proposes a formal model of format specification. In section 4, the process of grammar-driven fuzz testing is given. We detail the implementation of the assessment platform and the results obtained in testing experiments. Section 6 concludes the paper and highlights future works for our research.

II. RELATED WORK

Fuzz testing was first explored by Barton Miller as an automatic black-box testing approach [12]. The original method has shown that it can successfully discover software vulnerabilities. However, without any knowledge about the implementation, most of the test cases are usually rejected. Combined with other testing technologies, intelligent fuzzing has been proposed to improve efficiency of test practices. Intelligent fuzzing is usually classified into white box fuzzing, evolution fuzzing and model-based fuzzing.

White box fuzzing [13-15] takes advantage of symbolic execution and constraint solving. The approach symbolically evaluates the execution of software implementation and collects constraints on the input, then generates concrete test cases that satisfy the set of constrains. However, the white box fuzzing is far from effective as the number of feasible control paths may be infinite and because of the imprecision of symbolic execution, constraint solving is inherently limited [16].

Evolution fuzzing assumes no knowledge of the implementation under test. Instead of generating purely random input, the approach obtains feedback from each

doi:10.4304/jsw.8.3.645-651

execution of the system, and determines the next input according to received feedback. Sparks, et al. [17] uses genetic algorithm to guide input selection based on basic block coverage of past inputs that have been tested. To obtain more complete coverage, [18,19] introduce sub-instruction profiling to improve the effectiveness. However, evolution fuzzing still generates test cases in a random way, and thus cannot bypass checks and verifications efficiently.

Model-based fuzzing has detailed understanding of the format being tested to generate inputs more intelligently. The understanding is usually based on a specification. Using network protocol specification, PROTOS [3] and SPIKE [4] generates a set of inputs which can be accepted by the protocol implementation. Peach [5] and Sulley [6] are general fuzz platforms, which generate or mutate fields in data samples according to the format specification. Stateful protocol fuzzers such as SNOOZE [7] and KIF [8] use a specification of protocol state machine to reach deep protocol states. To build more automated, flexible and measurable fuzzers, formal approaches [9-11] have also been used in recent years.

White box fuzzing and evolution fuzzing require profound knowledge in software debugging and reverse engineering, and test practices are usually tedious and time consuming. In contrast, model-based fuzzing defines test cases on the basis of input models and simplifies the overall analysis process. Therefore, model-based approach is widely adopted, and most successful fuzzers include facilities to model the structure of the data that is to be generated.

However, models of these frameworks are not well described, which lead to high redundancy in test practices. Furthermore, unintentional differences between two implementations of the same specification are common. If the target of fuzzing is inconsistent with specification, test practices will be limited. In this paper, we give our solutions to these deficiencies for model-based fuzzing.

III. Format Description

A. Format Characteristics

From the view of linguistics, format is a set of rules and regulations that define the lexeme, syntax and semantic of data elements. Lexeme describes the features of characters, syntax refers to the principles governing the structure of data, and semantic is the meaning used to understand human expression. There are two kinds of constraints between elements in structured data, namely, *lexeme-related* constraints and *syntax-related* constraints. Figure 1(a) shows the constraints between data elements in PNG (Portable Network Graphic) format. The lexeme features of DATA field should be consistent with LENGTH and CHECKSUM, and the syntax of CHUNK depends on the value of TYPE field. These constraints should be satisfied to generate semi-valid test cases.

Various script languages have been used to describe data formats for fuzz testing. PROTOS [3], SPIKE [4], KIF [8] use block-based language in ABNF (Augmented Backus-Naur Form), which is the standard syntax definition of a protocol specification. However, neither of them can describe semantics and constraints between data elements as ABNF is context-free. Based on XML, Peach [6] and SNOOZE [7] define lexeme-related constraints as special attributes of tag, but both of them cannot describe syntax-related constraints. For the CHUNK structures shown in Figure 1 (b) and Figure 1 (c), existing methods generally define them respectively, and lead to high redundancy in test practices.

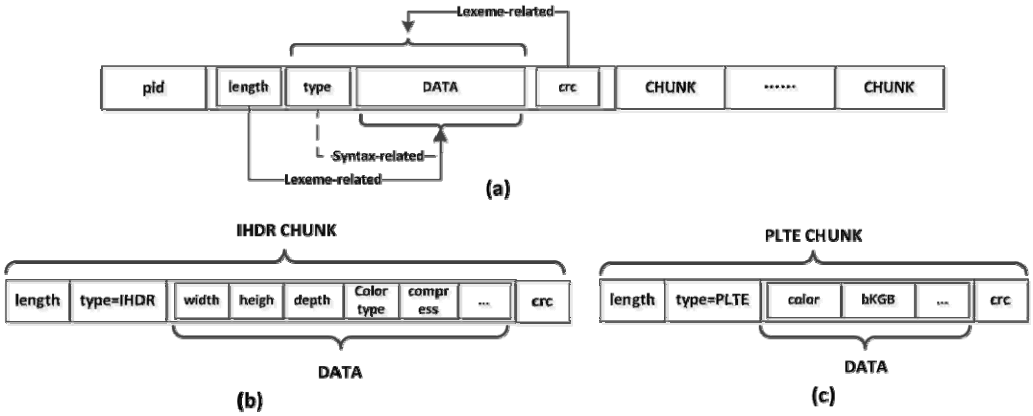

Figure 1. Constraints between data elements in PNG format

B. Model Definition

Attribute Grammar (AG) [20] is a formal way to define attributes for the productions of a formal grammar, and is suitable for data format with semantics. However, it is not powerful enough to model context-sensitive structure. Higher-order Attribute Grammar (HAG) [21] is an extension of AG in the sense that the distinction between the domain of parse-trees and the domain of attributes disappears, and attributes can appear in the left-hand side of a production. This extension allows HAG to select syntactically equivalent rules based on attribute value and describe context-sensitive structures. Furthermore, it has been proven that HAG has the same expressive power as Turing machines [21]. We define data format as a special HAG form.

Definition 1. Data format model is a 7-tuple <*F*, *S*, *P*, *Z*, *A*, *R*, *CS*>, where

- *F* is the set of atom fields in data.
- *S* is the set of dividable structures in data.
- *P* is the finite set of productions, production *p* has the form $x_0 \rightarrow x_1 \ldots x_n$, where $x_0 \in S$ and $x_i \in (F \cup S)$, $1 \le i \le n$.
- *Z* is the initial structure.
- $A=\cup A(x)$ is a finite set of attributes. The finite set of attributes $A(x)$ is associated with each data element $x \in (F \cup S)$, and if $a \in A(x)$, $x.a$ represents the attribute occurrence of x.
- $R=\cup R(p)$ is a finite set of attribute rules. Elements of $R(p)$ have the form $\alpha=f(\ldots, \beta, \ldots)$, f is the name of a function, α and β are the attribute occurrences of elements in production p.
- $CS=\cup CS(p)$ is the set of all context-sensitive structures. For each $p \in P$, $CS(p)$ is defined as $CS(p)=\{x_j \mid x_j=f(\ldots) \in R(p) \text{ and } 1 \le j \le n\}$.

Take PNG format in Figure 1 for instance, an abbreviated description of data model is given as follows:

```
F:={pid,length,type,crc,width,height,…}
S:={PNG,CHUNKS,CHUNK,DATA,DATA1,DATA2,…}
P:={<PNG>→<pid><CHUNKS>,
    <CHUNKS>→<CHUNK><CHUNKS>,
    <CHUNK>→<length><type><DATA><crc>,
    <DATA1>→<width><height><depth><colortype>…,
    <DATA2>→…}
Z:=PNG
A:={len,val,sem,…}
R:={length.val=DATA.len,
   crc.val=CRC32(chunktype.val,DATA.val),
   DATA=FUNC(type.val),…}
CS:={DATA}
```

In this model, the properties and semantics are viewed as field attributes. Moreover, both lexeme-related constraints and syntax-related constraints are defined as attribute rules. The structure of DATA is context-sensitive and can be calculated by function FUNC. We describe FUNC as a set of ordered pairs {<*IHDR*, DATA1>, <*PLTE*, DATA2>,…}. The first element of a pair is the value of TYPE field, and the second is the corresponding structure of DATA. The model describes data format more accurately than previous works.

IV. Grammar-driven Fuzz Testing

A. Overview

Once we formalize the description of data format, it is used to guide model-based fuzz testing. Many protocols are proprietary, or involve proprietary extensions to published specifications. In order to ensure the validity of test practices, we adopted mutation-based approach rather than generation-based approach to create test cases. Since single data sample only covers parts of file or network protocol format, it is essential to perform fuzz testing over multiple data samples [2]. However, if these samples are mutated blindly, the shared elements in different samples usually lead to numerous reduplications of test cases.

To improve efficiency of test practices, we propose a grammar-driven approach for fuzz testing. The process involves several phases, as shown in Figure 2.

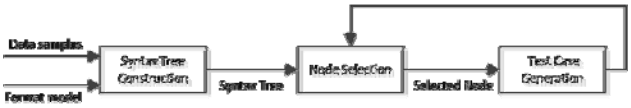

Figure 2. Overview of Grammar-driven fuzz testing

- **Syntax tree construction.** After data samples are collected, the corresponding format description model is formalized. Based on the model, the data samples are parsed, and syntax trees are constructed in top-down order.
- **Node selection.** In this phase, the syntax tree of each data sample is traversed to find untested node for mutation. To avoid redundancy in test practices, we record model elements that are tested and skip them in the future.
- **Test case generation.** The selected node is mutated according to its attributes, and it is combined with other nodes in the syntax tree to generate new test cases. To bypass checks and verification, we also modify the value of other nodes on the basis of attribute rules in the format model.

B. Syntax Tree Construction

Syntax trees serve as the basis for sample mutation and test case generation. A syntax tree can be defined as a 2-tuple (*N*, *E*), where *N* is the set of nodes, and $E \subseteq N \times N$ is the set of directed edges that indicate the hierarchical relation between nodes. Each node is labeled with the concrete value in data sample. Algorithm 1 recursively constructs the syntax tree of data samples in top-down order.

Algorithm 1 (Construction of Syntax Tree)

***Input*:** format model *M*, data sample *D*, syntax tree *ST*, structure *s*;
***Output*:** complete syntax tree *ST*;

```
ST_Construct(M, D, ST, s){
1.   p:=searchProduction(M, s);
2.   for each x in the right-hand side of p do
3.   n:=createNode(x);
4.       if (x∈F)
5.           label n with the concrete value of x in D;
6.       N:=N∪{n};
7.       E:=E∪{<node of s,n>};
8.       if (x∈S)
9.           if (x∈CS)
10.              x:=CS_evaluate(M, ST, x);
11.          ST:=ST_Construct(M, D, ST, x);
12.  return ST;
}
```

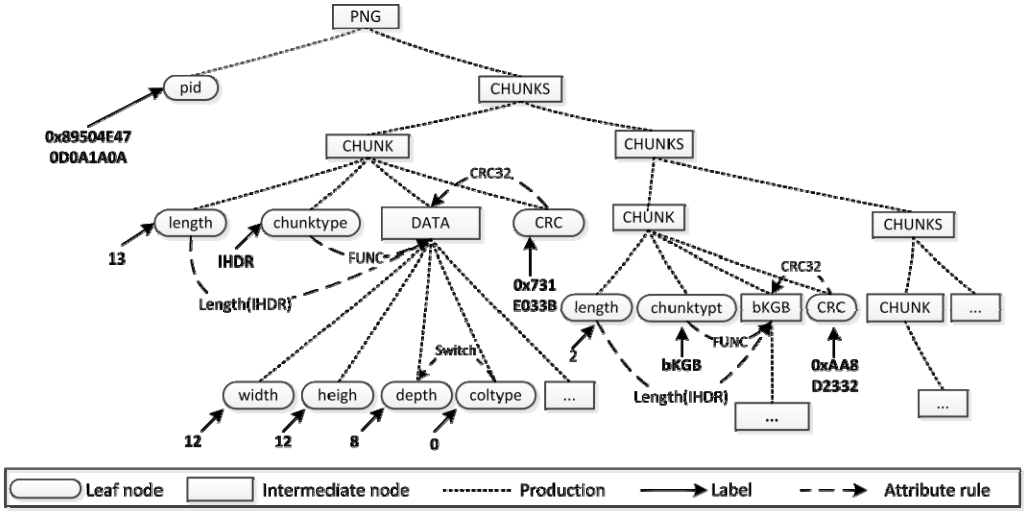

Figure 3. Syntax tree of PNG sample

Initially, *ST_Construct* is called where *ST* = <∅,{*Z*},∅> and *n*=*Z*. We use function *searchProduction* to find the production *p*, which indicates the composition of *s*. For each element *x* in the right-hand side of *p*, node *n* is generated as an instance of *x* (line 2-3). If *x* belongs to *F*, we label *n* with the concrete value of *x*, and add *n* to *ST* as a child node of *s* (line 6-7). If *x* belongs to *S*, function *ST_Construct* will be called recursively to construct sub *ST* (line 11). If *x* also belongs to *CS*, the specific form of *x is* evaluated in advance according to attribute rules (line 9-10).

We have formalized the description of PNG format in section 3.2. Taking data sample in table 1 and the format model as input, we are able to construct a syntax tree as illustrated in Figure 3. Compared with original data sample, syntax tree is more suitable to guide fuzz testing, since it contains information about structure and constraints.

TABLE I.
DATA SAMPLE OF PNG FILE

	Field	Size	Value
1	pid	8	0x89504E470…
2	length	1	13
3	chunktype	4	IHDR
4	width	1	12
5	height	1	12
6	depth	1	8
7	colortype	1	0
8	compression	1	0
9	filter	1	0
10	interlace	1	0
11	crc	4	0x731E033B
12	length	1	2
13	chunktype	4	bKGB
14	DATA	1	0
15	crc	4	0xAA8D2332
16	length	1	99
17	chunktype	4	IDAT
18	…	…	……

C. Node Selection

The purpose of the node selection phase is to find an instance of untested model elements in the syntax tree. The algorithm is given below.

Algorithm 2 (Node Selection)

Input: format model *M*, syntax tree *ST*, set of tested model elements *Ω*, intermediate node *n*;

node_Select(*M*,*ST*, *Ω*, *n*){

1. **for each** child node *v* of node *n* **do**
2. **if** *v.label*=NULL /* *v* is intermediate node*/
3. *node_Select*(*ST*, *Ω*, *v*);
4. *s* is the corresponding model element of *v*;
5. **if** (*s*∉*Ω*)
6. *testcase_Generate*(*M*,*ST*, *v*);
7. *Ω*:=*Ω*∪{*s*};

}

Algorithm 2 selects both leaf nodes and intermediate nodes as the objects of mutation. Since the concrete value of an intermediate node depends on its children, Algorithm 2 traverses the syntax tree in post order (line 1-3). If the model element of traversed node is untested, we add it to set *Ω* and mutate the node to generate test cases (line 5-7). Obviously, the efficiency of algorithm 2 depends on the expressive power of format model. For the redundancy existed in the format description will lead to repeated and pointless test cases (see Section 3.1), our approach is more efficient than previous work.

D. Test case Generation

In this phase, we generate test cases by node mutation. Depending on node type, there are mainly two different policies for node mutation.

- **Leaf node mutation.** Leaf nodes are instances of atom fields. Similar to PROTOS [3], we mutate leaf nodes to abnormal values according to semantic attributes to reduce meaningless test cases. For example, we concatenate '../' to valid string of file directory to trigger path traverse vulnerability.
- **Intermediate node mutation.** Intermediate nodes are instances of structures. Based on the policies in [10,22], we mutate them in the following ways: 1) remove or add a child node; 2) exchange location of

child nodes in data sample; 3) replace a child node with another node with different attributes.

After node mutation, we combine all the nodes of syntax tree into test cases. To meet the constraints between data elements, values of other nodes are modified according to attribute rules in format model. Take Figure 3 as an instance, the length of `DATA` node depends on the value of `length` node. If `DATA` is mutated, we will recount its bytes and modify the value of `length` node. Compared with previous works [4-6], our approach describes context sensitive constraints as attribute rules in HAG, and reduces the production of invalid test cases.

V. Experiments and Evaluation

A. Testing Framework

To verify the effectiveness of our approach, we implemented GDFT (Grammar-Driven Fuzzing Tool) based on Peach platform. The framework is illustrated in Figure 4, and the extensions of GDFT are shaded.

GDFT includes five modules: model editor, model parser, grammar-driven engine, testing agent and monitoring agent. Initially, model parser constructs syntax tree for each data sample. As the core of GDFT, the grammar-driven engine takes attribute rules and syntax tree as input, and guides the testing agent to generate test cases. In order to investigate crashes of target software, monitoring agent is used to interact with virtual execution environment and feedback runtime information. Compared with Peach platform, our approach has two advantages: 1) grammar model is used to guide test case generation instead of script language; 2) the attribute-based mutation policy is more intelligent and directed.

GDFT have been tested on a number of software implementations. The testing was performed on Windows 7, with Intel CoreE7500 CPU and 4GB memory. In addition, the overhead and performance of GDFT were compared with those of well-known fuzzing tools.

B. Case Study: Libpng1.2.43

We choose libpng [23] as case study for the format of PNG. Ligpng is the official reference library, and supports almost all PNG features. We tested FileFuzz [24], Peach [6] and GDFT on visualPNG [25] that embedded with libpng1.2.43. Only five abnormal values were used for FileFuzz to mutate bytes, or the overhead will be overwhelming.

The PNG specification defines twenty-one chunk types. Three chunk types are mandatory, and others are optional. In this case study, we randomly selected 20 PNG samples from Internet through Google Image Search, and calculated the redundancy of chunktype in these samples. Results are shown in Figure 5. As expected, all samples contain the mandatory `IHDR`, `IDAT` and `IEND` chunks, and most of the chunk types occurred more than twice in these samples.

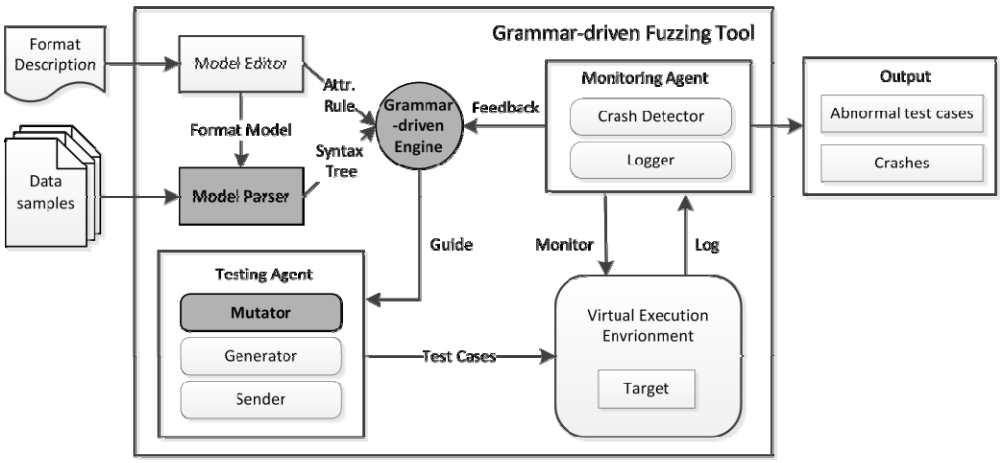

Figure 4. Framework of Grammar-Driven Fuzzing Tool

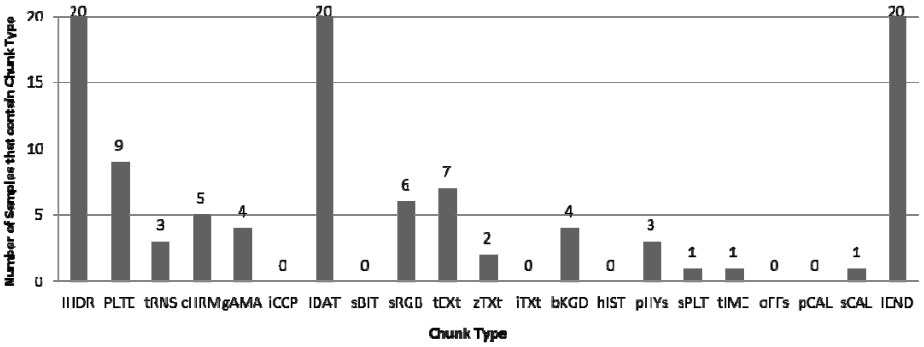

Figure 5. The redundancy of chunk type in samples

We measured the number of generated test cases of fuzz testing while the number of file samples increases from 1 to 20, as shown in figure 6. It can be seen that test cases generated by GDFT were obviously fewer than those generated by FileFuzz and Peach as the number of file samples increases. The number of test cases generated by FileFuzz and Peach are approximately linearly increased as the number of PNG samples increases, while the number of test cases generated by GDFT will reach saturation when the sample set exceeds a certain scale. This is because FileFuzz and Peach mutate all the chunks in PNG samples, while GDFT only mutates untested elements. It should be noted that GDFT generated more test cases for the first sample since more complex mutation policies were adopted.

In figure 7, we show the code coverage (calculated by Paimei [26]) of fuzz testing as the amount of PNG samples increases. Although there is not necessarily a correlation between code coverage and vulnerabilities uncovered, it is undoubtedly that unexecuted code will not reveal any vulnerability. GDFT achieved a higher level of code coverage than FileFuzz and Peach.

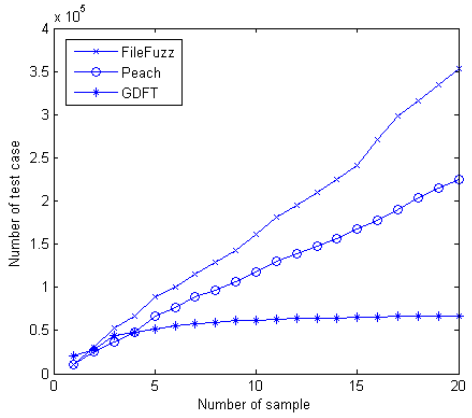

Figure 6. Number of generated test cases for visualPNG

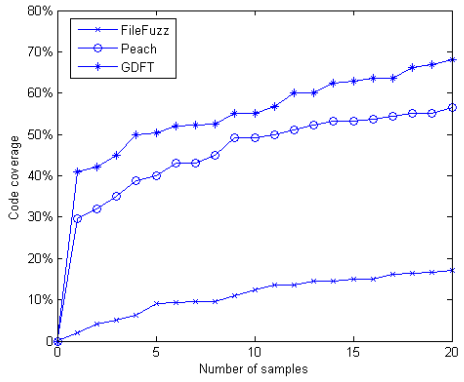

Figure 7. Code coverage of fuzz testing on visualPNG

TABLE II.
THE PERFORMANCE OF FUZZ TESTING ON VISUALPNG

tool	Test cases	Code coverage	Time(hr.)	Vulnerabilities
FileFuzz	353,318	15%	152.1	1
Peach	224,908	53%	102	3
GDFT	58,763	67%	27	5

To evaluate the performance of fuzz testing, we measured the code coverage, time cost and discovered vulnerabilities of these three tools when the amount of file samples increased to 20, as shown in table 2.

It can be seen that GDFT clearly outperforms Peach with multiple samples on visualPNG. We found that most of the test cases generated by FileFuzz were rejected by the checksum verification in visualPNG. Due to the inefficiency of random mutation policy, FileFuzz discovered just one known vulnerability with limited code coverage of 15%. Under the guidance of format specification in XML script, Peach tested visualPNG in depth with fewer test cases, and verified three known vulnerabilities (CVE-2010-2249, CVE-2011-3328, CVE-2011-3045). However, because of the generation of numerous repeated test cases, the testing process of Peach was still time-consuming and unacceptable.

Testing results show that GDFT took only one-quarter of the time cost of Peach and achieved higher code coverage. In addition to the three vulnerabilities verified by Peach, GDFT discovered two more vulnerabilities. One is CVE-2010-1205 and another is an unknown vulnerability. Take CVE-2010-1205 for instance, libpng contains a bug whereby visualPNG could receive an extra row of image data beyond the height reported in the header, potentially leading to out-of-bounds memory indexing. Because the structure of image data is context-sensitive, the bug was only detected by GDFT. The unknown vulnerability can be triggered by an abnormal sCAL chunk, which is also context-sensitive.

Furthermore, we found that sCAL chunk is uncommon in PNG format, and only the 14th and 17th samples contain instances for sCAL chunk. This confirms the necessity of fuzz testing with multiple samples.

VI. Conclusion and Future Work

In this paper, we proposed a grammar-driven approach for fuzz testing. Our approach proceeds in three steps: Firstly, we construct syntax trees of data samples according to the formal model of format specification. Secondly, each syntax tree is traversed to select untested nodes. Finally, we apply mutation-based technique to selected nodes and combine the syntax tree into test cases. Compared with previous work, our approach can significantly reduce invalid and duplicated test cases. Moreover, our approach is able to generate more intelligent test cases to trigger vulnerabilities. In the future, we plan to combine protocol state machine with our model, and research on stateful protocol fuzz testing. In addition, we will introduce reverse engineering techniques to extract the description of input format automatically.

Acknowledgment

This work is supported by Natural Science Foundation of Jiangsu China under Grant No. BK2011115, and also supported by Laboratory of Military Network Technology of PLA University of Science and Technology.

REFERENCES

[1] Michael Sutton, Adam Greene, Pedram Amini: Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley, USA, 2007.

[2] Xueyong Zhu, Zhiyong Wu, J. William Atwood, "A New Fuzzing Method Using Multi Data Samples Combination". Journal of Computers, vol.6, no.5, pp.881–888, May, 2011.

[3] Oulu University Secure Programming Group, PROTOS, http://www.ee.oulu.fi/research/ouspg/protos/index.html, visited on March 2012.

[4] Dave Aitel, The advantages of block-based protocol analysis for security testing, http://www.netsecurity.org/dl/articles/advantages_of_block_based_analysis.pdf, visited on March 2012.

[5] Pedram Amini, Sulley, http://code.google.com/p/sulley, visited on March 2012.

[6] Michael Eddington, Peach, http://www.peachFuzz.com, visited on March 2012.

[7] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, G. Vigna, "SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr", In Information Security Conference (ISC). LNCS, vol. 4176, Springer, Heidelberg, 2006, pp.343–358.

[8] Humberto J. Abdelnur, Radu State, Olivier Festor, "KiF: a stateful SIP fuzzer", In Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications, ACM, New York, 2007, pp.47–56

[9] Guoxiang Yao, Quanlong Guan, Kaibin Ni, "Test Model for Security Vulnerability in Web Controls based on Fuzzing", Journal of Computers, vol.7, no.4, pp.773–778, Apr, 2012.

[10] Chuanming Jing, Zhiliang Wang, Xia Yin, Jianping Wu, "A Formal Approach to Robustness Testing of Network Protocol. In:Network and Parallel Computing", LNCS, vol. 5245, Springer, Heidelberg, 2008, pp.24-37.

[11] Yang Yang, Huanguo Zhang, Mi Pan, Jian Yang, Fan He, Zhide Li, "A Model-Based Fuzz Framework to the Security Testing of TCG Software Stack Implementations", In IEEE MINES 09, IEEE Press, New York, 2009, pp.149–152.

[12] Barton P. Miller, Lars Fredriksen, Bryan So, "An Empirical Study of the Reliability of Unix Utilities", Communications of the ACM, ACM, vol.33, no.12, pp.32–44, December, 1990

[13] Cristian Cadar, Paul Twohey, Vijay Ganesh, Dawson Engler, "EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution", ACM Transactions on Information and System Security, ACM, vol.12, no.2, pp.1–38, February 2008.

[14] Cristian Cadar, Daniel Dunbar, Dawson Engler, "Klee: Unassisted andAutomatic Generation of High-coverage Tests for Complex Systems Programs", In Proceedings of the 8th USENIX conference on Operating systems design and implementation, ACM, New York , 2008, pp 209–224.

[15] Hyoung Chun Kim, Young Han Choi, Dong Hoon Lee, "Efficient File Fuzz Testing using Automated Analysis of Binary File Format", Journal of Systems Architecture, Elsevier, vol.57, no.3, pp.259–268, March 2011.

[16] Patrice Godefroid, Adam Kiezun, Michael Y. Levin, "Grammar-based Whitebox Fuzzing", In PLDI '08, ACM SIGPLAN Notices, ACM, New York, Vol 43, no.6, 2008, pp.206–215.

[17] Sherri Sparks, Ryan Cunningham, Shawn Embleton, Cliff C. Zou, "Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting", In 23rd Annual Computer Security Softwares Conference(ACSAC), IEEE Press, New York, 2007, pp.477–486.

[18] Will Drewry, Tavis Ormandy, "Flayer: Exposing Application Internals", In Proceedings of the first USENIX workshop on Offensive Technologies, USENIX Association, Boston, 2007, pp.1–9

[19] Bekrar S., Bekrar C., Groz R., Mounier L, "Finding Software Vulnerabilities by Smart Fuzzing", In Fourth International Conference on Verification and Validation, IEEE Press, New York, 2011, pp 427–430.

[20] JukkaPaakki, "Attribute Grammar Paradigms: a High-level Methodology in Language Implementation", ACM Computing Surveys, ACM, vol.27, no.2, pp.196–255, February, 1995

[21] Vogt H.H., Swierstra S.D., M.F. Kuiper, "High order Attribute Grammar", In Attribute Grammars, Applications and Systems, LNCS, vol. 545, Springer, Heidelberg, 1991, pp.256–296

[22] Petar Tsankov, Mohammad Torabi Dashti, David Basin, In-depth fuzz testing of IKE implementations, ftp://ftp.inf.ethz.ch/pub/publications/techreports/7xx/747.pdf, visited on March 2012.

[23] Greg Roelofs, libpng, http://www.libpng.org/pub/png/, visited on March 2012.

[24] Michael Sutton, FileFuzz, http://www.fileguru.com/FileFuzz/info, visited on March 2012.

[25] Willem Van Schaik, visualPNG, ttp://www.schaik.com png/visualpng.html, visited on March 2012.

[26] Pedram Amini, Paimei, htp://code.google.com/p/paimei, visited on March 2012.

Fan Pan is a PhD candidate at the Institute of Command Automation, PLA University of Science and Technology (PLAUST). His research interests include network security, and protocol reverse engineering. He received MS degree in computer science from the Uni. of Sci. & Tech in 2009. Contact him at Institute of Command Automation, PLA Uni. of Sci. & Tech., Haifu Road 1, Nanjing China, 210007; Email: dynamozhao@163.com.

Ying Hou is a Master candidate at the Institute of Command Automation, PLAUST. His research interests include network security, and protocol reverse engineering. He received BS degree in computer science from the Uni. of Elec. Sci. & Tech in 2010. Her email address is yinghou26@gmail.com.

Zheng Hong is an associate professor at the Institute of Command Automation, PLAUST. He received PhD degree in computer science from PLAUST in 2007. His research interests include malware detection and network security. His email address is hongzhengjs@139.com.

Lifa Wu is a professor at the Institute of Command Automation, PLAUST. He received PhD degree in computer science from the Nanjing University. His research interests include network security and protocol reverse engineering. His email address is wulifa@vip.163.com.

Haiguang Lai is an associate professor at the Institute of Command Automation, PLAUST. He received PhD degree in computer science from Nanjing University in 2006. His research interests include network security and management. His email address is hongzhengjs@139.com.