Fuzzing the Font Parser of Compound Documents

Hongliang Liang
School of Computer Science
Beijing University of Posts and Telecommunications
Beijing, China
hliang@bupt.edu.cn

Yuying Wang
School of Computer Science
Beijing University of Posts and Telecommunications
Beijing, China
wangyuying@bupt.edu.cn

Huayang Cao
National Key Laboratory of Science and Technology on Information System Security
Beijing, China
caohuayangwork@163.com

Jiajie Wang
China Information Technology Security Evaluation Center
Beijing, China
wangjj@itsec.gov.cn

***Abstract*—Currently, complex software (e.g. PDF readers) usually takes various inputs embedded with multiple objects (e.g. fonts, pictures), which may result in bugs. It is a challenge to generate suitable test cases to support fine-grained test to the PDF readers. Compared with the traditional blind fuzzing which does not utilize the information of input grammars, fuzzing with the model of the file format is an effective technique. In this paper, we leverage the structure information of the font files to select seed files among the heterogeneous fonts. A general construction method for generating suitable test cases is proposed. By this means, we can obtain test cases with low overhead. Moreover, to improve the expression ability of the font template in fuzzing PDF readers, we combine file reconstruction and template description. Our methods are evaluated on five common-used PDF readers, and proved effective in triggering crashes.**

Keywords—Compound Document Object; Fuzz Testing; Test Case Construction; Model-based Fuzzing; TrueType Font

I. INTRODUCTION

According to the CVE (Common Vulnerabilities and Exposures [1]) and NVD (National Vulnerability Database [2]), there are lots of high-risk vulnerabilities in PDF readers, browsers and office software. The number of PDF vulnerabilities has increased significantly[1]. PDF documents are often used as the carrier of the malicious code since they are portable and widely used. A PDF file is usually a compound document which may contain many objects, such as fonts, pictures, videos, audios. For these objects, PDF readers provide corresponding parsing modules.

In this paper, we focus on mining the vulnerabilities in font parsing engines[2] of PDF Readers, such as CVE-2010-2883 Adobe CoolType - SING Table 'uniqueName' Stack Buffer Overflow. In the existing methods, black-box testing is widely used, which is highly available, reproducible and simple for mining vulnerabilities. Especially, fuzzing [3] is an effective, fast and practical method to find bugs in applications. The key idea behind fuzzing is to generate and feed the target program plenty of test cases which are hopeful to trigger bug-inducing code fragments in the target program [4]. The cost and effectiveness of fuzzing are determined by the quantity and quality of the test suite. A set of effective seeds can save substantial machine time and trigger many crashes.

To test the font parsing engines of PDF Readers, some issues have to be solved. One, the PDFs gathered from the Internet are usually blended and ineffective for the font parsing engines. Considering that people usually use favorite fonts in their PDFs, so the test cases (PDFs) may have a low coverage of the font parsing engine. Two, if we crawl different fonts from Internet, we need to embed each font into a PDF file as test cases. Among the fonts we collected, there may be a large part that trigger the same functions. So we need to filter a smaller set as a representative. Three, since traditional blind fuzzing test does not utilize the information of input grammar, the cases generated with random mutation strategy often do not conform to the PDF file format. It is inefficient to deal with the embedded objects. It's also hard to trigger the inner font parsing engine in the PDF reader to find vulnerabilities. Fuzzing with the model based on the file format is an effective technique, but the adoption of a method is hindered by the difficulty of creating a generic test template. Constructing a thorough set of templates can be time-consuming because many different templates for one format are in need.

The main contributions of this paper are described as follows:

1) We propose a method for generating test cases to test the font parser in PDF readers. According to the physical structure and logical structure of PDF, we adjust the internal structure of the embedded font object and ignore other irrelevant objects (such as audio, pictures), which improves the efficiency of fuzzy testing.

2) We combine files reconstruction and template description to improve the expression ability of templates in fuzzing. We define a uniform specification according to the format of TrueType Font to reconstruct the heterogeneous samples, which makes a generic template for the heterogeneous fonts possible. We select the seeds according to the structure information of fonts rather than collecting coverage information for all the heterogeneous seeds.

3) We implement a tool called PFF and evaluate it by fuzzing five common-used PDF readers to find bugs using the constructed seeds and template in model-based fuzzing. The evaluation results show that our method can trigger more crashes in these readers than other tools.

[1] There are 2449 pdf-related vulnerabilities in NVD by the end of 2016.

[2] There are 30 PDF font-related vulnerabilities and 42 TrueType Font-related vulnerabilities in NVD by the end of 2016.

978-1-5090-6644-5/17 $31.00 © 2017 IEEE
DOI 10.1109/CSCloud.2017.42

The remainder of the paper is organized as follows. In section II, we describe the design and implementation of the methods we proposed. We present the evaluation and application of our methods in Section III. We discuss a few limitations in Section IV. And we present related work in Section V and conclude in Section VI.

II. THE DESIGN AND IMPLEMENTATION OF PFF

In this paper, we choose the font engine of PDF Readers as our test target. In Fig.2 the whole architecture diagram of PFF tool is presented, which includes modules of heterogeneous fonts collection, font processing and model building, fuzzing, result processing and final data processing. The workflow is as follows: First, the collected heterogeneous fonts are filtered and validated, and the fonts are structured and modeled. Then we embed the formatted font into the PDF files, and send the embedded PDF file and the PDF data model description file to the gray box fuzzy test module to test. We collect and backup the test results. Then we verify the correctness of crashes and finally get a repository to backup abnormal results. The vulnerability analysis just need to analyze the classified information.

As shown in Fig.1, we obtain font files (S) from the Internet. Then we filter the fonts according to the characteristics and the correctness verification (such as a checksum) of TrueType font. Then we select some correct fonts (S') to reconstruct. Finally, we construct the final PDF seeds with the reconstructed fonts.

A. The Structure of the Fonts and PDF

The fonts we crawled from the Internet contain several kinds of fonts, such as PostScript fonts, TrueType fonts, OpenType fonts, for multiple language, such as English, Chinese, Japanese, wherein TrueType and OpenType fonts are widely used in PDF documents.

A TrueType font file consists of a sequence of concatenated tables. A table is a sequence of words. Each table must be length-aligned and padded with zeroes if necessary. The first of the tables is the font directory as the index to other tables in the font. The directory is followed by a sequence of tables containing the font data. These tables can appear in any order. The tables have names known as tags.

Each table in the font file must have its own table directory entry. Table I shows the structure of the table directory.

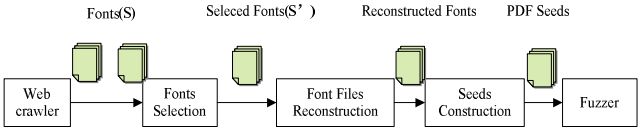

Fig. 1. The workflow for generating the set of seeds.

The PDF files [5] are highly structured and complex. The physical structure of a PDF file can be roughly divided into the following sections: Header, located in the first line of the file, is used to identify which version of the PDF file is. The file body (Obj), as an important part of the PDF file, consist of a series of objects. The cross-reference table (Xref) enumerates the offset information of each object in the body of the file. The Trailer summarizes the information of the author and the objects. At the end of the file, it records the cross-reference table offset to facilitate the PDF reader to locate the cross-reference table and specific objects quickly.

To parse the PDF files, the parse engine firstly gets the first cross-reference table location according to the end of the file. Secondly, it gets the root object and locates the root node to parse. Finally, it parses all the objects according to the relation between objects.

B. Seeds Selection

To test each module, we usually collect a large number of heterogeneous test cases to achieve high coverage of test objectives. Given million, billions, or even trillions of heterogeneous font files, which should we use when fuzzing the font engine of a PDF reader? Their types or structures may be different. How should we select seed files to use for the fuzzer? Specifically, we crawled a set of seed files S consisting of 26,897 heterogeneous fonts from diverse websites. Fuzzing each program for a sufficient amount of time to be effective across all seed files is computationally expensive. Maybe some subset of these test cases can meet the test requirements. So which subset of seed files $S' \in S$ shall we use for fuzzing?

Existing research [6,7,8] suggest using executable code coverage as a seed selection strategy. It is expensive and hard to collect accurate coverage information. We make use of the structure information of TrueType font to select seed files.

Since the heterogeneous fonts collected from Internet have different types and structures, it is difficult to guarantee the correctness (e.g., some checksums are not correct). Therefore, we filter the original samples according to the features of fonts rather than the file extension. Then we remove the duplicate fonts which have the same hash value. Finally, we parse the rest font files. We verify the checksum, offset and length. The incorrect fonts will be removed.

We collect the tags of tables that each font contains (statistical information is shown in Table II).

According to the statistics, we could find out a set of font samples with high coverage. In order to cover more property for every table, we can select multi-round.

TABLE I. THE TABLE DIRECTORY

Type	Name	Description
uint32	tag	4-byte identifier
uint32	checkSum	checksum for this table
uint32	offset	offset from beginning of sfnt
uint32	length	length of this table in byte

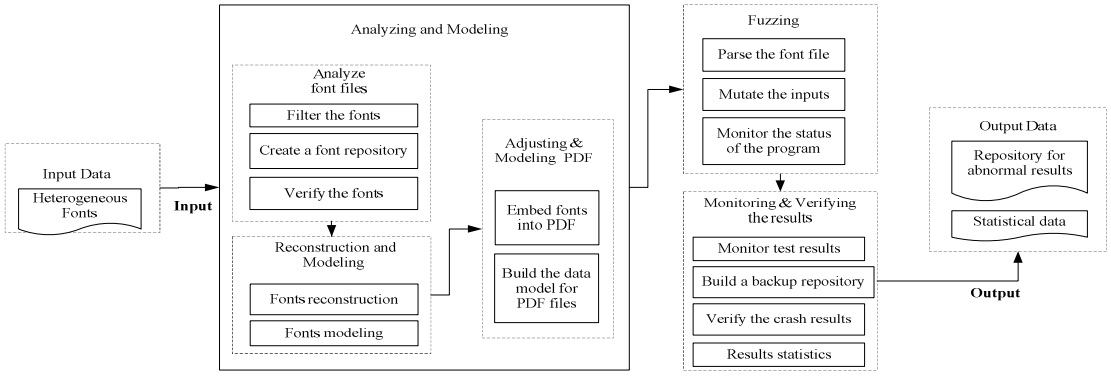

Fig.2. Architecture of PFF

TABLE II. THE TABLE OF FONT DESCRIPTION STATISTICS

	head	name	…	LTSH	PCLT	VDMX
dol-1.ttf	1	1	…	0	0	0
fd-40.ttf	1	1	…	1	0	1
fwm-70.ttf	1	1	…	0	1	1
gf-127.ttf	1	1	…	1	1	0
…						

In Table II, the first row is a complete tag set of tables. The first column indicates the name of the font, which is composed of the abbreviation of the source address and the number. It is convenient for analysis. The 0 represents that the font don't contain the table of the column, and the 1 represents the contrary.

After the collected description information, we use the following algorithm to select fonts seeds which calculates and returns a cover set *C*.

Algorithm 1: SelectFonts(*F*)

Intput: *F*, the set of collected fonts
Output: *C*, the selected fonts set

```
1   begin
2    A ⇐ {"head": 0, "name": 0, … , "VMDX": 0}
3    C  ⇐ Ø                    // The set of selected fonts
4    while F≠Ø do
5      S ⇐ random(F)       // S is a randomly selected font
6      F ⇐ F\S             // Remove S from the set F
7      for i from 1 to L_S  // L_S is the number of tables in S
8        if A[S.tables[i]] = 0
9          C ⇐ C∪S
10         A[S.tables[i]]: = 1 //The table is covered by set C.
11   return C
12  end
```

F means the fonts remained after filtering. *A* is a dictionary that records which tables have been covered by the *C*, and all keys are assigned 0 initially. 'head' represents the tag of a table named head. The *C* represents the selected fonts set, and it is empty initially. S is a randomly selected font. We can use function SELECT_FONTS to select a set of fonts to cover as many tables as possible. In order to avoid choosing all the fonts from a same source, the algorithm selects the next font object randomly. A TrueType font file consists of a sequence of concatenated tables. We traverse the tables in *S*. If a new table is covered by the selected font, we put it into the final set. Then the states of the covered tables are updated.

C. File Reconstruction and Template Description

Though the official website of TrueType Font [9, 10] has offered the standard file format, most of the font files we gathered don't conform the format. For example, the number and the order of tables in each font is different, and there may be different versions for the same table, even 4-byte alignment appears in the description tables. According to the official font specifications, we create models for the data and type, then add relationships (such as offset, length, count, or fixup) between the data elements, by using Pit [8] language. We can't directly construct an effective unified template directly for different fonts, for there are many attributes to be determined dynamically. It will cost much to construct templates for every testing cases.

In order to overcome the shortcomings of the expression ability of common template, we reconstruct the font files, reorder the tables in the font to a uniform standard and add the auxiliary information to help PFF parse the file content according to the data models.

According to the unified table name list obtained in II.B, we sort the tables to reconstruct the font. We make the order of all internal font tables consistent. After parsing the original font file to obtain the table directory, we copy the table directory from the original font to the new font. The contents of the corresponding table are read in the table name list order and written into the new font. Before each description table is added, we add a tag which is the name of the description table. When we use the fonts to test, PFF will identify the content of the tables according to the added labels and select the corresponding data model to mutate. Note that some of these tables have unique formats that require special construction. For example, the table 'head' involves font verification. We need to record the new offset of table 'head' and calculate the value of checkSumAdjustment.

Algorithm **2** shows part of the reconstruction of the cmap table. According to the structure of cmap table, the data should be aligned every four bytes. There are supplementary bits, which will affect the description of template, such as the size of

each parameter. So we add some aid information to help the fuzzer know where the target parameter is according to data models.

Algorithm 2: ReconstructCmapTable(T_{old})

Intput: T_{old}, the table to be reconstructed
Output: T_{new}, a new version of cmap table

```
1  begin
2    Summary ← parseHead(T_old)
3    offsetList ← sortSubtablesOffsets(Summary)
4    T_new ← modify (Summary)
5    L_pad ← Offset_first − L_Summary
6    if(L_pad > 0)
7        for i from 1 to L_pad
8            T_new.append('0')
9    for each subtable ∈ Summary.subtables
10       T_new.append("formatxx")
11       copyOldSubtable2NewTable(subtable, T_new)
12   return T_new
13 end
```

First, (*line 2 in Algorithm 2*) we parse the head of table cmap to get the *Summary*. It contains the length of the whole cmap table and the directory of subtables. The directory contains the offset and length of each subtable. (*line 3 in Algorithm 2*) We get the *offsetList* from the *Summary*. We sort the offsets to get the offset of the first subtable. Because we will add some info to the new cmap table. (*line 4 in Algorithm 2*) We should modify the summary to construct a new cmap table (T_{new}). We will modify the length for the whole cmap table and the new offsets for each subtable. For the order of subtables is disordered. $Offset_{first}$ is the offset of first subtable and $L_{Summary}$ is the length of Summary. L_{pad} is the length between *Summary* and the first subtable. We will pad zeros between them in the new cmap table. Then we get a subtable from the subtables in *Summary* and add a token string ("formatxx") before every subtable. Then copy the old subtable to the new cmap table (*line 11 in Algorithm 2*). Finally, we can get a new cmap table with aid information.

The corresponding data model is as follow. If the seed which the PFF is parsing begins with 'formatxx', it will be cracked as format 'cmap_format_4_block'. The cracker will recognize the position of next subtable according to the 'formatxx'. It allows the template to automatically identify how many items of glyphIndex-Array occur in 'cmap_format_4_block'. (occurs = '255' represents that the number of 8bites-object which the glyphIndex-Array can contain is from 0 to 255)

```
<Block name="cmap_format_4_block">
  <String name="pad8" value="formatxx" token="true"
mutable="false"/>
  ……
   <Number name="glyphIndexArray" size="8" signed="false"
endian="big" mutable="true" occurs="255"/>
</Block>
<Block name="cmap_format_x_nop_block" />
```

D. Modular Generation of Test Cases

Now the question is how to send the fonts to the font engine? It is impractical to embed so many fonts into PDFs manually.

We propose a general method for building test cases in no compression mode. We can generate legal and effective testing case by embedding the specific object (such as font) into PDF file with little redundant data. This method reduces the unrelated data parsing cost and builds the foundation for the model-based fuzzing.

If we use fonts with different size to directly replace the font-object in PDF sample, it will break the dependencies of logical structures in a PDF file, and make the font unable to be parsed. Thus, this issue can be deal with as follows.

According to the structure and parsing process of PDF files, we adjust the internal structure of unzipped PDF template file and modularize it. Then we quickly create effective test cases using the reconstructed fonts, as shown in Fig.3. First, with the PDF sample F0 which embedded font file, we reorder the objects in the file. We move the font object to the end of the object list and modify the related properties in Xref to get a new file F0'. Then we divide F0' into three files: F1, F2 and font. F1 contains header and the objects except font-object. F2 contains the new Xref and Tail. Afterwards we can replace the font module directly, and get the PDF file (F') embedded font file by splicing F1, F2 and a reconstructed font.

III. Experimental Evaluation

We now present our experiments to verify the correctness of the methods proposed in Section II. All of experiments were run on virtual machine (VM) instances with the same operating system. We use workstation HP Z840, equipped with 128G RAM, 2TB SATA * 2 + 512GB SSD * 2 disk storage. We got 26,897 true type font files by crawling the Internet and 5314 samples after the filtering, verification and reconstruction as discussed in Section II. We fuzz five common-used PDF readers using the methods proposed in the paper. They are Expert PDFReader-v3.5.70.0, Xchange PDFViewer 2.5.316.1, Nuance PDFReader 6.0, Apabi Reader_4.5.2.1790 and Perfect PDFReader_8.0.

A. Seed Coverage Statistics

In order to illustrate the validity of obtaining test cases with the above method, we collect the coverage of constructed test samples. On Linux platform, we use Okular to open the PDF samples and trace the program execution with Valgrind. We collected the number of completed blocks produced in 3 minutes.

We divide the area into 16 parts equally according to the completed blocks. We also divide the area into 20 parts, 24 parts and 30 parts. We assure the different code coverage is caused by different fonts, because other parts are the same except fonts in the test cases. These four different methods of division show that the cases did not distribute in a small range. We obtain a set of samples with a certain coverage.

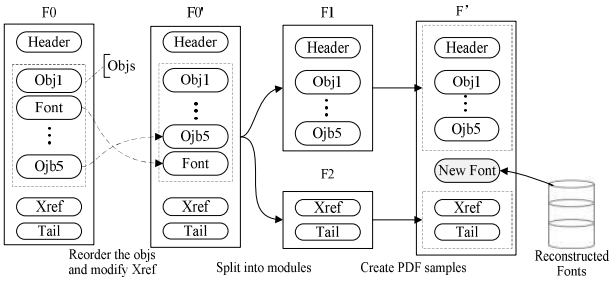

Fig. 3. The process to create PDF samples with fonts

B. Seed Selection Algorithm Comparison

In order to illustrate the effectiveness of filtering strategy proposed in this paper, we designed the following comparison experiments:

- Filtering strategy 1#: based on the coverage statistics of the file description table (section II.B), 61 samples were obtained after 3 rounds of filtering.
- Filtering strategy 2#: 61 samples were randomly selected from 5314 samples.
- Filtering strategy 3#: the samples were sorted according to the number of code blocks completed by each sample, and then 61 samples were taken at equal intervals.

For the comparative experiments of the three different filtering strategies, we use the same experimental environment to test Expert PDFReader. 61 seeds were distributed to two virtual machines. Each VM is given 72 hours to run and collect the crash information of each strategy.

The number of different crashes in Table III represents the number of different tags generated by msec.dll [11]. The total number of crashes in Table IV represents the number of all crashes. Obviously a tag may correspond to multiple crashes.

Table III and Table IV show that, the use of filtering strategy 1# (Filter based on file structure) get more exploitable crashes than the other strategies. The total number of crashes also shows the effectiveness of the strategy based on the file structure.

C. The Evaluation of Test Case Construction and Model Description

In order to illustrate the effectiveness of test case construction combined with data model, we designed the following experiments:

The first experiment is fuzzing based a simple template which described the seed as a block. The second experiment is fuzzing based the methods in Section II, we reconstruct the seeds and design a detailed template. The fuzzing results are shown in Table V.

In table V, the first row is classified by the msec.dll. The first column represents the templates we used. The numbers show the unique crashes identified by their stack hash.

It can be seen that fuzzing based on the method that combines the reconstructed font and data model can trigger more unique crashes than the simple template-based fuzzing.

To compare PFF with other fuzzer, we select BFF (the CERT Basic Fuzzing Framework) and PeachFuzzer. We ran these fuzzers on same samples. The fuzzing results are shown in Table VI.

The experiment results show that the method we proposed can get more different crashes than the traditional blind fuzzing with random mutation strategy. Fine-grained template can assist in model-based fuzzing. At the same time, the unified data model is practical for the constructed TrueType fonts. In addition, we also constructed test cases with OpenType fonts, the method in II.C is suitable too.

TABLE III. FILTERING STRATEGIES AND THE NUMBER OF UNIQUE CRASHES

Filtering strategies	EXP	PEX	PNE	UNK
Filtering strategy 1#	20	20	3	64
Filtering strategy 2#	17	26	6	62
Filtering strategy 3#	19	21	4	51

Note: (EXP= Exploitable, PEX= Probably Exploitable, PNE=Probably Not Exploitable, UNK=Unknown)

TABLE IV. FILTERING STRATEGIES AND THE TOTAL NUMBER OF CRASHES

Filtering strategies	Total number of crashes
Filtering strategy 1#	4,243
Filtering strategy 2#	1,553
Filtering strategy 3#	1,668

TABLE V. THE UNIQUE CRASHES FROM DIFFERENT TEMPLATES

	EXP	PEX	PNE	UNK
Simple template	4	4	1	9
Detailed template	28	45	4	81

TABLE VI. THE NUMBERS OF CRASHES

	EXP	PEX	PNE	UNK
BFF	590	181	2	289
PeachFuzzer	130	5	5	28
PFF	2,620	253	28	2,761

D. Case Analysis

We present a generic template for TrueType font and demonstrate the efficacy by detecting bugs using the constructed seeds and template in model-based fuzzing. We reproduced a classic vulnerability (AdobeReader CVE-2010-2883). Besides, we fuzz five common-used PDF readers using the methods proposed in the paper. There are thousands or even millions of iterative tests which can cause the target program crash. We discover that there is an integer overflow defect in xchange PDFViewer and a high-risk vulnerability which can be exploited to execute arbitrary code in Expert PDFReader.

IV. DISCUSSION AND LIMITATIONS

In this paper, we focus on the construction of test cases for fuzzing a certain module (e.g. font engines) embedded in complex software. The approaches proposed in this paper can also be applied to fuzz engines which are used to parse other kind of objects, such as pictures, audios and videos. Test cases achieving high coverage can accelerate the fuzzing process no

matter the fuzzer is based on mutations or symbolic execution. For example, when complex software is tested with symbolic execution, our approaches may help mitigate the path explosion and constrained solution problem.

The experiments show that the methods we proposed are effective in making use of the information of file formats, but require a lot of manpower to analyze the file formats. If there is no official specification for the formats of input files, our methods will degenerate into a simple random variation.

V. Related Work

A. Rebert et al [12] provided several algorithms for selecting seed files. They selected seeds from millions of PDF files collected from Internet. Je-Gyeong Jo [13] used "PDF/XML Architecture Working Samples" file that is provided from Adobe company to create abnormal PDF. This file uses most PDF grammars and functions. They cared little about the internal structure of the target software. They tested the whole application using the seeds selected. Some fuzzers use runtime code coverage as the matrix of seed selection. However, it is expensive to collect the coverage information of millions seeds. We focus on the construction of test cases for fuzzing certain modules (e.g. font engine) embedded in complex software. Hua Yin et al propose a novel classification algorithm using Ensemble Feature Selections (EFS) for imbalanced-class dataset. This algorithm utilizes the superiority of EFS in accuracy, then considers the diversity and imbalance in the designing appropriate feature subset objective function to make it fit for the imbalanced dataset [14]. In this paper, we made use of the feature of fonts to select the seeds. The structure information of fonts is clear and structured. It is suitable for filtering as the basis.

The traditional fuzzing tools, such as BFF [15] (the CERT Basic Fuzzing Framework), do not utilize any information of the input files, but only use some predefined rules to randomly mutate the given well-formed seed file(s) to create malformed inputs. The mutation rules include bit flips, byte removals, and byte copies [16]. It always is inefficient to get through the checksum of compound documents.

To make the fuzzer smarter, some researchers introduce symbolic execution [17], taint analysis [18-20] and code instrumentation into fuzzing [21]. White-box fuzzing is based on the knowledge of internal logic of the target program [22]. It uses a method that in theory can explore all execution paths of the target program. But the drawback of these techniques is obvious when applied to complex software because of the path explosion problem.

Some black-box fuzzing tools also utilize in-put-specific knowledge [23] or grammar [24] to generate semi-valid inputs. For PDF readers, fuzzing with the model based file format is more effective. FileFuzz, SpikeFile and PeachFuzzer apply this method. However, it is difficult to build a general test model for one file format. The emerging new data types bring tremendous challenges to data mining. Finding a complicated classifier is not an easy way and such a classifier may overfit for the data. Preprocessing these data before classification is a more direct method [25]. We preprocessed the collected fonts to have a unified format before we created a general model.

VI. Conclusion

This paper selected sub-modules of complex software as test targets (e.g. font parser in PDF readers) and constructed effective test cases in bulk and in a modular manner. Seed files were filtered based on their structural information. Moreover, we reconstructed font files with added auxiliary information. We evaluated our methods on five common-used PDF readers and the experimental results demonstrated the validity of our approach. In the future, we will combine mutation strategy with symbolic execution for achieving efficiency and high coverage.

References

[1] Common Vulnerabilities and Exposures. https://cve.mitre.org/,2017.

[2] National Vulnerability Database. http://www. nvd.nist.gov/,2017

[3] B. P. Miller, L. Fredriksen and B. So, "An empirical study of the reliability of UNIX utilities," Communications of the ACM, vol. 33, pp. 32--44, 1990.

[4] P. Oehlert, "Violating Assumptions with Fuzzing," IEEE Security & Privacy, vol. 3, pp. 58-62, 2005.

[5] Portable Document Format]. https://en.wikipedia.org/wiki/Portable_Document_Format,2017.

[6] H. Abdelnur, R. State, O. J. Lucangeli, and O. Festor, "Spectral Fuzzing: Evaluation & Feedback," HAL - INRIA, 2010.

[7] D. Duran, D. Weston and M. Miller, "Targeted taint driven fuzzing using software metrics,": CanSecWest, 2011.

[8] Mark. PeachFuzzer. http://peachfuzzer.com/,2017.

[9] TrueType™ Reference Manual. https://developer.apple.com/fonts/TrueType-Reference-Manual/,2017.

[10] OpenType Tables. https://www.microsoft.com/typography/otspec/otff.htm#otttables,2017.

[11] !exploitable Crash Analyzer - MSEC Debugger Extensions. https://msecdbg.codeplex.com/,2017.

[12] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing,", 2014, pp. 861-875.

[13] J. G. Jo and J. C. Ryou, "HTML and PDF fuzzing methodology in iOS,", 2016, p. 8.

[14] H. Yin, K. Gai and Z. Wang, "A Classification Algorithm Based on Ensemble Feature Selections for Imbalanced-Class Dataset,", 2016, pp. 245-249.

[15] A. D. Householder and J. M. Foote, "Probability-based parameter selection for black-box fuzz testing," 2012.

[16] E. Jääskelä, "Genetic Algorithm in Code Coverage Guided Fuzz Testing," 2015.

[17] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," Communications of the Acm, vol. 56, pp. 82-90, 2011.

[18] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection,", 2010, pp. 497-512.

[19] V. Ganesh, T. Leek and M. Rinard, "Taint-based directed whitebox fuzzing,", 2009, pp. 474-484.

[20] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A Taint Based Approach for Smart Fuzzing,", 2012, pp. 818-825.

[21] W. Drewry and T. Ormandy, "Flayer: exposing application internals,", 2007.

[22] J. DeMott, "The evolving art of fuzzing," DEF CON, vol. 14, 2006.

[23] J. De Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations,", 2015, pp. 193-206.

[24] Y. K. Su, S. Cha and D. H. Bae, "Automatic and lightweight grammar generation for fuzz testing," Computers & Security, vol. 36, pp. 1–11, 2013.

[25] H. Yin and K. Gai, "An Empirical Study on Preprocessing High-Dimensional Class-Imbalanced Data for Classification,", 2015.