2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE) | 979-8-3503-5388-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISSRE62328.2024.00037

Multi-level Fuzzing for Document File Formats with Intermediate Representations

Yifan Wang
Computer Science Department
Stevens Institute of Technology

Jun Xu
Kahlert School of Computing
The University of Utah

***Abstract*—This paper focuses on fuzzing *document software* or precisely, software that processes document files (e.g., HTML, PDF, and DOCX). Document software typically requires highly-structured inputs, which general-purpose fuzzing cannot handle well. We propose two techniques to facilitate fuzzing on document software. First, we design an intermediate document representation (DIR) for document files. DIR describes a document file in an abstract way that is independent of the underlying format. Reusing common SDKs, a DIR document can be lowered into a desired format without a deep understanding of the format. Second, we propose multi-level mutations to operate directly on a DIR document, which can more thoroughly explore the searching space than existing single-level mutations. Combining these two techniques, we can reuse the same DIR-based generations and mutations to fuzz any document format, without separately handling the target format and re-engineering the generation/mutation components.**

To assess utility of our DIR-based fuzzing, we applied it to 6 PDF and 6 HTML applications (48-hour) demonstrated superior performance, outpacing general mutation-based fuzzing (AFL++), ML-based PDF fuzzing (LEARN&FUZZ), and structure-aware mutation-based fuzzing ((NAUTILUS) by 33.87%, 127.74%, and 25.17% in code coverage, respectively. For HTML, it exceeded AFL++ and generation-based methods (FREEDOM and DOMATO) by 28.8% and 14.02%.

1. Introduction

Document files (e.g., HTML, PDF, and WORD) are used everywhere. For instance, Adobe estimated there may be up to 2.5 trillion PDF files in the world [1]. Software processing document files (e.g., PDFium Viewer and Office Word) has, thus, become a part of our daily life. However, this type of software often has a large codebase and high complexity, consequently containing many bugs that endanger security. For instance, over 600 vulnerabilities in Acrobat Reader were reported to the CVE database in the past 10 years [2], which affected a huge number of users. Therefore, it is critical to develop techniques to discover bugs in such software.

To find bugs in software, fuzzing [3] is one of the most practical techniques, thanks to its easy application to production-grade software. General-purpose fuzzing tools, such as AFL [4] and AFL++ [5], can also be applied to document-processing software (or *document software* for short). However, such fuzzing tools are less effective, as they cannot produce the highly-structured inputs desired by document software. Compared to general-purpose fuzzing, generation-based fuzzing is a more effective approach. Technically, generation-based fuzzing follows a grammar model of the target document format—summarized manually [6], [7] or obtained using machine learning [8], [9]—to assemble structure-correct testcases. Generation-based fuzzing can be further enhanced by integrating coverage-guided, structure-aware mutation [6], [10], [11].

While existing generation-based fuzzing and structure-aware mutation tools have demonstrated success on document formats such as HTML [6] and XML [8], they still carry two major limitations when applied or extended to other formats. First, following these tools, we need to create a grammar model for every new format. If done manually, this can require intensive human effort [6], [7]. Alternatively, we may leverage automated methods like machine learning [8], [9] to derive the grammar models. However, these methods tend to only cover simple or partial grammar. Second, these tools typically only run mutations at a single level. For instance, NAUTILUS [10] represents a document as a tree and performs mutations at the sub-tree level. In contrast, FREEDOM [6] conducts mutations at the attribute level of HTML files. Focusing mutations on a single level cannot sufficiently gain the benefits of other mutations and, thus, limit the exploration space.

In this paper, we present a new scheme to overcome the above two challenges in fuzzing document software. Sitting at the core of our scheme are two ideas. First, we introduce an intermediate document representation, named DIR, to describe a document file in an abstract way. At a high level, DIR represents a document as a set of pages where each page consists of a group of objects (e.g., text, figure, table, etc.). Each object is described by its attributes (e.g., a text object can be described by its location, data, font, size, and color). By design, DIR is general enough to support the various document formats we are using today. Thus, the generators and mutation components built for DIR can be reused for any format. To actually fuzz the target software, we still need to lower a DIR document, produced by the generators or derived by the mutation components, to the target format, which would still require a grammar model of the target format. Fortunately, we find that many SDKs exist to process common document formats. These SDKs often provide interfaces that can be reused to lower a DIR document to the target format without understanding the underlying grammar. Thus, by combining the DIR-based generation/mutation and the

2332-6549/24/$31.00 ©2024 IEEE
DOI 10.1109/ISSRE62328.2024.00037

proper SDKs, we can easily fuzz any desired document format.

Second, we propose to run structure-aware mutations at various levels of the DIR documents, including the page level, the object level, and the attribute level. Contrary to single-level mutations, our multi-level mutations offer a more comprehensive exploration of the search space and thus, can more thoroughly cover the code space and discover bugs. To further introduce unexpectedness, we finally couple the structure-aware mutations with random mutations offered by popular tools such as AFL and AFL++.

To understand the feasibility and utility of our DIR-based, multi-level fuzzing scheme, we apply the scheme to build a fuzzing tool called MODIFIER for both PDF and HTML. We show that by leveraging our DIR and the Foxit PDF SDK [12]/WebTookKit [13], MODIFIER can effectively generate and mutate PDF or HTML files with only a basic understanding of their grammar. This demonstrates the feasibility of our fuzzing scheme. The performance of MODIFIER was compared to other fuzzing tools on 6 PDF applications and 6 HTML applications. Using edge coverage as the metric, MODIFIER outperforms general mutation-based fuzzing (AFL++ [5]) by 33.87%, ML-enabled generation-based fuzzing (LEARN&FUZZ [9]) by 127.74%, and structure-aware mutation-based fuzzing (NAUTILUS [10]) by 25.17% on PDF. Applied to HTML applications, MODIFIER outperformed general mutation-based fuzzing (AFL++ [5]) by 28.83% and generation-based fuzzing (FREEDOM [6] and DOMATO [7]) by 14.02%.

In summary, we make the following contributions.

- We design a general intermediate representation (DIR) to describe document files, facilitating fuzzing on various downstream document formats without a deep understanding of their grammar.
- We propose multi-level DIR mutations for document files. Our multi-level mutations can enable better exploration of the code space and improve the fuzzing effectiveness on document files.
- We apply DIR-based fuzzing and multi-level mutations to PDF and HTML. Our evaluation shows that our fuzzing tool outperforms the existing ones, demonstrating the benefit of our fuzzing scheme. **Our code has been anonymously released at** https://github.com/junxzm1990/hierachy-fuzzing.

2. Background and Motivation

2.1. Document Software Fuzzing

To test document software, an effective approach is *generation-based fuzzing*. Generation-based fuzzing [14] follows the grammar of the target document to assemble inputs that are lexically and syntactically correct, offering a higher probability of reaching deeper code. There are two major strategies to build the grammar. The first strategy relies on manual efforts to summarize the grammar and encode the grammar as production rules. The strategy has been applied on HTML [6], CSS [7], and SVG [7].

```
%PDF-1.4 /*header*/
1 0 obj /*root object*/
<<
  /Type /Catalog
  /Pages 2 0 R
>>
endobj
2 0 obj /*list of pages*/
<<
  /Type /Pages
  /Count 1
  /Kids [3 0 R]
>>
endobj
3 0 obj /*page object*/
<<
  /Type /Page
  /Parent 1 0 R
  /MediaBox [0 0 614 794]
  /Contents 4 0 R
  /Resources 5 0 R
>>
endobj
4 0 obj /*stream object*/
<<
  /Length 58
>>
stream
BT
/F0 1 Tf
12 0 0 12 10 750 Tm
(Hello, World) Tj
ET
endstream
endobj
5 0 obj /*font object*/
<<
  /ProcSet [/PDF]
  /Font <<
    /F0 6 0 R
  >>
>>
endobj
6 0 obj /*font content*/
<<
  /Type /Font
  /Subtype /Type1
  /BaseFont /Helvetica
>>
endobj
xref /*xref table*/
0 6
0000000000 65535 f
0000000009 00000 n
0000000062 00000 n
0000000125 00000 n
0000000239 00000 n
0000000343 00000 n
0000000412 00000 n
trailer /*trailer section*/
<<
/Size 6
/Root 1 0 R
>>
startxref
428
%%EOF
@
```

Figure 1: An example PDF file displaying "Hello, World" at the beginning of the first page. The file spans 2 columns.

In contrast, the second strategy learns the grammar from a large corpus of existing documents, which has been employed to fuzz XML [8], XSL [8], and PDF [9].

Pure generation-based fuzzing runs in a blackbox manner without using any runtime feedback, which may lack the guidance to cover many code paths. This motivated the follow-up research to introduce coverage-guided, structure-aware mutation on generated document files [6], [10]. NAUTILUS [10] represents a generated input as a derivation tree (similar to an abstract syntax tree) and runs mutations at the subtree level (e.g., replacing a subtree with another). New inputs produced during the mutations, if covering new code, will be kept as targets of future mutations. GRAMATRON [11] adopts a similar approach. Alternatively, FREEDOM [6], focusing on HTML fuzzing, performs mutations at a more fine-grained level. It mutates the attributes on the DOM tree and the CSS rules referred to by the DOM tree.

2.2. Limitations of Existing Tools

While existing tools running generation-based fuzzing plus structure-aware mutation demonstrated effectiveness in testing document software, they still carry two limitations.

First, they need to build a separate grammar model for each document format. If all is done manually, this can incur extensive labor costs. Using machine learning can reduce such costs. However, machine learning may only learn simpler grammar or partial grammar due to imperfections in algorithms and data. Consider PDF as an example. PDF has a very complex grammar, whose specification requires a document with nearly 1,000 pages [15]. Just displaying the text of "Hello, World" would need a file as complicated as Fig. 1. Even the state-of-the-art ML-based tool [9] can only learn grammar models to generate independent PDF objects instead of complete PDF documents. Another issue of using separate grammar models

⟨doc⟩ ⊨ ⟨pages⟩
⟨pages⟩ ⊨ ⟨page⟩ | ⟨page⟩; ⟨pages⟩
⟨page⟩ ⊨ ⟨size⟩; ⟨objects⟩
⟨objects⟩ ⊨ ⟨object⟩ | ⟨object⟩; ⟨objects⟩
⟨object⟩ ⊨ ⟨text⟩ | ⟨figure⟩ | ⟨table⟩ | ⟨form⟩ | ...
⟨text⟩ ⊨ ⟨position⟩; ⟨font⟩; ⟨data⟩; ⟨ref⟩; ⟨event⟩...
⟨figure⟩ ⊨ ⟨position⟩; ⟨size⟩; ⟨title⟩; ⟨source⟩; ⟨event⟩
⟨font⟩ ⊨ ⟨Helvetic⟩...
...

Figure 2: Grammar of DIR. Other production rules are omitted for the simplicity of presentation.

is that the mutations need to be engineered separately for different document formats.

Second, the existing tools run mutations at a single level. For instance, NAUTILUS [10] does mutations at the coarse-grained, subtree level, and FREEDOM [6] conducts mutations at the fine-grained, attribute level. However, mutations at different levels can bring non-overlapping benefits in covering code and activating bugs. Thus, those tools cannot sufficiently exploit the potential of mutations.

3. Our Approach

In this paper, we propose a new fuzzing scheme to test document software. In principle, our scheme also follows the idea of combining generation-based fuzzing and structure-aware mutations. However, it involves two new techniques — *intermediate document representation* and *multi-level document mutations* — to address the two limitations discussed in Section. 2.2

3.1. Intermediate Document Representation

A document file, no matter which format it follows, can be described in an abstract way that is independent of its actual format. For instance, the document presented in Fig. 1 can be described as:

A document that has one page and displays "Hello, World" at the location of (pos1, pos2) using font type A, font size B, and color black.

The above observation brings us the key insight of creating an intermediate representation, named DIR, for document files. Following the grammar of the DIR, we can generate documents that can be lowered to any desired format. This way, we only need to build a grammar model for the DIR, but we can enable generation-based fuzzing for various formats. Moreover, we can perform many mutations directly on the DIR, avoiding the necessity of re-engineering the mutation components for different document formats.

Design of DIR. DIR is intended to support heterogeneous document formats. Thus, its design should be generic enough to represent various types of documents. To satisfy this requirement, we propose DIR with grammar shown in Fig. 2.

In general, DIR represents a document as a set of `<pages>` where each `<page>` consists of a group of `<objects>`. An `<object>` can be a piece of `<text>`, a `<figure>`, a `<table>`, or other common types of document components (e.g., a fillable `<form>`). Each object is described by its `<attributes>`. Some attributes are shared by all kinds of objects, such as `<position>` of the object on the current page. Other attributes are unique to an object type. For instance, text objects have the `<font>` attribute that other objects do not have. A special attribute is `<event>`, which defines handlers to handle events pertaining to an object (e.g., the handler of clicking a button). The DIR also restricts the attributes of an object based on the object's type. For instance, it disallows adding a `<source>` attribute to a `<text>` object. To better illustrate the idea of DIR, we re-present the document shown in Fig. 1 as a DIR document below:

```
Doc:
Pages: Page1
Page1: Text1
Text1: [Pos: 0, 0]; [Font: Helvetica]; [Size: 11pt];
       [Color: Black]; [Data: ``Hello, World'']
```

DIR has a different design goal from IRs used in previous document fuzzing tools. Consider the FD-IR from FREEDOM [6] as an example. FD-IR is created to describe Document Object Model (DOM) specifically. Thus, its design is heavily tailored toward DOM (e.g., following the DOM structure, encoding CSS rules, and maintaining event handlers). Reusing FD-IR for other document formats, if possible, will mandate heavy extensions or customizations.

3.2. Multi-level Document Mutation

Leveraging the DIR, we will follow the workflow presented in Fig. 3 to fuzz a target document software. The first step is to build a corpus of seed documents. Unlike existing tools [6], [10] that directly build the target documents, we alternatively create documents in the format of DIR, using two different methods. The first method is to follow the DIR grammar to generate DIR documents randomly. The other method is to convert existing documents in any format to DIR documents. At this point, we have developed DIR converters for PDF and HTML.

After building the DIR documents, our scheme will run a series of mutations. Most mutations will be applied directly to the DIR documents, avoiding the need to build the mutation components for every target format. Most importantly, our mutations span multiple granularity levels, which can more thoroughly exploit the benefits of mutations than existing tools.

Page-level mutations work on pages contained in a DIR document. They issue the following operations:

- *Duplicating pages* randomly duplicates pages. Similar to the idea of *random recursive mutation* proposed in [6], the operation will also recursively duplicate a page for a random number of times.
- *Removing pages* randomly removes pages.

- *Reordering pages* perturbs the orders of different pages, such as swapping the first page and the last page.
- *Borrowing pages* resembles the *splicing mutation* used in [10] and [11]. It exchanges pages across DIR documents. The exchanging can be done on individual pages (i.e., exchanging a single page) or a sequence of pages (i.e., exchanging a set of contiguous pages).

In certain cases, an object may cross the boundary of two pages. To avoid breaking the object, we will consider the two pages as a single one during mutations.

Object-level mutations are applied to objects in a single page. The mutations will include all the four operations we designed for pages. In addition, the mutations will further incorporate *object insertions*, where newly generated objects (using approaches like the one presented in [9]) are inserted into random places in the page. Different from pages that are typically independent of each other, objects can have dependencies. For instance, a `<figure>` object may have a child `<text>` object in its title. When running object-level mutations, we will aim to maintain the dependency by adjusting the newly introduced object. For example, when replacing a `<figure>` object with a `<table>` object, we will tailor the `<table>` object to reuse the `<text>` object in the `<figure>`'s title for its own title.

Attribute-level mutations operate on the attributes of a single object in the following ways.

- *Adding attributes* adds attributes to the target object. Strictly following the DIR grammar, the operation can only add attributes that are compatible with the object. To enlarge the mutation space and create unexpectedness, we loosen the grammar to allow adding any attributes to an object when performing this operation.
- *Replacing attributes* replaces an existing attribute with a new one. The new attribute can be generated following the grammar or borrowed from other objects.
- *Deleting attributes* randomly removes an object's attributes.
- *Changing an attribute value* alters the value of an attribute, such as replacing the `<Data>` attribute in a `<text>` object with a longer string.

Lowering DIR to target formats is the step following the above DIR mutations. The goal is to translate a DIR document, derived from generation or mutations, to a target document (e.g., a PDF file). If the lowered document brings new branch coverage in the target software (e.g., PDFIUM), the corresponding DIR document will be added to the seed queue for Post-DIR mutations.

A key challenge in this step is the need for a separate converter for each target format. Building such converters from scratch can be labor-intensive. To tackle this challenge without incurring high manual costs, we exploit the observation that many SDKs exist to process common document files. These SDKs provide interfaces to create and manipulate document files. For better usability, the interfaces provide an abstraction of the underlying format, which often resembles our DIR. For instance, the Foxit PDF SDK [12] provides APIs to insert/delete pages in a PDF document, add/remove objects into pages, and manipulate the attributes of objects. There are also similar SDKs for other popular document formats, such as the Open XML SDK [16] for Word and the API2HTML SDK [17] for HTML. Leveraging these SDKs, we can easily produce a document in the target format by calling the APIs corresponding to the DIR constructs. The example below shows the sequence of Foxit SDK APIs needed to generate the PDF document presented in Fig. 1.

```
APIs:
FoxitSDK * FQL = new FoxitSDK(Linux);
/*Set up the location of the starting page*/
FQL->SetGlobalOrigin(5);
/* Create first page; A dummy page */
FQL->InsertPages(-1, 1);
/* Append 2 pages */
FQL->InsertPages(1, 2);
/* Select 2nd page for creating text */
FQL->SelectPage(2);
/* Add font */
int Font\_ID = FQL->AddStandardFont(3);
/* Select font */
FQL->SelectFont(Font\_ID);
/* Set text color */
FQL->SetTextColor(0.85, 0, 0);
/* Set text size */
FQL->SetTextSize(11);
/* Draw text at position (100, 100) */
FQL->DrawText(100, 100, L"Hello, World");
FQL->SaveToFile(L"hello\_world.pdf");
```

Post-DIR mutations are performed on the target documents lowered from DIR documents. For example, we may simply borrow the mutations together with the dictionaries from AFL/AFL++.

4. Applications

We apply our fuzzing scheme to both PDF and HTML following the workflow presented in Fig. 3. The selection of PDF and HTML is driven by their significance in our daily life. In addition, the high complexity of the two formats, indicated by their reference manual containing hundreds of pages [18], [19], presents a more significant challenge to fuzzing.

4.1. Building Seed DIR Files

We start with building a set of seed DIR files to support follow-up mutations, using two different methods.

Generating DIR Files From Scratch. This method follows the DIR grammar to randomly assemble DIR files.

Extracting DIR Files From Existing Documents. As introduced before, we have built converters from HTML and PDF to DIR. Both converters adopt the same idea: reusing existing HTML/PDF tools to parse a given file and map its structures to DIR structures:

❶ The format of PDF is very close to DIR, which consists of pages, objects, and attributes. We build a parser on top of PyPDF2 [20], an open-source toolkit enabling comprehensive analysis of PDF files.

❷ The design of HTML differs more from DIR. HTML does not have the concept of a page. Thus, we convert every HTML file into a single DIR page and lay out all the objects on that page. Our HTML parser is developed based on Beautiful Soup version 4.9 [21].

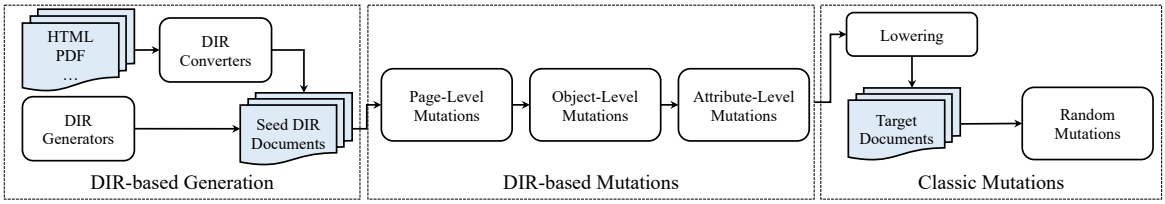

Figure 3: Workflow of our fuzzing scheme.

It should be noted the DIR files generated by PDF/HTML-to-DIR converters can be reused for any other format, and we do not need to create separate converters for different formats.

4.2. DIR-based Mutations

At the high level, DIR-based mutations can simply follow the guidance presented in Section. 3 to derive new DIR documents from the initial corpus. However, applying the guidance in practice can encounter two challenges.

Challenge 1: Getting coverage feedback is very slow. By design, we need to lower every mutated DIR document to a PDF file or HTML file, and measure the code coverage of the PDF or HTML software. This process involves many heavy computations and is very slow. As a result, it significantly reduces the efficiency of mutations and becomes a bottleneck of the fuzzing system.

To address the challenge, we propose an optimization. As demonstrated before, a DIR document can be lowered to a PDF or HTML document through a sequence of SDK API calls. In this sense, the API sequence is another representation of the DIR document. This inspires us to directly mutate the API sequence instead of mutating the DIR document. Once generating an API sequence, we run it and measure its code coverage in the SDK as feedback. If new SDK code is covered, the API sequence is considered valuable and kept for future mutations. This way, we avoid running the lowered PDF/HTML document in the target software for feedback. Technically, we use APIs from the Foxit PDF SDK [12] for PDF and APIs from WebToolKit [13] for HTML. A summary of the utilized APIs can be found on an anonymized page at [22]. We leverage the FRIDA mode [23] offered by AFL++ to support coverage-guided mutations on the APIs. The FRIDA mode supports binary-only instrumentation on code ranges picked at run-time. In our applications, we run the FRIDA mode to only collect coverage of code belonging to the PDF SDK or WebToolKit when executing the APIs.

We run FRIDA to enable two forms of mutations: API sequence mutations and API argument mutations. To enable API sequence mutations, we encode the API sequence as a part of the fuzzer input, where the value of each byte of the input indicates which API to include. By mutating those input bytes, FRIDA can essentially mutate the API sequence. We can use a similar approach to achieve coverage-guided mutations on API arguments. Briefly, we can assign different parts of the fuzzer input to the arguments. This way, FRIDA can automatically mutate the arguments based on code coverage of the corresponding APIs.

Challenge 2: SDK APIs may not support certain fine-grained mutations. The SDK APIs follow deterministic templates to generate PDF or HTML constructs. The APIs may not support the desired mutations in two scenarios.

First, the APIs often enforce internal restrictions to ensure the rationality of the resulting document. For instance, `RotatePage(int32)` from Foxit PDF SDK [12] only accepts an integer argument of 90, 180, or 360. This is reasonable in a general sense since we typically only rotate a PDF page for 90/180/360 degrees. However, this may limit the unexpectedness in the resulting PDF file, potentially hurting the fuzzing effectiveness.

Second, the APIs can only generate complete, valid objects. It cannot support fine-grained mutations such as mixing two heterogeneous objects and adding an incompatible attribute to an object. For instance, we can neither insert a table cell object into a figure object nor add a font attribute to the table cell. This limits the mutation space of fuzzing and the potential of finding bugs.

To overcome the above two limitations, we introduce a set of post-DIR mutations. These mutations directly work on the generated PDF and HTML documents, using the code coverage in the target software as feedback.

4.3. Post-DIR Mutations

PDFobj Mutations. Internally, PDF organizes everything as "objects". To avoid confusion with the concept of object in DIR, we will use *PDFobj* to refer to the internal "objects" contained in a PDF document. More specifically, PDF utilizes PDFobjs to organize all types of constructs (page, object, attributes, etc). For example, line `15 - 23` in Fig. 1 shows a single PDFobj which represents the first page in the PDF file. We apply three fine-grained mutations on the PDFobjs that cannot be completed at the DIR level.

Same-type mutations: PDFobjs can be classified into 8 basic types: boolean, integer and real number, string, name, array, dictionaries, streams, and the null object. Our first mutation respects the type of the target PDFobj. The specific operations are as follows.

- If the PDFobj has a primitive type (boolean, integer and real number, string, or name), we randomly assign a new value of the same type to the PDFobj.
- If the PDFobj has an array type, we randomly resize the array and randomly replace the values in the array.
- If the PDFobj has a stream type (a sequence of bytes that encode multiple PDF objects together), we replace it with another stream object from the current/another document.

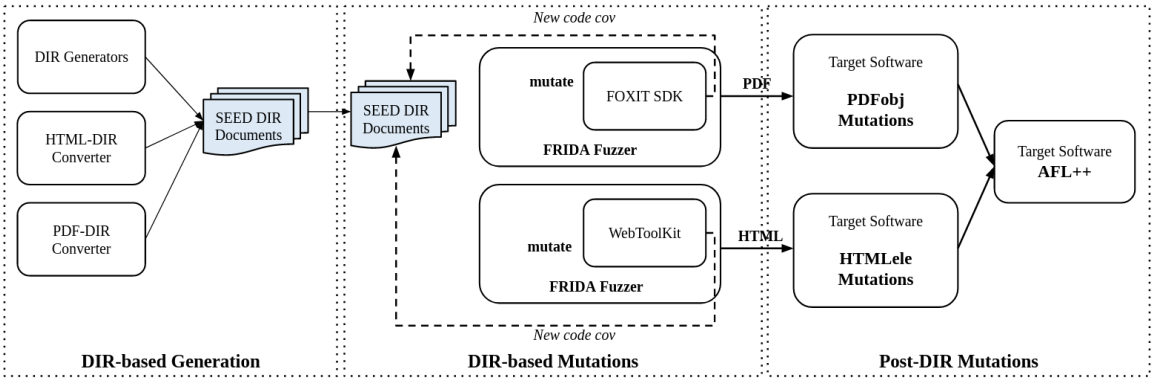

Figure 4: Our Framework of PDF and HTML fuzzing.

- If the PDFobj has a type of null object (i.e., a nonexistent PDFobj), we replace it with a random non-null object picked from the current/another document.

Cross-type mutations: These mutations run similar operations to same-type mutations but without respecting the types (e.g., we may replace an array with a PDFobj of any type).

Type-independent mutations: The last group of mutations is independent of the type of the target PDFobj. They involve three major operations: (i) inserting a random PDFobj picked from the current document or other documents to a random location; (ii) duplicating a PDFobj for a random number of times; (iii) deleting a randomly selected PDFobj.

PDF uses an XRef table to manage all the PDFobjs. The XRef table records the basic information of each PDFobj, such as location and size. Changing a single PDFobj can result in an invalid XRef table, making the PDF document not parsable. This will lead the PDF software to dump the document at an early stage. To handle this problem, we update the XRef table when we finish a mutation to a PDFobj. This will keep the PDF file parsable.

HTMLele Mutations. The structure of all HTML files is organized using elements (HTMLele for short). Each element is defined by a start tag, the content, and an end tag. The elements can be classified into different categories, such as document structure, head, and body elements. An HTML file can be represented as trees of elements. We apply three mutations designed for PDFobj to HTMLele, including same-type mutations, cross-type mutations, and type-independent mutations. The mutations are applied to not only individual elements but also trees of elements.

AFL++ Mutations. At the lowest level, we ran AFL++ on the target software, using PDF documents produced by other mutations as the seed inputs.

4.4. Putting Everything Together

Putting all our components together, we assemble a fuzzing framework called MODIFIER, shown as Fig. 4. Running MODIFIER in practice, we will have to answer two general questions that every mutation-based fuzzer faces: (i) how to schedule the seed DIR documents (or API sequences) and (ii) when to stop the mutation on a specific DIR document. To answer the first question, we design an algorithm to rank the DIR documents. Specifically, we generate the API sequence for each DIR document and calculate the appearance frequency of each API among all the sequences. We further measure the execution time of the API sequence of each DIR document. A DIR document with less-frequently covered APIs and a shorter execution time will receive a higher ranking. To answer the second question, we rely on the quality of a seed DIR document to decide when to stop the mutations. If mutating the DIR document (i.e., the API sequence) does not lead to new code coverage after a certain amount of time, we switch the mutation to the next DIR document. In our experiment, we move on to the next DIR file only after observing a 15-minute period during which there is no new coverage.

5. Evaluation

To understand the utility of our DIR-based fuzzing, we conduct a series of evaluations on MODIFIER. Our evaluation centers around three questions:

- Q1: *Can* MODIFIER *outperform existing fuzzing tools?*
- Q2: *Can multi-level mutations help* MODIFIER*?*
- Q3: *Can* MODIFIER *discover bugs from real-world software?*

5.1. Experimental Setup

Benchmarks. To support our evaluation of PDF and HTML fuzzing, we collect a group of 6 PDF applications and 6 HTML applications. Details are summarized in Table. 1. The applications are widely used and tested in both industry and academia. They also vary in functionality, size, and complexity. To help detect bugs, we enable AddressSanitizer [37] when building these programs.

Seed Inputs. MODIFIER can work with both a single seed input and a large corpus. Thus, we create two different sets of seed inputs. The first set includes a single PDF

TABLE 1: PDF and HTML benchmarks used in our evaluation.

	Applications / Libraries				AFL Settings	
	Name	Version	Driver	Source	Seed(s)	Options
PDF	MuPDF	1.18.0	mutool	[24]	[25]/ [26]	draw @@
	PDFium	commit [27]	pdfium_test	[27]	[25]/ [26]	@@
	Xpdf	4.02	pdftops/text	[28]	[25]/ [26]	@@ /dev/null
	PoDoFo	0.9.7	txtextract	[29]	[25]/ [26]	@@
	QPDF	10.0.4	[30]	[31]	[25]/ [26]	@@
HTML	Chromium	111.0.5535.2	blink_html_tokenizer_fuzzer	[32]	[33]/ [34]	@@
	Chromium	111.0.5535.2	html_preload_scanner_fuzzer	[32]	[33]/ [34]	@@
	Chromium	111.0.5535.2	mhtml_parser_fuzzer	[32]	[33]/ [34]	@@
	Chromium	111.0.5535.2	css_parser_fast_paths_fuzzer	[32]	[33]/ [34]	@@
	HTML Tidy	5.9.20	tidy	[35]	[33]/ [34]	@@
	libxml2	21100	xmllint	[36]	[33]/ [34]	–html @@

TABLE 2: MODIFIER vs. AFL++ & NAUTILUS on PDF fuzzing.

Prog.	Code Coverage Comparison							
	1 Seed File			100 Seed Files				
	AFL++	MODIFIER		AFL++	NAUTILUS	MODIFIER		
	Edge #	Edge #	Increase (%)	Edge #	Edge #	Edge #	Increase vs. AFL++ (%)	Increase vs. NAUTILUS (%)
MUPDF	10503	16280	55.00	16175	17068	22205	37.28	30.10
PDFIUM	29407	31468	7.01	29966	30274	44706	49.19	47.67
XPDF	6512	9620	47.73	10518	11051	11893	13.07	7.62
QPDF	14134	16398	16.01	17781	18013	18767	5.55	4.19
POPPLER	5957	15055	152.73	20214	22053	26555	31.37	20.41
PODOFO	2660	3784	42.26	3885	3908	3996	2.86	2.25
Ave.	11529	15434	33.87	16423	17061	21355	30.03	25.17

file [25] that contains semantic-rich structures (tables, forms, and vector graphics). The second set includes 100 PDF files refined from 10K real-world samples. Specifically, we crawl 10K diverse PDF files from the wild using Google Search. Ranking the PDF files with the approach described in Section. 4.4, we pick the top 100 as the second seed set. For HTML, we also use two sets of seed input, one with a single HTML file [33] and one with 100 HTML files [34]. The 100 HTML files were selected from 10K real-world HTML files, using the same method applied to the PDF files.

Configurations. We run AFL++ [5], Learn&Fuzz [9], and NAUTILUS [10] as baselines for PDF evaluation, and AFL++, FreeDom [6], and Domato [7] for HTML evaluation.

- MODIFIER is tested separately on the two sets of seed inputs. We allocate three CPU cores to respectively run the DIR documentation generation, the DIR-based mutations, and the post-DIR mutations. We further launch 4 AFL++ instances to randomly mutate the generated testcases.
- AFL++ [5] [for both PDF and HTML] is a general-purpose fuzzing tool that can be applied to any software. We run two experiments with AFL++, respectively on the two sets of seed inputs. In each experiment, we launch 7 parallel fuzzing instances on the target program, where each instance is affiliated with a CPU core.
- Learn&Fuzz [9] [for PDF] is the only PDF-focused fuzzing solution that we can identify in the literature. At a high level, Learn&Fuzz is a purely generation-based fuzzing tool. In our evaluation, we re-implemented the algorithm presented in [9] to train a Learn&Fuzz model, using the 10k PDF files we described above as training data. The resulting model is a sequence-to-sequence recurrent-neural-network consisting of 2 layers and 512 hidden units in each layer. We then run the model to generate individual PDF objects and insert those objects into two template PDF files to derive new testcases. The two template files are picked from the training dataset, which have larger code coverage than the remaining seed files. We run the model on 3 parallel CPU cores to generate PDF files.
- NAUTILUS [10] [for PDF] is a tool invented for fuzzing targets that require highly structured inputs. By design, NAUTILUS is very similar to MODIFIER: It applies structure-aware mutation on top of generation-based fuzzing. In the case of fuzzing PDF programs, we adapt the NAUTILUS algorithm following [10] to work on PDF. Specifically, we exchange the PDFobjs across different PDF files. If a PDFobj has child PDFobjs, the child PDFobjs will be exchanged recursively. In our evaluation, we run NAUTILUS on one CPU core and run 6 AFL++ instances in parallel to NAUTILUS. All the testcases generated by NAUTILUS are sent to the AFL++ instances, if the testcases cover new code.
- FreeDom [6] [for HTML] is a Document Object Model generator created specifically for HTML. It shares similarities with MODIFIER in that both rely on a context-aware intermediate representation to describe file formats. However, MODIFIER is more generic, as it can describe other highly structured formats than HTML. When fuzzing HTML programs, FreeDom was configured to generate HTML files for 48 hours on 3 CPU cores.

- DOMATO [7] [for HTML] follows a set of grammars to generate random, valid, or semi-valid structures of HTML pages with CSS and JavaScript objects from scratch. DOMATO was also run for 48 hours on 3 CPU cores.

Environments. We run all our experiments on a 64-core server running Ubuntu 20.04.1 LTS with Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz and 128GB of RAM, following the above settings. We run each experiment for 48 hours and repeat each experiment 10 times. All the results reported in this section are averaged on the 10 times of experiments.

5.2. Comparing with Existing Tools

In this evaluation, we compare MODIFIER with the three baseline tools, using edge coverage as the metric.

Comparing with AFL++ on both PDF and HTML. We compare MODIFIER and AFL++ in two settings: using one seed input and using 100 seed inputs. We observe that MODIFIER can generate excessively large test cases, significantly slowing down AFL++. As such, we exclude test cases over 2MB from the mutations of AFL++. We also do not count those test cases when measuring MODIFIER's code coverage.

Using One Seed File: In PDF evaluation, as shown in Table. 2, MODIFIER covers 33.87% more edges than AFL++. The evaluation of HTML shows similar results. On average, MODIFIER covers 21.27% more edges than AFL++ (see Table. 4).

Using 100 Seed Files: When applied to PDF, MODIFIER covers 30.03% more edges than AFL++ on average. The evaluation of HTML fuzzing shows a similar trend. Averagely, MODIFIER produces 28.83% more edge coverage than AFL++.

Comparing with LEARN&FUZZ on PDF. In 48 hours, LEARN&FUZZ-MODEL generates 18,568 PDFobjs. Injecting the PDFobjs into the two PDF templates described before, we obtain 37,163 valid PDF files. In this part of evaluation, we perform two types of comparison. First, we compare the number of "valuable" PDF files generated by LEARN&FUZZ and the DIR-mutation components of MODIFIER. Specifically, we run AFL-CMIN (AFL-CMIN is a AFL utility to find the smallest subset from a corpus of inputs to cover the same amount of code.) on the PDF files generated by the two tools and then count the number of resulting PDF files. As the result of AFL-CMIN varies across different target programs, we repeat the comparison separately on each benchmark program. Eventually, the 37,136 PDF files generated by LEARN&FUZZ are narrowed down to 6 - 280 PDF files, depending on which benchmark program we consider. In contrast, MODIFIER generated 147,564 PDF files, which were refined to 1,923 - 7,039 PDF files. These results demonstrate that MODIFIER is better at generating PDF files that are valuable to fuzzing.

Second, we compare the number of edges covered by LEARN&FUZZ and the DIR-mutation components of MODIFIER. As shown in Table. 3, MODIFIER outperforms LEARN&FUZZ on every program. On average, MODIFIER covers 127.74% more edges than LEARN&FUZZ. This evaluation shows that MODIFIER has a better generation capability than LEARN&FUZZ, illustrating the utility of our DIR-based mutations.

TABLE 3: MODIFIER vs. LEARN&FUZZ on PDF fuzzing with 100 seed inputs. For MODIFIER, we only considered test cases generated by DIR-based mutations and post-DIR mutations (including those greater than 2MB). **Test cases produced by AFL++ mutations are excluded**.

Programs	Code Coverage Comparison		
	LEARN&FUZZ	MODIFIER	
	Edge #	Edge #	Increase (%)
MUPDF	16268	34380	111.35
PDFIUM	34610	77871	125.00
XPDF	14841	32091	116.23
QPDF	23359	61212	162.05
POPPLER	21602	48816	125.98
PODOFO	4244	7592	78.89
Ave.	19171	43660	127.74

Comparing with NAUTILUS on PDF. As aforementioned, NAUTILUS and MODIFIER both run structure-aware mutations atop generation-based fuzzing. This evaluation compares the edge covered by [NAUTILUS and AFL++] and [MODIFIER and AFL++] in 48 hours. As shown in Table. 2, MODIFIER presents a better performance than NAUTILUS on every PDF program. On average, MODIFIER covers 25.17% more edges than NAUTILUS. Further, we can observe that MODIFIER presents a bigger advantage over NAUTILUS on larger software. For instance, MODIFIER outperforms NAUTILUS the most on the largest program (i.e., PDFIUM) and the least on the smallest program (i.e., PODOFO). This brings evidence that our multi-level mutations performs better than the single-level mutation adopted by NAUTILUS.

Comparing with FREEDOM on HTML. FREEDOM generated 487,831 HTML files in 48 hours. We run AFL-CMIN on the outputs to determine the number of "valuable" HTML files. Depending on the HTML program, the 487,831 HTML files were refined to 191 - 6,398 HTML files. In comparison, MODIFIER generated 311,566 HTML files in 48 hours, which were narrowed down to 388 - 9,579 HTML files.

Further, we compared the edge coverage of FREEDOM and MODIFIER. In this evaluation, we only considered the test cases generated by MODIFIER's DIR-based mutations and post-DIR mutations. All test cases produced by AFL++ are ignored. The results, as presented in Table. 5, show that MODIFIER consistently outperforms FREEDOM on all programs, with coverage of 13.02% more edges on average.

Comparing with DOMATO on HTML. DOMATO generated 500,065 HTML files within 48 hours. After running AFL-CMIN, this corpus was reduced to 56 - 6,302 HTML files, depending on which target program we run. In contrast, MODIFIER generated 311,566 HTML files, which were AFL-CMINed to 388 - 9,579 HTML files.

A comparison was also made between the edges covered by DOMATO and MODIFIER. Again, we only

TABLE 4: MODIFIER vs. AFL++ on HTML fuzzing.

Programs	Code Coverage Comparison					
	1 Seed File			100 Seed Files		
	AFL++	MODIFIER		AFL++	MODIFIER	
	Edge #	Edge #	Increase (%)	Edge #	Edge #	Increase (%)
HTML_PRELOAD_SCANNER_FUZZER	59097	68984	16.73	63576	76651	20.57
BLINK_HTML_TOKENIZER_FUZZER	50689	53083	4.72	55200	66129	19.80
MHTML_PARSER_FUZZER	40261	47694	18.46	46703	60361	29.24
CSS_PARSER_FAST_PATHS_FUZZER	37894	42599	12.42	40513	47702	17.75
TIDY	50053	66871	33.60	51150	73575	43.84
XMLLINT	32466	46009	41.71	33959	50611	49.04
Ave.	45077	54206	21.27	48517	62505	28.83

TABLE 5: MODIFIER vs. DOMATO and FREEDOM on HTML fuzzing with 100 seed inputs. For MODIFIER, we only considered test cases generated by DIR-based mutations and post-DIR mutations (including those greater than 2MB). The two columns under Increase (%), from left to right, represent the increase of code coverage brought by MODIFIER to DOMATO and FREEDOM, respectively.

Programs	Code Coverage Comparison				
	DOMATO	FREEDOM	MODIFIER		
	Edge #	Edge #	Edge #	Increase (%)	
HTML_PRELOAD_SCANNER_FUZZER	60039	62311	76177	26.88	24.62
BLINK_HTML_TOKENIZER_FUZZER	60569	65089	63700	5.17	1.57
MHTML_PARSER_FUZZER	54803	55609	57872	5.60	4.07
CSS_PARSER_FAST_PATHS_FUZZER	43206	44017	45918	6.28	4.32
TIDY	58976	58031	67062	13.71	15.56
XMLLINT	35377	36029	46110	30.34	27.98
Ave.	52162	53514	59473	14.02	13.02

count edge coverage of MODIFIER's DIR-based mutations and post-DIR mutations. All test cases produced by AFL++ are disregarded. The results, presented in Table. 5, demonstrate that MODIFIER consistently outperformed FREEDOM on all programs, producing an average 14.02% increase in edge coverage.

The comparison with both FREEDOM and DOMATO illustrates the advantages of MODIFIER over purely generation-based fuzzing. Guided by its structure-aware and multi-level mutations, MODIFIER can generate more valuable test cases and contribute to higher code coverage.

5.3. Contribution of Multi-level Mutations

Our comparison between MODIFIER and NAUTILUS shows that our multi-level mutation outperforms the single-level mutation of NAUTILUS. In this part of evaluation, we aim to understand the contribution of mutations at different levels. Specifically, we calculate the unique edge coverage (An edge is considered unique only when it is never covered by more than one level of mutations.) brought by each level of mutations in the 48-hour experiments.

The results are summarized in Table. 6. Evidently, each level of mutation brings unique contributions. Specifically, DIR-based mutation contributes the most unique edges in HTML fuzzing. In PDF fuzzing, each level's contribution varies across the programs. DIR-based mutations contribute the most in MUPDF fuzzing, while Post-DIR mutations contribute the most in POPPLER fuzzing. Other than those, AFL++ mutations bring the most unique coverage edges.

5.4. Finding Bugs

In the evaluation described above, MODIFIER triggers **8,973** total unique crashes in the PDF and HTML programs according to AFL++'s metric. After triaging the crashes based on the bug type and location reported by AddressSanitizer [37], we manually analyze the results and identify **16** previously unknown bugs (summarized in Table. 7). All the bugs have been reported to the developers. In addition, as we can observe from Table. 7, MODIFIER finds more bugs than AFL++, LEARN&FUZZ, and NAUTILUS. More importantly, all the bugs discovered by the three tools are also covered by MODIFIER.

To verify that the finding of the bugs is indeed attributed to the strategies of MODIFIER instead of AFL++ alone, we construct the lineage of the inputs triggering the bugs (i.e., we trace how the inputs are derived from the seed inputs and the MODIFIER-generated test cases). We find that the first input triggering every bug is directly generated by MODIFIER or mutated from PDF/HTML files generated by MODIFIER. This demonstrates that the finding of the bugs indeed benefits from the strategies of MODIFIER.

Case Study. To further illustrate how the mutation strategies of MODIFIER can help find bugs, we present a case study. In List. 1, we show the PDFobjs corresponding to a set of form fields organized in a parent-child relation. Specifically, the "10 0 obj" is the parent field, while "11 0 obj" and "12 0 obj" are both child fields. List. 2 shows a code snippet from XPDF that handles the form fields in List. 1. The code will recursively call the `scanField` function to handle the child fields. When our

TABLE 6: Unique edge coverage by mutations at different levels. In this evaluation, **test cases larger than 2MB are included**.

	Programs	Settings					
		DIR-based Mutations		Post-DIR Mutations		AFL++ Mutations	
		1 Seed	100 Seeds	1 Seed	100 Seeds	1 Seed	100 Seeds
PDF	MUPDF	4582	4745	314	603	908	1996
	PDFIUM	529	759	252	1208	3694	4590
	XPDF	72	125	379	1016	605	1994
	QPDF	57	67	234	1392	507	666
	POPPLER	217	492	1272	4920	837	2318
	PODOFO	56	326	117	187	155	166
HTML	BLINK_HTML_TOKENIZER_FUZZER	1798	2116	676	1076	77	101
	HTML_PRELOAD_SCANNER_FUZZER	308	1582	273	2306	201	792
	MHTML_PARSER_FUZZER	1350	5011	490	3361	137	532
	CSS_PARSER_FAST_PATHS_FUZZER	1498	1757	395	1607	176	302
	TIDY	767	1505	1055	1169	359	563
	XMLLINT	1553	2090	578	723	197	201

TABLE 7: Unique crashes/bugs discovered by different fuzzing tools. The bugs are accumulated from all 10 experiments. DOMATO and FREEDOM did not trigger crashes.

	Prog.	AFL++		Nautilus		Learn & Fuzz		Modifier		Bug Types
		Crash	Bug	Crash	Bug	Crash	Bug	Crash	Bug	
PDF	MUPDF	0	0	0	0	0	0	632	3	Heap Buffer Over-read NULL Pointer Dereference
	XPDF	85	1	589	3	12	1	4,572	7	Heap Buffer Over-read NULL Pointer Dereference Stack Exhaustion
	POPPLER	0	0	0	0	0	0	362	1	Stack Buffer Over-read
	PODOFO	736	1	901	1	0	0	2,063	2	Heap Buffer Over-read NULL Pointer Dereference
HTML	TIDY	0	0	–	–	0	0	344	3	Memory Leak NULL Pointer Dereference
	Total	**821**	**2**	**1,490**	**4**	**12**	**1**	**8,973**	**16**	—

DIR-mutations duplicate the child field a large number of times, too many recursions of `scanField` will happen and result in a stack-exhaustion bug. As shown in the case, triggering this bug requires duplicating the correct structure of the child field quite a few times, which is hard to be achieved by random mutations like AFL++.

Listing 1: Illustration of form fields organized in a parent-child structure.

```
10 0 obj /* Parent : Radio button field */
<< /FT /Btn
  ... ...
 /Kids [ 11 0 R 12 0 R ]
>>
endobj

11 0 obj /* Child : First checkbox */
<< /Parent 10 0 R
 ... ...
>>
endobj

12 0 obj /* Child : Second checkbox */
<< /Parent 10 0 R
   ... ...
>>
endobj
```

Listing 2: Simplified version of a buffer overflow in XPDF

```
void AcroForm::scanField(Object *fieldRef) {
   /* create Objects */
   Object fieldObj, kidRef, kidObj
   ... ...
   for (i = 0; i < kidsObj.arrayGetLength(); ++i) {
      ... ...
      /* recursively call scanField */
      scanField(&kidRef);

      kidRef.free();
   }
   /* The two objects will never be freed if the kid
         array is too long */
   kidsObj.free();
   ... ...
   fieldObj.free();
}
```

6. Related Works

6.1. Generation Based Fuzzing

Generation-based fuzzing can produce highly structured inputs for real-world applications. Three main approaches for structured test case generation are commonly used.

Manually summarizing grammar rules. Generation-based fuzzers require well-written grammar rules prior to generating test cases. Examples of such fuzzers, designed for producing syntax-correct HTML files, include DOMATO [38], FREEDOM [6], and DOMFUZZ [39]. DOMFUZZ also employs a grammar-based splicing technique, which inspires our hierarchy object exchanging method. For fuzzing JavaScript codes, techniques like [40], [41], and [42] use random generation or combination of code based on provided syntax rules. Favocado [43] generates syntactically correct binding code for fuzzing JavaScript engines, using semantic information.

Grammar generation with machine learning. Learn&Fuzz [9] is a generation-based fuzzer that leverages machine learning to learn the grammar rules of PDF objects. However, it only generates random PDF objects and fails to capture the complexities of other elements in the PDF format, such as header, Xref, and trails. Skyfire [8] uses a context-sensitive grammar model with a probabilistic ML algorithm for fuzzing HTML and XSL files. DEPPFUZZ [44] employs a generative Sequence-to-Sequence model for C code generation, and Godefroid et al. [45] implement a dynamic test case generation algorithm for fuzzing IE7's JavaScript interpreter.

IR assisted generation. PolyGlot [46] is a fuzzing framework that creates high-quality test cases for different programming languages by using a uniform immediate representation (IR). Unlike other generation based fuzzing frameworks, PolyGlot uses grammar for mutation instead of pure seed generation, allowing for better code coverage. However, PolyGlot is limited by the requirement for a BNF grammar, and can still generate syntactically incorrect test cases due to inconsistent grammar inputs.

6.2. Mutation Based Fuzzing

Mutation-based fuzzing distinguishes itself from generation-based fuzzing by necessitating less human effort. However, the success of this approach is highly contingent on the availability of a high-quality input corpus. Many of the mutation-based fuzzing tools, following the footsteps of AFL [4], begin with a collection of initial seeds and then devise new test cases drawing from the feedback received during the execution of the target program.

Various methods can be applied to boost the efficacy of mutation-based fuzzing. One common approach is to refine the process of seed scheduling, mutation, and distribution, taking into account code coverage feedback. Another perspective is to explore more effective feedback. AFL [47] considers code branches as feedback, which is refined by Steelix [48], CollAFL [49], and PTrix [50] to incorporate extra control-flow information. More aggressively, TaintScope [51], Vuzzer [52], GREYONE [53], REDQUEEN [54], and Angora [55] exploit feedback informed by data flow.

Mutation-based fuzzing has been extended to the testing of document software, including software dealing with PDFs. For instance, Feldmann [56] used 220 unique paths—narrowed down from an initial set of 80,000 PDF files—as the input to mutation-based fuzzing. This was an attempt to uncover vulnerabilities in software like FOXITREADER, PDFXCHAN-GEVIEWER, and XnView. Alon and Ben-Simon [57] have also worked on mutation-based fuzzing for PDF applications, using WINAFL on manually created fuzzing harnesses derived from the JP2KLIB.DLL library in Adobe Reader.However, this conventional mutation-based fuzzing struggles with preserving the structure of complex file formats, leading to generated seeds failing early-stage syntax checks during program execution. In contrast, MODIFIER selectively filters the most fuzzing-friendly and diverse files from a large corpus. Thanks to its use of intermediate representation, our file collection isn't limited to just PDF formats.

7. Conclusion

This paper presents a new scheme to facilitate fuzzing on document software. The approach involves an intermediate document representation, called DIR, to describe document files in various mainstream formats. By further applying multi-level mutations on the DIR and lowering the DIR document to a desired format using common SDKs, our approach can use the same set of DIR-based tools to enable fuzzing on various document formats without intensive human efforts. Applying our DIR-based approach to PDF and HTML, we showcase that the approach can work well on complicated document formats and outperform the existing tools. Our source code has been released at https://github.com/junxzm1990/hierachy-fuzzing, which can be reused for further research in the same direction.

8. Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was supported by National Science Foundation (NSF) awards CNS-2213727 and OAC-2319880. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government or NSF.

References

[1] iTEXT, "Do you know how many pdf documents exist in the world?" https://itextpdf.com/en/blog/technical-notes/do-you-know-how-many-pdf-documents-exist-world, 2022.

[2] CVE, "Adobe acrobat reader : List of security vulnerabilities," https://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-497/Adobe-Acrobat-Reader.html, 2020.

[3] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[4] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/technical_details.txt, 2014.

[5] "Afl++," https://github.com/AFLplusplus/AFLplusplus, 2022.

[6] W. Xu, S. Park, and T. Kim, "Freedom: Engineering a state-of-the-art dom fuzzer," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 971–986.

[7] I. Fratric, "Domato," https://github.com/googleprojectzero/domato, 2017.

[8] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 579–594.

[9] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 50–59.

[10] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *Proceedings of 2019 Network and Distributed System Security Symposium(NDSS)*, 2019.

[11] P. Srivastava and M. Payer, "Gramatron: effective grammar-aware fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 244–256.

[12] "Foxit pdf sdk libraries," https://developers.foxit.com/, 2022.

[13] "Webtoolkit libraries," https://github.com/emweb/wt/archive/4.9.1.tar.gz, 2022.

[14] C. Miller, Z. N. Peterson *et al.*, "Analysis of mutation and generation-based fuzzing," *Independent Security Evaluators, Tech. Rep*, vol. 4, 2007.

[15] J. Preface By-Warnock and C. Preface By-Geschke, "Pdf reference: Adobe portable document format version 1.4," *Version 1.4*, 2001.

[16] "Welcome to the open xml sdk 2.5 for office," https://docs.microsoft.com/en-us/office/open-xml/open-xml-sdk, 2021.

[17] "Convert raw api data into beautiful webpages," https://api2html.com/, 2018.

[18] "Portable document format," https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/PDF32000_2008.pdf, 2008.

[19] "Html standard," https://html.spec.whatwg.org/print.pdf, 2023.

[20] "Home page for the pypdf2 project," http://mstamy2.github.io/PyPDF2/, 2011.

[21] "Beautiful soup documentation," https://www.crummy.com/software/BeautifulSoup/bs4/doc/, 2021.

[22] "Modifier operations and mutations," https://drive.google.com/drive/folders/1cxIQW7haegP1OWfBV6VKciQOk9ChX4H6?usp=sharing, 2023.

[23] https://github.com/AFLplusplus/AFLplusplus/tree/stable/frida_mode, 2022.

[24] A. Software, "Mupdf," https://github.com/ArtifexSoftware/mupdf/commit/0e90241161, 12 2020.

[25] "Testcase of pdf," https://www.irs.gov/pub/irs-pdf/f1040.pdf, 12 2020.

[26] "Testcases of pdf," https://anonymous.4open.science/r/hierachy-fuzzing-5573/testcases/100_seed/, 12 2020.

[27] Google, "Pdfium," https://pdfium.googlesource.com/pdfium/+/refs/heads/chromium/2021, 12 2021.

[28] G. . Cog, "Xpdfreader," https://dl.xpdfreader.com/xpdf-tools-linux-4.02.tar.gz, 12 2020.

[29] "Podofo," https://sourceforge.net/projects/podofo/files/podofo/0.9.7/, 12 2020.

[30] Jay Berkenbilt and Thorsten Schöning, "Wrapper of qpdf-fuzzer," https://github.com/qpdf/qpdf/blob/78b9d6bfd4cbd3e947b1c5ffe73eb97b040e312a/fuzz/qpdf_fuzzer.cc, 12 2020.

[31] Jay Berkenbilt, "Qpdf," https://github.com/qpdf/qpdf/commit/78b9d6bfd4cb, 12 2020.

[32] Google, "The official github mirror of the chromium source," https://github.com/chromium/chromium.git, 1 2023.

[33] "Testcases of html," https://anonymous.4open.science/r/hierachy-fuzzing-5573/testcases/ONE_HTML/9929-reno.html, 12 2022.

[34] "Testcases of html," https://anonymous.4open.science/r/hierachy-fuzzing-5573/testcases/HTML_testcase/, 12 2022.

[35] HTACG, "tidy-html5," https://github.com/htacg/tidy-html5.git, 1 2022.

[36] "libxml2 source code," https://github.com/GNOME/libxml2.git, 12 2022.

[37] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[38] Symeon, "Grammar based fuzzing pdfs with domato," https://rb.gy/gi4cbz, 2020.

[39] Mozilla Fuzzing Security, "Dom fuzzers," https://github.com/MozillaSecurity/domfuzz, 2 2019.

[40] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1629–1642.

[41] M. F. Security, "A collection of fuzzers in a harness for testing the spidermonkey javascript engine." https://github.com/MozillaSecurity/funfuzz, 2 2019.

[42] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[43] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé, and Y. Shoshitaishvili, "Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases," in *NDSS*, 2021.

[44] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 1044–1051.

[45] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2008, pp. 206–215.

[46] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, "One engine to fuzz 'em all: Generic language processor testing with semantic validation," *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 642–658, 2021.

[47] AFL, "Afl-tmin," https://github.com/google/AFL/blob/master/afl-tmin.c, 2015.

[48] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 627–637.

[49] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 679–696.

[50] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *Proceedings of the 2019 ACM on Asia Conference on Computer and Communications Security*. ACM, 2019.

[51] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy (SP)*, 2010, pp. 497–512.

[52] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *Proceedings of 2017 Network and Distributed System Security Symposium(NDSS)*, vol. 17, 2017, pp. 1–14.

[53] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data flow sensitive fuzzing," in *Proceedings of the 29th USENIX Security Symposium*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[54] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *Proceedings of 2019 Network and Distributed System Security Symposium(NDSS)*, vol. 19, 2019, pp. 1–15.

[55] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725.

[56] S. Feldmann, "Fuzzing closed source pdf viewers," https://rb.gy/91rmy3, 2019.

[57] Y. Alon and N. Ben-Simon, "50 cves in 50 days: Fuzzing adobe reader," https://rb.gy/1w3veo, 2018.