Computers & Security 129 (2023) 103178

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Generation-based fuzzing? Don't build a new generator, reuse!

Chengbin Pang[a,*], Hongbin Liu[b], Yifan Wang[c], Neil Zhenqiang Gong[b], Bing Mao[a], Jun Xu[d]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
[b] Duke University, NC 27705, United States
[c] Stevens Institute of Technology, Hoboken 07030, United States
[d] University of Utah, Salt Lake City 84112, United States

ARTICLE INFO

Article history:
Received 3 March 2022
Revised 21 September 2022
Accepted 7 March 2023
Available online 15 March 2023

Keywords:
Structure-aware fuzzing
Vulnerability mining
Pdf fuzzing
Generator
Converter

ABSTRACT

Generation-based fuzzing is effective in testing programs that require highly structured inputs. However, building a new generator often requires heavy manual efforts to summarise a large body of grammar rules to generate correct structures.

In this paper, we introduce the idea of generator reuse, aiming to avoid the manual efforts required to build new generators. Our key insight is that for a format X (e.g., PDF) whose grammar rules are not yet supported by existing generators, we can often find generators that support the grammar rules of a different format Y (e.g., HTML) and converters between Y and X (e.g., HTML-to-PDF converters). Reusing the generators for Y and the converters, we can effortlessly assemble a generator to support the grammar rules of X (e.g., given an HTML generator and an HTML-to-PDF converter, we can connect them to form a PDF generator).

To explore the validity of our idea of generator reuse, we apply the idea to build a PDF generator, reusing a popular HTML generator and a set of mainstream HTML-to-PDF converters. Throughout the application, we also gain an initial understanding of the limitations of our generator reuse idea and introduce a set of strategies aiming to mitigate those limitations. We evaluated Arcee using 6 infrastructural, popular PDF applications and libraries. Our empirical results show that the PDF generator is indeed useful: running it on the 6 applications and libraries for 2 weeks, we discovered **39** bugs, **28** of which are new bugs and **23** of which have explicit security implications. Our empirical results also show that the PDF generator is a better tool than existing PDF fuzzers: it outperforms the fuzzing tools that can be applied to PDF software (using code coverage and the number of discovered bugs as metrics).

1. Introduction

Generation-based fuzzing (Manès et al., 2019; Sutton et al., 2007) is an effective technique to test programs which require highly-structured inputs (e.g., HTML, XML, PDF, etc.). It works by following pre-defined "grammar" rules (Fratric, 2017; Godefroid et al., 2017; Wang et al., 2017; Xu et al., 2020) to synthesize structurally correct test cases.

To build an effective generation-based fuzzer, the key is to construct a rich set of grammar rules. Generally speaking, there are two major approaches to constructing the grammar rules: *learning such rules from existing data* (Godefroid et al., 2017; Wang et al., 2017) or *summarising such rules with manual efforts* (Fratric, 2017; Xu et al., 2020). However, the first approach may be less effective because the rules can be fairly complex and learning may fail to capture them. As such, in practice, the second approach is preferred.

To use the second approach, a common (and usually the biggest) barrier is the heavy manual efforts required to summarise the grammar rules and engineer them into generative code. For instance, to build a Portable Document Format (PDF) generator, one needs to interpret its 1000-page description document (Preface By-Warnock and Preface By-Geschke, 2001) and program the thousands of grammar rules inside, which is labor intensive. In this paper, we explore an idea aiming to "bypass" this barrier, which can be applied in various cases. Our key observation is that for a format **X** whose grammar rules are not yet supported by any existing generator, we may possibly find generators that support the grammar rules of a different format **Y** and converters between the grammar rules of **Y** and **X**. Reusing the generators of **Y** and the converters, we can effortlessly assemble a "generator" supporting the grammar rules of **X**.

* Corresponding author.
E-mail address: binpangsec@smail.nju.edu.cn (C. Pang).

https://doi.org/10.1016/j.cose.2023.103178
0167-4048/

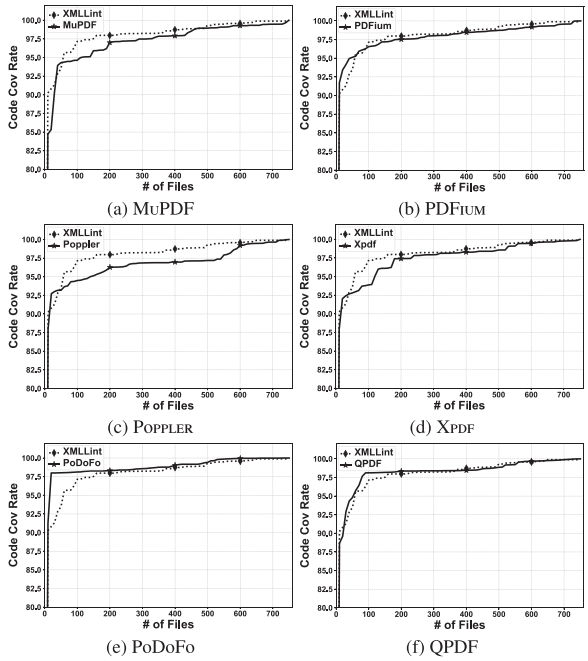


Fig. 1. Increase of code coverage of HTML files generated by DOMATO in XMLLINT vs. increase of code coverage of PDF files converted from the HTML files in various PDF applications (we only considered HTML files that bring new code coverage in XMLLINT and the baseline for "Code Cov Rate" is the accumulated code coverage of all the HTML/PDF files). These results illustrate that when the generated HTML files carry diversity, the converted PDF files inherit the diversity.

While our idea seems to avoid the manual efforts, its effectiveness is unclear and (more importantly) its limitations are unclear. To establish a first-step understanding about these two matters, we apply our idea to building a PDF generator. We target PDF for several reasons: (i) PDF software (e.g., PDFium Viewer in Chrome Google, 2020) has been a part of our daily life. It is important to develop tools like fuzzers to find bugs in PDF software; (ii) PDF is complex, which has a large body of grammar rules, as we pointed out above; (iii) we found no generators that already summarised PDF grammar rules; (iv) we could identify a group of HTML generators (e.g., DOMATO Fratric, 2017, FREEDOM Xu et al., 2020) and a group of HTML-to-PDF (H2P) converters (e.g., WKHTMLTOPDF Kulkarni and Truelsen, 2020 and CHROME in headless mode Bidelman, 2019), meeting the pre-conditions of our idea. For the simplicity of presentation, we will refer to our PDF generator as ARCEE[1].

Specifically, ARCEE reuses a popular HTML generator, DOMATO (Fratric, 2017), to produce HTML files and then leverages mainstream H2P converters to convert the HTML files into PDF files. After running ARCEE under experimental settings, we found that it indeed generates valid PDF files. Moreover it seems to properly inherit the diversity created by the HTML generator, as illustrated by Fig. 1. However, after further dissecting the results, we also identified several limitations of ARCEE. First, the H2P converters largely follow templates to convert HTML tags into PDF objects. The templates tend to produce similarly-structured PDF objects, which puts a restriction on the possible diversity ARCEE may create. Second, the H2P converters convert HTML files to valid PDF files, which lack the unexpected/abnormal structures to trigger bugs.

[1] ARCEE is borrowed from the name of a Female Autobot Transformer, who is one of the best sharpshooters on record.

As a follow-up step, we extended initial efforts towards mitigating the above limitations. To increase the diversity in the converted PDF files, we take a two-fold action. On the one hand, we include more H2P converters. By doing so, we diversify the templates used to convert the HTML tags (❶) and, as such, we may obtain more diverse PDF objects. On the other hand, we exchange objects across PDF files converted by different H2P converters (❷). This way we introduce higher diversity into the combination of objects. To create unexpected/abnormal structures, we further run mutation-based fuzzing, using the PDF files after our diversification as seed inputs. In this step, we found that ARCEE brings more limitations (i) the PDF files produced by H2P converters are often very large. This hurts the execution speed (and thus efficiency) of the mutation-based fuzzing; (ii) the PDF files produced by H2P converters broadly contain compressed objects, which are vulnerable to mutations and become barriers to the effectiveness of mutations. Accordingly, we incorporate two strategies aiming to tackle those limitations: ❸ we shrink the size of the converted PDF files by trimming less-useful tags from the HTML files, removing redundant objects from the PDF files, and replacing large objects with smaller ones; ❹ we unpack compressed objects in the PDF files to help improve the effectiveness of the mutation-based fuzzing.

To more systematically understand the effectiveness of ARCEE and the effectiveness of our strategies to address ARCEE's limitations, we implemented ARCEE on top of DOMATO, two H2P converters (WKHTMLTOPDF Kulkarni and Truelsen, 2020 and CHROME in headless mode Bidelman, 2019), and AFL (Zalewski, 2014). We evaluated ARCEE with 6 infrastructural, popular PDF applications and libraries. Our results show that ARCEE outperforms the existing fuzzing tools that are applicable to PDF software. Running ARCEE on the 6 applications and libraries for 2 weeks, we discovered **39** bugs. **28** of the them are new bugs and **23** of them have explicit security implications. These demonstrate that ARCEE is a useful fuzzer for PDF software and more importantly, showcase the utility of our generator-reusing idea. Our results also show that each of our strategies (❶–❹) indeed mitigates the limitations of ARCEE to certain extent. For instance, ❷ and ❹ can bring an increase of code coverage by 10% and 6.7%, respectively. We envision that those limitations are shared by generator-reuse in general and thus, our strategies should also help other applications of our generator-reuse idea. It is worth mentioning that we have received $7K of bug bounty award from Google and ARCEE has been invited to participate in the Chrome Fuzzer Program (Google, 2020).

In summary, we make the following contributions.

- We bring up the idea of generator reuse. It can help reduce the manual efforts in building generators to support new grammar rules.
- We showcase the feasibility of our idea of generator reuse by applying it to build a PDF generator. We demonstrate the utility of this idea through extensive evaluation of our PDF generator.
- We identify the limitations of our generator reuse idea through the case study of the PDF generator. We include strategies to mitigate those limitations and demonstrate the effectiveness of our strategies.
- We implement our PDF generator and release it at https://github.com/bin2415/ARCEE.

In the rest of this paper, we focus on the application of our generator-reuse idea to build ARCEE. In §2, we give background about the format of PDF and why summarising PDF grammar rules is burdensome. In §3, we explain the design of ARCEE, followed by more implementation details in §4. In §5, we present the evaluation of ARCEE. §6 discusses our limitations and §7 summarizes the related work. We conclude this paper in §8.

```
1  %PDF-1.4 /*header*/
2  1 0 obj /*root object*/
3  <<
4    /Type /Catalog
5    /Pages 2 0 R
6  >>
7  endobj
8  2 0 obj /*list of pages*/
9  <<
10   /Type /Pages
11   /Count 1
12   /Kids [3 0 R]
13 >>
14 endobj
15 3 0 obj /*page object*/
16 <<
17   /Type /Page
18   /Parent 1 0 R
19   /MediaBox [0 0 614 794]
20   /Contents 4 0 R
21   /Resources 5 0 R
22 >>
23 endobj
24 4 0 obj /*stream object*/
25 <<
26   /Length 58
27 >>
28 stream
29 BT
30 /F0 1 Tf
31 12 0 0 12 10 750 Tm
32 (Hello, World) Tj
33 ET
34 endstream
35 endobj
36 5 0 obj /*font object*/
37 <<
38   /ProcSet [/PDF]
39   /Font <<
40     /F0 6 0 R
41   >>
42 >>
43 endobj
44 6 0 obj /*font content*/
45 <<
46   /Type /Font
47   /Subtype /Type1
48   /BaseFont /Helvetica
49 >>
50 endobj
51 xref /*xref table*/
52 0 6
53 0000000000 65535 f
54 0000000009 00000 n
55 0000000062 00000 n
56 0000000125 00000 n
57 0000000239 00000 n
58 0000000343 00000 n
59 0000000412 00000 n
60 trailer /*trailer section*/
61 <<
62 /Size 6
63 /Root 1 0 R
64 >>
65 startxref
66 428
67 %%EOF
68 @
```

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

2. Background

2.1. Portable document format

Portable Document Format (PDF), developed by Adobe in 1993, is a file format to present a wide variety of contents (e.g., texts, images, hyperlinks, etc.) in a manner independent of the applications, the operating systems, and the hardware (Bienz et al., 1993). In the following, we will use the example in Fig. 2 to explain the structure of PDF.

The PDF file starts with a header section giving basic information such as its version ("PDF-1.4" in `line 1`). The header is followed by a body section consisting of *objects* (`line 2–50`). These objects, which are typed, keep all the data shown to the user, such as text streams, images, multimedia elements, etc.Each object consists of two numbers (e.g., "1 0" in `line 2`) and a body encapsulated between "obj" and "endobj" (e.g., `line 3–6`). The two numbers respectively represent the identification of the object and the revision number of the object (i.e., how many times the object has been revised). Coming after the objects is a *xref table* (`line 51–59`), providing references to the list of objects in the document. The first line (`line 52`) in the x-ref table gives identification of the first object in the list ("0") and total number of objects in the list ("6"). Each following line in the x-ref table represents one object,[2] containing three fields respectively showing the offset of the object in the file, the generation number of the object (i.e., how many times the object has been recycled), and whether the object is free or in use ("f" or "n"). It is worth noting that since PDF version 1.5, xref table is not mandatory and it can be represented by special objects. At the end of the PDF file, a *trailer* section (`line 60–64`) records the total number of objects ("6" in `line 62`) and a reference to the *root object* in the format of "id-of-obj gen-of-obj R" ("1 0 R" in `line 63`). After that is the keyword *startxref* followed by the offset of the xref table in the file ("428" in `line 66`).

When rendering the example PDF file, PDF software first parses the xref table to identify all the objects and then parses the trailer section to determine the root object (object 1 in `line 3–6`). Following the root object, PDF software recursively processes other referenced objects. In our example, the root object gives a reference to object 2 (spanning `line 9–13`), which contains an array of references to page objects or precisely, objects representing the pages to display. Our example PDF file only has one page object, which is object 3 spanning `line 16–22`. The page object provides metadata such as parent object of the page (`line 18`) and size of the page (`line 19`), followed by references to objects of the contents to be displayed in this page (object 4 in `line 25–34`) and references to objects of resources used by the contents (object 5 in `line 37–42`). Object 4 is a stream object consisting of a set of commands: ❶ "BT" (`line 29`) and "ET" (`line 33`) respectively indicate the start and end of texts to display; ❷ "Tf" (`line 30`) indicates the texts will use font "F0"; ❸ "Tm" (`line 31`) indicates the position to display the texts in the current page; ❹ "Tj" (`line 32`) indicates the texts to display are "Hello, World". Object 5 is a reference to a font object (object 6 in `line 45–49`) with label "F0" and type "Helvetica".

[2] The first object — `line 53` in Fig. 2 — simply represents the head of the object list, presenting no actual data.

2.2. Why summarising PDF grammar rules is burdensome

To build a PDF generator, a pretty standard approach is to adopt the idea of the Domatoproject (Fratric, 2017). Domatofollows manually-summarized HTML/CSS "grammar" rules to assemble structurally correct HTML files. Replacing the HTML/CSS grammar rules with PDF's, we can adapt Domatoto generate PDF files. However, in comparison to HTML, summarizing PDF grammar rules is much more burdensome.

First, we can describe HTML with context-free grammars. Consider the PDF file in Fig. 2 as an example. An HTML file like the following can display identical contents:

```
1 <html>
2 <body>
3 <p style="font-family:Helvetica">Hello, World</p>
4 </body>
5 </html>
```

To support the generation of the HTML file, we just need a few context-free production rules (newlines are suppressed):

```
1 <html> = <lt>html<gt><body><lt>/html<gt>
2 <body> = <lt>body<gt><bodyelements><lt>/body<gt>
3 <bodyelements> = <HTMLParagraph>
4 <HTMLParagraph> = <lt>p <font><gt><texts><lt>/p<gt>
5 <font> = font-family: Helvetica/Georgia/Times
6 <texts> = *
```

In contrast, we need grammars with certain context-sensitivity to describe PDF. For instance, to describe the production rule from object 3 (a page object) to object 4 (a content object) in Fig. 2, we need to describe the context of locations (i.e., the content object must lie in the page object). The requirement of context-sensitivity increases the complexity, not even mentioning that we have to extend Domato's language to support context sensitivity.

Second, PDF has a more complicated design than HTML. The most direct evidence is that the description of PDF (e.g., version 1.4) requires a document with nearly 1000 pages (Preface By-Warnock and Preface By-Geschke, 2001) while the description of HTML (version 2.0) only needs a 100-pages document (Berners-Lee and Connolly, 1995). Without a doubt, handling PDF will incur much higher workload and will require much deeper domain knowledge.

3. Build a PDF generator via generator reuse

In this section, we apply our generator-reuse idea to build Arcee, a generation-based PDF fuzzer. Fig. 3 shows the workflow. In the following, we elaborate on the key steps.

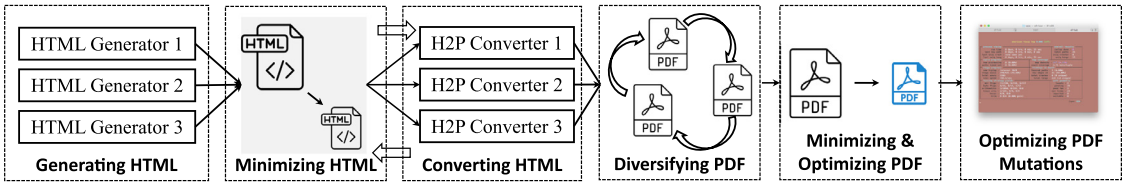


Fig. 3. Workflow of Arcee.

3.1. Generating HTML files

Any HTML generator could be assembled into Arcee, such as generation-based generators (Symeon, 2020; Wang et al., 2017) and mutation-based generators (Xu et al., 2020). By default, Arcee runs Domato, the aforementioned rule-based generator, to generate HTML files. Instead of running Domatoas it is, we enhanced it as follows.

We observe that Domatocan generate various types of HTML tags. However, it tends to result in limited diversity in the same type of tags. To this end, we combine Domatowith Learn&Fuzz (Godefroid et al., 2017), a deep learning based algorithm that can generate very diverse (despite less valid), same-typed tags. First, we followed the algorithm presented in Godefroid et al. (2017) to train the Learn&Fuzz model, using 20 K HTML files randomly crawled via Google Search interfaces as training data. The resulting Learn&Fuzz model is a sequence-to-sequence recurrent-neural-network, consisting of 2 layers and 512 hidden units in each layer. Second, we run the Learn&Fuzz model to generate individual HTML tags and use them to replace the same-typed ones in the HTML files generated by Domato. To avoid replacing all the Domato-generated tags, we follow a certain probability to determine whether or not to replace an HTML tag.

Arcee uses the enhanced Domatoin a **batch mode**. In each batch, Arcee runs Domatoto generate a certain number of HTML files and then sent the HTML files together to follow-up steps.

3.2. Converting HTML files

At the high level, this step is simply to run existing H2P converters to convert the HTML files into PDF files. However, we found that H2P converters largely follow templates to convert HTML tags to PDF objects, producing similarly patterned PDF objects. This often limits the diversity in the PDF files (**limitation I**). To mitigate this limitation, we take two actions.

First, we include more H2P converters to convert the same HTML files, such that we have more templates and thus, higher diversity. By intuition, Arcee should run all the available H2P converters. However, we found that many H2P converters do not bring much benefit while introducing side effects. For instance, the PDF files converted by Pandoc (Pandoc, 2020) do not increase the code coverage in the PDF software, compared to the PDF files converted by Wkhtmltopdf (Kulkarni and Truelsen, 2020) and Chrome in headless mode (Chrome-Headless) (Bidelman, 2019). In addition, the PDF files converted by Pandoc are much larger, reducing the execution speed of the follow-up mutation-based fuzzing. The observations inspired us to pick H2P converters that meet one of the following two conditions: ❶ the PDF files produced by the converter cover new code in the PDF software, in comparison to other converters; ❷ the PDF files produced by the converter are smaller than PDF files produced by other converters, even if the PDF files bring no much new code coverage.

Following the above criteria and using branch coverage as the metric, we identified four H2P converters, including Wkhtmltopdf, Chrome-Headless, Foxit PDF Toolkit (Harris, 2017), and Acrobat PDF Converter (Adobe, 2020). We further excluded Foxit PhantomPDF and Acrobat PDF Converter by default, because the former is commercial while the latter has no batch mode.

Second, we adapted the *splicing mutation* techniques proposed in Aschermann et al. (2019a) to further increase the diversity in the converted PDF files. Assuming in a batch of HTML generation, and H2P conversion, we obtain t PDF files. We then sample p unique pairs from the t PDF files, following a uniform probability distribution to pick PDF files for each pair. Given a pair of PDF files (PDF_1, PDF_2), we exchange objects of the same type between PDF_1 and PDF_2, creating two more PDF files, PDF_3 and PDF_4. We further fix the x-ref tables, the trailer sections, and all other affected objects (e.g., stream objects) in PDF_3 and PDF_4 to maintain their validity. If PDF_3 (PDF_4 similarly) brings new code coverage in the target PDF software, we keep it together with PDF_1 and PDF_2 for the next stage. Otherwise we discard it. The types of objects we consider for exchange include Font, Image, Form, Pattern, PostScript, and Shading.

3.3. Creating unexpected/abnormal structures

Directly applying the converted PDF files (after diversification) as test cases, we discovered few bugs in popular PDF software. The reason, according to our analysis, is that the H2P converters produce valid PDF files, which usually lack unexpected/abnormal structures to trigger bugs (**limitation II**). To create unexpected/abnormal structures in the converted PDF files, we explore a straightforward approach where we consider the converted PDF files as seed inputs and run mutation-based fuzzing on them. Following a common practice, we can first distill the converted PDF files (using tools like AFL-cmin[3]) and then run mutation-based fuzzers (such as AFL) on the remaining PDF files. However, we observed more limitations brought by the HTML generator and the H2P converters while we try to adopt this common practice. In the following, we elaborate on the limitations and our attempts to mitigate them.

3.3.1. Minimizing converted PDF files

The HTML generator we use mostly generate HTML tags in a random manner. It often produces many less-useful tags, leading to large PDF files and hurting the execution speed of mutation-based fuzzing. The H2P converters can worsen the situation because they often use large objects and insert dummy objects (**limitation III**). To mitigate this limitation, we use two approaches.

Trimming HTML files. Our first approach is to trim the HTML files without hurting the code coverage of their PDF counterparts in the target PDF software. To trim an HTML file, an intuitive approach is to recursively remove byte sequences, following the strategy of AFL-tmin (AFL, 2015).[4] However, the approach is ineffective

[3] In our problem context, AFL-cmin picks the subset of smallest PDF files that have the same code coverage as the original set of PDF files

[4] AFL-tmin works in an error-and-trial manner. It recursively deletes each consecutive K bytes from a test case. If the deletion hurts the code coverage of the test case, it revokes the deletion. It typically repeats this process to enumerate different K values.

because it easily breaks the HTML structures and fails the converting process. Alternatively, we use an approach inspired by hierarchical delta debugging (Misherghi and Su, 2006).

Given a target PDF software S_{pdf}, we run the HTML generator for one batch and generate k HTML files. We convert each HTML file with both H2P converters into 2 PDF files. We further run AFL-CMIN on all the PDF files, producing k' PDF files whose code coverage in S_{pdf} is denoted as Cov. For each HTML file converted to one of the k' PDF files, we parse the HTML file into a DOM tree and recursively cut the sub-trees in a top-down manner. Once we cut a sub-tree, we convert the resulted HTML file into a new PDF file and recalculate the code coverage of the new PDF file and the remaining $k'-1$ PDF files, notated as Cov'. If $Cov' = Cov$, we remove the sub-tree from the HTML file. Otherwise, we proceed with other sub-trees. We repeat the trimming process until no more sub-trees can be cut. In many cases, we found that removing the entire sub-tree can hurt code coverage, while shortening the values of many text-based attributes (e.g., `href=URL`) does not. Accordingly, we also recursively replace the values of text-based attributes with shorter ones (e.g.,href="https://www.google.com" → href="a.com") when handling a sub-tree.

Our structure-aware trimming recursively processes each sub-tree and its attributes, which is time-consuming. We use several heuristics to reduce the time cost. First, we skip HTML files whose initial converting time exceeds a threshold. Second, we stop trimming an HTML file once the processing time reaches a threshold. *Shrinking PDF files* In this round of minimization, our key insights are (i) H2P converters often insert dummy objects, for goals such as descriptions of the PDF creator; (ii) when no fonts already exist, H2P converters tend to insert self-prepared font objects (which often have large font definitions). We take two actions accordingly. First, we recursively delete objects from a PDF file. After we delete an object, we fix the x-ref table, the trailer sections, and other affected objects. If deleting an object hurts the overall code coverage of PDF files in the current batch, we revoke the deleting operation. In many cases, the deleted object is referenced by other objects. We do not attempt to fix this because we observed that PDF software typically ignores non-existing objects when resolving references to them. Second, we collect all the font objects in a PDF file and shrink them. Fonts in PDF files typically belong to standard fonts (Wikipedia contributors, 2021), *Type1* fonts, and *TrueType* fonts (Preface By-Warnock and Preface By-Geschke, 2001). Standard fonts are described by the PDF standard and no font definitions are needed in the PDF file. In contrast, *Type1* and *TrueType* fonts must include font definitions in the PDF file, which are often encoded as large-sized objects. For each *Type1* or *TrueType* font in a PDF file, we replace it with the smallest font of the same type, following a probability proportional to size of the font. Again, if the replacement of a font hurts the code coverage of PDF files in the current batch, we revoke the replacement operation.

Note that it requires expert knowledge of PDF format to modify PDF files. Specifically, to shrink PDF files, we need to learn the specifications of font inside PDF (section 5.1 on pages 388–396 of Incorporated (2006)). It is still more efficient than constructing PDF grammar rules manually that needs to learn more than 1000 pages of specifications and abstracting them into rules.

3.3.2. Optimizing converted PDF files

The H2P converters also tend to compress stream objects, bringing higher difficulties to the mutations (**limitation IV**). We mitigate this issue by decompressing stream objects in the PDF files. By intuition, the strategy increases the size of the PDF files, offsetting the effects of our previous minimization strategies. However, it may not necessarily slow down the mutation-based fuzzing since we essentially avoid the decompressing operations at fuzzing time. In addition, we run the decompressing in a probabilistic manner to reduce the increase of file size: the smaller the decompressed object is, the higher probability we follow to do the decompressing. By adjusting the probability, we can limit the increase of size under a certain threshold.

4. Implementation

We have implemented ARCEE with around 1200 lines of C code, 1600 lines of Python code, and 500 lines of Bash Scripts. We release ARCEE at https://github.com/bin2415/ARCEE.

Generating HTML files Our implementation relies on DOMATOand LEARN&FUZZ to generate HTML files. For DOMATO, we simply reused its public code (Symeon, 2020). To build the LEARN&FUZZ model, we re-used the public code of DeepFuzz (Liu et al., 2019), which adopts the same learning algorithm of LEARN&FUZZ to train a sequence-to-sequence model for generating single lines of C code. We built the generator that combines Domato and LEARN&FUZZ from scratch, with help of the ANTLR4 library (Parr, 2016).

Converting and minimizing HTML files We reused WKHTMLTOPDF and CHROME-HEADLESS to convert HTML files into PDF files, respectively using command line *wkhtmltopdf* <*input*> <*output*> and *google-chrome –headless –disable-gpu* <*input*> –*print-to-pdf*=<*output*>. For better efficiency, our implementation sets 3 s as the timeout limit when converting an HTML file. To minimize an HTML file, we reused the BEAUTIFUL SOUP project (Bea, 2004) to parse an HTML file into a DOM Tree and then trim sub-trees.

Diversifying, minimizing, and optimizing PDF files We reused the MUPDF library (Software, 2020) to process PDF files. API wise, we used *fz_new_context* and *pdf_open_document* to parse PDF files and extract objects, *pdf_dict_put_drop* to replace objects by removing the old ones and adding the new ones, and *_deleteObject* (from the Python bindings of MUPDF) to delete an object. To fix the xref table and the trailer section, we used API *save(pdfname,garbage)* by setting the *garbage* argument to 1. Finally, we reused MUTOOL shipped with the MUPDF library to help decompress stream objects.

5. Evaluation

In this section, we present our evaluation of ARCEE, centering around three questions.

- Is ARCEE a useful PDF fuzzing tool? Can ARCEE find bugs from PDF applications?
- Is ARCEE a better PDF fuzzing tool? Can ARCEE outperform existing PDF fuzzing tools?
- Can our strategies presented in §3 help mitigate the limitations associated with generator reuse?

5.1. Experimental setup

To support our evaluation, we collected a group of 6 popular PDF applications (or libraries), listed in Table 1. The applications vary in functionality, size, and complexity. The applications are also widely integrated into infrastructural projects (e.g., PDFIUMin CHROMIUM (Chromium, 2020), MUPDFin WEBOS (LG, 2009), POPPLERin KONQUEROR (KDE, 2016) and REKONQ (KDE, 2009), etc.). To facilitate the detection of bugs, we enabled ASan (Serebryany et al., 2012) when building the PDF applications.

We configured ARCEE following our designs presented in §3 to generate and mutate PDF files. While running AFL, we enlarged its bitmap from 64 KB to 256 KB (which is also applied to all other baselines where AFLis used). This is because the PDF applications contain a large number of control flow edges, which can easily lead to collisions in the default bitmap. We ran all our evaluation on

Table 1
PDF applications / libraries used in our evaluation.

	Applications/libraries			AFL settings	
Name	Version	Driver	Source	Seeds	Options
MuPDF	1.17.0	mutool	Software (2020)	Dor1s (2020)	draw @@
PDFium	commit Google (2020)	pdfium_test	Google (2020)	Dor1s (2020)	@@
Poppler	20.12.1	pdftohtml/ps	freedesktop (2020)	Dor1s (2020)	@@ /dev/null
Xpdf	4.02	pdftops/text	Cog (2020)	Dor1s (2020)	@@ /dev/null
PoDoFo	0.9.6	txtextract	Sourceforge (2020)	Dor1s (2020)	@@
QPDF	10.0.4	Jay Berkenbilt and Thorsten Schning (2020)	Jay Berkenbilt (2020)	Dor1s (2020)	@@

AWS c5a.8xlarge instances (32-core AMD EPYC 7R32 @ 3.266 GHz, 64 GB of RAM, Ubuntu-18.04).

5.2. Finding bugs in PDF software

We ran Arcee on the applications in Table 1 for **2 weeks**, where we allocated **4 cores** for PDF generation and **12 cores** for parallel AFLinstances (6 AFL instances with PDF dictionary and 6 without). In this testing, we set up Arcee as follows: (i) we ran Arcee to generate 5,000 HTML files and converted each HTML file into 2 PDF files using Wkhtmltopdf and Chrome-Headless; (ii) we applied splicing mutation (recall §3.2) to 4**x** random pairs of the converted PDF files (**x** is the total number of PDF files in each batch); (iii) we performed our HTML trimming, PDF shrinking, and PDF optimization (recall §3.3) to the PDF files after splicing mutation; (iv) we performed a round of AFL-cmin after PDF optimization; (v) we ran AFLusing the PDF files after AFL-cmin as seed inputs.

In total, Arcee triggered **28,550** crashes that are considered unique by AFL. We first triaged the crashes based on the bug type and the bug location reported by ASan and then analyzed their root causes. As summarized in Table 2, the crashes were caused by **39** bugs, spanning all the 6 applications. **28** of the bugs have never been reported and **23** of them have explicit security implications (i.e., stack/heap buffer overwrite/over-read, use after free, and uninitialized memory dereference). We have reported all bugs to the developers: 39 bugs have been confirmed or patched and 4 of them have received bug bounty awards or CVE numbers. The results reflect that Arcee is a useful fuzzer to discover bugs in PDF software.

To verify that the finding of the PDF bugs was truly attributable to the strategies of Arcee instead of AFLalone, we constructed the lineage of the inputs triggering the bugs (i.e., we traced how the inputs were gradually mutated from the converted PDF files and the initial AFLseed). We found that, in 38 out of the 39 cases, the first input triggering the bug was derived from both converted PDF files and AFL's seed. In the other case, the first input triggering the bug was simply a converted PDF file. This demonstrates that finding of all the bugs is indeed attributable to Arcee.

Table 2
Bugs found by Arcee.

Target	#	Bugs	New	Status
MuPDF	1	Use After Free	√	Patched
	2	Use After Free	√	Patched
	3	Heap Buffer Over-write	√	Patched
	4	Heap Buffer Over-write	√	Patched
	5	Heap Buffer Over-read	√	Patched
	6	Heap Buffer Over-read	√	Patched
	7	Heap Buffer Over-read	√	Patched
PDFium	8*	Heap Buffer Over-read	√	Patched
	9*	Heap Buffer Over-read	√	Patched
QPDF	10	Use After Free	√	Patched
	11	Heap Buffer Over-write	√	Patched
Poppler	12*	Heap Buffer Over-write	√	Patched
	13*	Heap Buffer Over-read	√	Patched
	14	Heap Buffer Over-read	√	Patched
	15	NULL Pointer Dereference	√	Patched
	16	Stack Exhaustion	√	Patched
	17	Stack Exhaustion	√	Patched
PoDoFo	18	Heap Buffer Over-read	√	Confirmed
	19	NULL Pointer Dereference	√	Patched
	20	Division by Zero	√	Patched
	21	Unhandled Exception	√	Confirmed
	22	Heap Buffer Over-read	✗	Confirmed
	23	Heap Buffer Over-read	✗	Confirmed
Xpdf	24	Heap Buffer Over-read	√	Confirmed
	25	Heap Buffer Over-read	√	Confirmed
	26	Null Pointer Dereference	√	Confirmed
	27	Null Pointer Dereference	√	Confirmed
	28	Division by Zero	√	Confirmed
	29	Unhandled Exception	√	Confirmed
	30	Stack Exhaustion	√	Confirmed
	31	Heap Buffer Over-read	✗	Confirmed
	32	Heap Buffer Over-read	✗	Confirmed
	33	Integer Overflow	✗	Confirmed
	34	Uninitialized Memory Dereference	✗	Confirmed
	35	Division by Zero	✗	Confirmed
	36	Stack Exhaustion	✗	Confirmed
	37	Stack Exhaustion	✗	Confirmed
	38	Stack Exhaustion	✗	Confirmed
	39	Stack Exhaustion	✗	Confirmed

* indicate bugs that earn bug bounty rewards or receive CVE numbers.

5.3. Comparing with existing PDF fuzzers

In this evaluation, we compared Arcee with four fuzzing tools that can be applied to PDF software:

- **The Learn&Fuzz PDF generator presented in** Godefroid et al. (2017). The Learn&Fuzz algorithm was initially proposed to generate PDF files and this is the only PDF fuzzer we identified in the literature. Specifically, we followed the algorithm presented in Godefroid et al. (2017) to train a Learn&Fuzz model to generate PDF files, using PDF files converted from the 20K HTML files we described in § 3.1 as training data. For simplicity, we will refer to this model as Learn&Fuzz-pdf. By default, Learn&Fuzz-pdfgenerates individual PDF objects. To form complete PDF files, we injected each PDF object into two PDF template files (the templates were picked from the training data and they have the largest code coverage in the training dataset). The injection followed (Godefroid et al., 2017): we replaced the last object of a template file with a newly generated PDF object, incremented its generation number, and fixed the xref table as well as the trailer section.
- **AFL** (Zalewski, 2014). We ran AFL using the seeds listed in Table 1.
- **MOpt** (Lyu et al., 2019). We also ran MOptbecause it reported more PDF bugs than other public fuzzing tools (see Lyu et al., 2019). We used the seeds listed in Table 1 when running MOpt.
- **AFL-crawl**. We further ran AFLwith a large-corpus of seeds from the wild. We call this setting AFL-crawl.

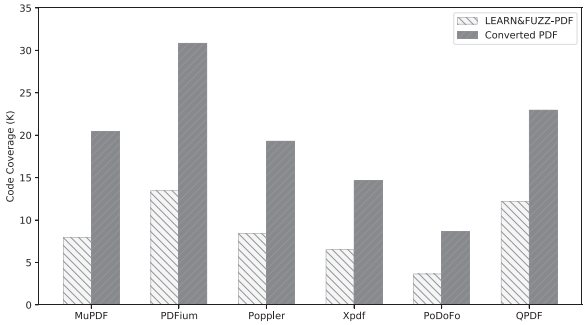


Fig. 4. Code coverage of PDF files generated with LEARN&FUZZ-PDF*v.s.* code coverage of PDF files converted from HTML files generated by our enhanced DOMATO.

Comparing with LEARN&FUZZ-PDF In this comparison, we ran the enhanced DOMATOto generate 20,000 HTML files and converted them to 40,000 PDF files (using our two H2P converters). We also ran LEARN&FUZZ-PDFto generate 20,000 PDF objects and injected them into the two template PDF files to generate 40,000 PDF files.

We first measured the pass rate of the two sets of PDF files, i.e., how many of the PDF files are structurally correct. To be specific, we ran `mutool draw <pdf file>` on each PDF file and checked if `mutool` reported error messages that indicate incorrect structures. Overall, the pass rate of the converted PDF files is multiple times higher than LEARN&FUZZ-PDF(97.17% vs.14.50%). The major reason is that PDF is more complex and its objects follow very diverse patterns. Learning PDF structures is, therefore, more difficult. We then compared the code coverage by the two sets of PDF files. We show the results in Fig. 4. On average, the code coverage (using branch coverage as the metric) by the converted PDF files is 2.28 times as high as the code coverage of LEARN&FUZZ-PDF. The results demonstrate that ARCEE (even without the mutation-based fuzzing) outperform LEARN&FUZZ-PDF.

Comparing with AFL , MOPT , *and* AFL-CRAWL In this evaluation, we compare ARCEE with AFL, MOPT, and AFL-CRAWL. To run AFL-CRAWL, we randomly crawled 20,000 PDF files from Google Search (with a size limit of 2 MB) and ran AFLusing the PDF files as seed inputs (after AFL-CMIN). To run ARCEE, we used DOMATOand the H2P converters to randomly generate 20,000 PDF files, and then processed the PDF files with our strategies (splicing mutation with 4**x** PDF pairs, HTML minimization, PDF minimization and PDF optimization). We finally used the resulting PDF files as seed inputs to run AFL. In all tests, we ran 12 AFLinstances (6 with PDF dictionary and 6 without) for 24 h.

We repeated the evaluation 5 times and report the average results in Table 4 (the number of bugs is aggregated from all the 5 runs). We also show the increase of code coverage across the 24 h in Fig. 5. There are several major observations.

1. AFLand MOPTcovered much less code and only triggered a smaller number of crashes in XPDF, despite the use of PDF dictionary. This is not surprising since mutations alone are less effective to produce correct PDF structures.
2. PDF files crawled from the wild covered more code in POPPLERand PoDoFo, while PDF files generated by ARCEE covered more code in the remaining four applications (see the column of "Code Cov Initial" in Table 3). This result shows that ARCEE can generate PDF files with diversity comparable to (or even higher than) PDF files in the wild.
3. In cases where crawled PDF files have higher initial code coverage, ARCEE reduced the code-coverage gap (calculated as "AFL-CRAWL– ARCEE") at the end of the 24-h test (gap with POPPLER: 2.5 K → 1.9 K; gap with PoDoFo: 2.3 K→ 0.9 K). In cases where PDF files generated by ARCEE have higher initial code coverage, ARCEE enlarged the code-coverage gap (calculated as "ARCEE – AFL-CRAWL") at the end of the 24-h test (gap with MUPDF: 3.1 K → 8.0 K; gap with PDFIUM: 4.8 K → 8.6 K; gap with XPDF0.6 K → 2.0 K; gap with QPDF: 0.5 K → 1.2 K). The results demonstrate that the PDF files generated by ARCEE are more friendly to mutations. In particular, they have much smaller size in comparison to PDF files crawled from the wild (22.8 KB vs.91.5 KB), which benefits the execution speed of AFL. Moreover, they have a much lower ratio of compressed stream objects than PDF files crawled from the wild (59.2% vs.97.5%), which benefits the effectiveness of AFL.
4. ARCEE triggered more crashes and more bugs than AFL-CRAWL, even when ARCEE covered less code (see the results of POPPLERand PoDoFo). This is because the HTML generator used by ARCEE generates rare structures, which are less encountered during initial tests and thus, more likely to trigger hidden bugs. We finally note that all the bugs triggered by AFL-CRAWLwere also triggered by ARCEE, despite this is not guaranteed in principle.

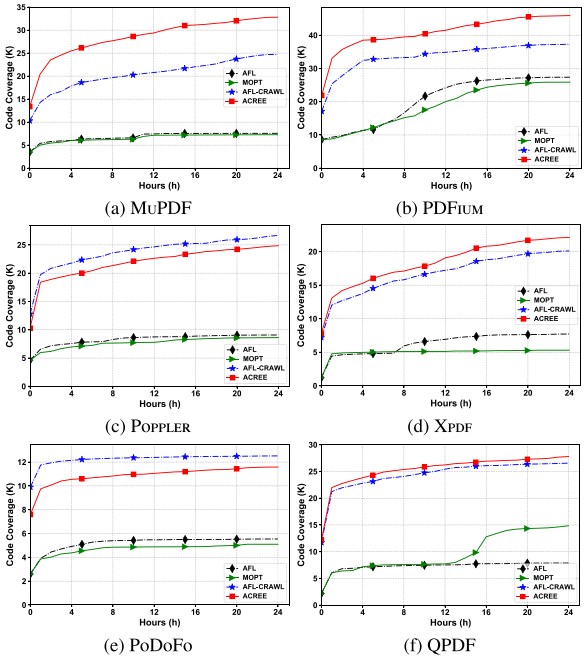


Fig. 5. Code coverage of different tools in a 24-h test.

5.4. Effectiveness of individual strategies

In this evaluation, we separately measure whether the strategies we presented in §3 help mitigate the related limitations

Effectiveness of using multiple H2P converters We use multiple H2P converters to convert each HTML file. We performed an evaluation to verify that every H2P converter is helpful. Specifically, we separately ran WKHTMLTOPDF and CHROME-HEADLESS to convert the HTML files we obtained in the above evaluation (20,000 HTML files from the HTML generator). We then measured the code coverage of PDF files produced by different H2P converters. We show the results in Fig. 6. On average, WKHTMLTOPDF and CHROME-HEADLESS uniquely brought 10.49% and 17.81% of all the code coverage. This shows that both converters are helpful and CHROME-HEADLESS has a better utility.

Effectiveness of splicing mutation We exchange objects across different PDF files to increase their diversity. To measure the effectiveness of the strategy, we applied it to PDF files produced by our evaluation of PDF minimization. We then measured the increase of code coverage this strategy brought. Considering that the number

Table 3
Comparison of different fuzzing tools.

Metrics	Tools	Applications / Libraries					
		MuPDF	PDFium	Poppler	Xpdf	PoDoFo	QPDF
# of Seeds	AFL	1	1	1	1	1	1
	Mopt	1	1	1	1	1	1
	AFL-crawl	826	288	296	148	227	440
	Arcee	1038	1465	1051	1064	529	885
Ave. Size (KB)	AFL	0.2	0.2	0.2	0.2	0.2	0.2
	Mopt	0.2	0.2	0.2	0.2	0.2	0.2
	AFL-crawl	59.7	87.6	105.4	133.6	112.8	49.9
	Arcee	21.2	19.9	24.3	22.8	18.8	30.0
Speed (Exes/Sec)	AFL	369.4	82.6	126.5	119.5	574.4	213.8
	Mopt	279.1	55.9	104.5	100.4	440.6	164.1
	AFL-crawl	134.3	54.4	81.7	65.2	272.4	23.9
	Arcee	154.8	61.7	96.2	69.3	398.2	31.7
Code Cov Initial (K)	AFL	3.5	8.6	4.6	1.2	2.6	2.2
	Mopt	3.5	8.6	4.6	1.2	2.6	2.2
	AFL-crawl	10.3	17.1	12.8	7.2	9.9	11.7
	Arcee	13.4	21.9	10.3	7.8	7.6	12.2
Code Cov Final (K)	AFL	7.6	27.4	9.0	7.7	5.5	7.9
	Mopt	7.3	25.9	8.6	5.3	5.1	14.8
	AFL-crawl	24.7	37.3	26.7	20.1	12.5	26.6
	Arcee	32.7	45.9	24.8	22.1	11.6	27.8
# of Crashes	AFL	0	0	0	2,161	0	0
	Mopt	0	0	0	1,832	0	0
	AFL-crawl	3	0	25	3,181	718	0
	Arcee	127	0	28	4,052	2,929	0
# of Bugs	AFL	0	0	0	1	0	0
	Mopt	0	0	0	1	0	0
	AFL-crawl	1	0	1	4	3	0
	Arcee	2	0	1	5	3	0

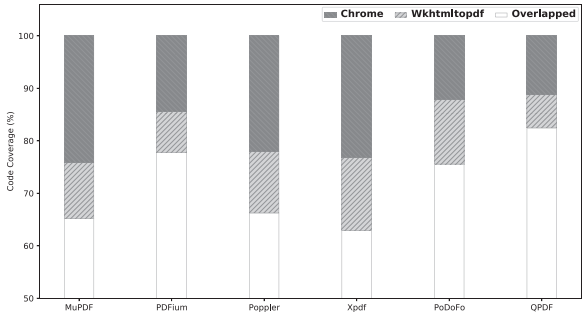


Fig. 6. Ratio of unique code coverage by different H2P converters. Baseline is the total code coverage by both converters. `Overlapped` indicates code covered by both converters.

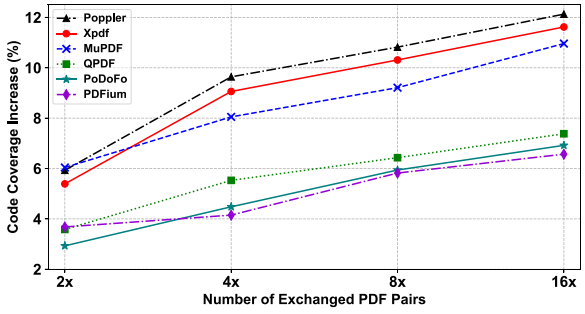


Fig. 7. Increase of code coverage brought by applying object exchange to different number of PDF pairs, where **x** is the total number of PDF files used for object exchange.

of PDF pairs where we apply the object exchange could make a difference, we repeated the evaluation with different number of PDF pairs. We show the results in Fig. 7. Our object exchange indeed increased the code coverage, indicating it enriches the diversity in the PDF files. Moreover, the increase of code coverage grows with the number of PDF pairs. When we applied object exchange to 16**x** PDF pairs, the code coverage increased by 10% on average.

Effectiveness of HTML minimization To reduce the size of the converted PDF files, Arcee adopted a strategy to trim less helpful tags from the initial HTML files. We evaluated the effectiveness of the strategy. Given PDF files produced from our evaluation of different H2P converters, we ran AFL-cmin on them and then applied our HTML minimization to HTML files that correspond to the remaining PDF files.

We show the results in Table 4. First, the H2P converters significantly increased the file size. In our evaluation, the two converters increased the average size of files from 14.5 KB (HTML files) to 37.6 KB (PDF files). Second, our HTML minimization reduced the average PDF size from 37.6 KB to 32.5 KB, bringing a reduction rate of 13.6%. This demonstrates that our HTML minimization is helpful, in particular considering that we only enabled partial HTML minimization for better efficiency.

Effectiveness of PDF minimization Besides HTML minimization, Arcee adopts two other strategies to reduce the PDF size, including trimming redundant objects and replacing large fonts from the PDF files. Given PDF files produced by the above evaluation of HTML minimization, we applied the two strategies to the resulted PDF files. We separately measured the effectiveness of the two strategies and show the results in Table 4. Both strategies are effective. By deleting redundant objects from the PDF files, we averagely reduced the size by 20.6%; By replacing the large fonts from the PDF files, we averagely reduced the size by 23.6%. Putting our HTML minimization and the two strategies together, we reduced the PDF file size by 47.3% (the average size was reduced from 37.6 KB to 19.8 KB).

Effectiveness of PDF optimization To improve the effectiveness of mutations, we optimize PDF files by decompressing the stream objects. We evaluated the strategy. To be specific, we ran AFLwith 12 parallel instance (6 with PDF dictionary and 6 without) for 24 h, using PDF files produced by our evaluation of PDF minimization

Table 4
Effectiveness of strategies to reduce the size of PDF files. `PDF Size` and `PDF Size Δ` respectively indicate the average size of PDF files before and after application of the corresponding strategy. `Reduce Rate` is calculated as $1 - \frac{\texttt{PDF Size } \Delta}{\texttt{PDF Size}}$. To calculate `Reduce Rate` for `Together`, we use `PDF Size Δ` after `Replace Font` and `PDF Size` before `HTML Trim`.

Strategy	Metrics	Applications / Libraries					
		MuPDF	PDFium	Poppler	Xpdf	PoDoFo	QPDF
Initial Data	# of HTML	1,711	2,386	1,775	1,769	895	1,351
	HTML Size (KB)	14.5	15.2	14.5	14.1	13.1	15.6
HTML Trim	PDF Size (KB)	38.0	40.3	37.9	36.2	33.0	40.0
	PDF Size Δ (KB)	32.1	34.1	35.2	31.5	28.3	33.7
	Reduce Rate	15.5%	15.4%	7.1%	13.0%	14.2%	15.8%
PDF Trim	PDF Size (KB)	32.1	34.1	35.2	31.5	28.3	33.7
	PDF Size Δ (KB)	23.3	23.9	29.8	25.3	20.9	31.9
	Reduce Rate	27.4%	29.9%	15.3%	19.7%	26.1%	5.3%
Replace Font	PDF Size (KB)	23.3	23.9	29.8	25.3	20.9	31.9
	PDF Size Δ (KB)	18.0	17.2	20.9	19.5	16.7	26.2
	Reduce Rate	22.7%	28%	29.9%	22.9%	20.1%	17.9%
Together	Reduce Rate	52.6%	57.3%	44.9%	46.1%	49.4%	34.5%

Table 5
Evaluation results of decompressing stream objects. `PDF Size` and `PDF Size Δ` respectively indicate the average size of PDF files before and after decompressing. `Code Cov` and `Code Cov Δ` respectively show the code coverage at the end of a 24-h test without and with object decompressing. `Increase Rate` is calculated as $\frac{\texttt{PDF Size } \Delta}{\texttt{PDF Size}} - 1$ or $\frac{\texttt{Code Cov } \Delta}{\texttt{Code Cov}} - 1$.

Metrics	Applications / Libraries					
	MuPDF	PDFium	Poppler	Xpdf	PoDoFo	QPDF
# of PDF	1,711	2,386	1,775	1,769	895	1,351
PDF Size (KB)	18.0	17.2	20.9	19.5	16.7	26.2
PDF Size Δ (KB)	21.2	19.9	24.3	22.8	18.8	30.0
Increase Rate	17.6%	15.7%	16.3%	16.9%	12.6%	14.5%
Code Cov (K)	30.4	42.3	22.1	18.4	10.5	27.3
Code Cov Δ (K)	31.7	45.6	24.0	20.9	10.8	28.2
Increase Rate	4.3%	7.8%	8.6%	13.6%	2.9%	3.3%

as seed inputs. We then repeated the tests after we decompressed the stream objects. When decompressing stream objects, we limited the increase of file size to be less than 20%.

We repeated the above evaluation 5 times and report the average results in Table 5. On average, decompressing stream objects increased the code coverage by 6.8%, demonstrating that the strategy is indeed helpful. We also measured the increase of file size. Our strategy increased the file size by 15.6%, below our threshold of 20%.

6. Discussion

6.1. Limitations of generator-reuse

While useful, generator-reuse has several limitations, in comparison to manually-crafted generators. We elaborate on the limitations using Arcee as an example. First, in principle, Arcee inherits the generation capacity of the HTML generators. The structures that Arcee can create are largely restricted by the HTML generators. This brings a bound to Arcee's capability. Second, the H2P converters can also impact the effectiveness of Arcee. On the one hand, the H2P converters could be immaturely engineered. They may skip certain HTML tags or incompletely translate the tags, hurting the semantics created by the HTML generators. On the other hand, as we have described, the H2P converters typically follow templates to translate HTML tags, which are unable to produce semantically-equivalent but structurally-diverse PDF objects. This limits the diversity in the resulted PDF files. Finally, even with perfect HTML generators and perfect H2P converters, we may still not cover certain regions in the input space of PDF software because HTML is less expressive than PDF (i.e., certain PDF files have no HTML equivalents). This problem puts a restriction that we may only explore the overlapped sub-space between HTML and PDF.

6.2. Generality of generator-reuse

In this paper, we only present a single application of the idea of generator-reuse. However, it can be applied in many other challenging fuzzing scenarios. For example, fuzzing C/C++ compilers often requires valid C/C++ programs. Directly generating C/C++ programs is considered challenging. We can alternatively generate assembly code and then run high-quality de-compilers (e.g., Hex-Rays De-compiler and Ghidra De-compiler) to convert the assembly code into C/C++ programs. Since assembly code has no syntax and grammar requirements, we envision its generation should be much easier than C/C++ code.

More broadly speaking, we can even chain the conversions. For example, after we convert assembly code into C/C++ code, we can further convert the C/C++ code into JavaScript code with tools like Emscripten (Voyles, 2015). This way we can further broaden the applications of our approach for fuzzing JavaScript engines. However, chaining too many conversions may introduce more template code, which could essentially reduce the diversity carried by the initial files.

7. Related work

7.1. Mutation-based fuzzing and its applications on PDF

Mutation-based fuzzing is one of the most popular and effective fuzzing techniques. It typically starts with a set of seed inputs, or *seeds*, and mutates each seed to generate new test cases based on feedback from execution of the target program (Sutton et al., 2007). Many research works have been done to improve mutation-based fuzzing from different perspectives.

First, they explore new kinds of *feedback* to facilitate seed scheduling and mutation. AFL (Zalewski, 2014) considers code branches covered in a round of execution as feedback, which is

refined by Steelix (Li et al., 2017), CollAFL (Gan et al., 2018), and PTrix (Chen et al., 2019) with more fine-grained, control-flow related information. Going beyond, TaintScope (Wang et al., 2010), Vuzzer (Rawat et al., 2017), GREYONE (Gan et al., 2020), REDQUEEN (Aschermann et al., 2019b), and Angora (Chen and Chen, 2018) use taint analysis to identify data flows that can affect code coverage.

Second, they propose new schemes of seed *scheduling*. AFLFast (Böhme et al., 2016), DigFuzz (Zhao et al., 2019), and MOPT (Lyu et al., 2019) consider code-coverage feedback from execution of the target software as guidance to schedule seeds for mutation. These solutions prefer seeds with higher potentials of leading to new code coverage. In contrast, SAVIOR (Chen et al., 2020) runs static analysis to estimate the distribution of potential vulnerabilities. It then prioritizes seeds that can lead to more potential vulnerabilities.

Third, they develop new *mutation* methods to remove common barriers that prevent fuzzers from reaching more code. Majumdar and Sen (2007) introduce the idea of hybrid fuzzing, which leverages concolic execution to solve complex path conditions that are usually difficult for pure fuzzing to satisfy. This idea was followed by many works to improve the combination of fuzzing and concolic execution (Pak, 2012; Stephens et al., 2016), the scheduling of concolic execution (Zhao et al., 2019), and the efficiency of concolic execution (Yun et al., 2018). TFuzz transforms tested programs to bypass complex conditions and forces the execution to reach new code. It later uses a validator to reproduce inputs that satisfy the complex conditions in the original program. Angora (Chen and Chen, 2018) assumes a black-box function at each complex condition and applies gradient descent to find satisfying inputs. This method is later improved by NEUZZ (She et al., 2019) with a smooth surrogate function to approximate behaviors of the tested program.

People have applied mutation-based fuzzing to PDF software (Alon and Ben-Simon, 2018; Feldmann, 2019). Alon and Ben-Simon (Alon and Ben-Simon, 2018) manually generated fuzzing harnesses (i.e., valid sequences of API calls) from the JP2KLib.dll library in Adobe Reader. They then ran WinAFL (Allievi and Johnson, 2017) on the fuzzing harnesses. Feldmann (Feldmann, 2019) collected 80,000 PDF files and minimized them to 220 unique PDF files, using an approach similar to AFL-cmin. Taking the 220 PDF files as seeds, Feldmann ran dumb mutation-based fuzzing and discovered bugs from FoxitReader, PDFXChangeViewer, and XNView. Despite the two approaches found bugs from PDF software, their finding of bugs was not entirely attributable to mutation-based fuzzing. The first approach relies on creation of fuzzing harnesses and the second approach depends on the large corpus of seed PDF files. Arcee follows a principle similar to the two approaches: it combines more effective approaches with mutation-based fuzzing to find PDF bugs. However, Arcee has the advantage of avoiding the manual efforts to create fuzzing harnesses or collect PDF files. More importantly, Arcee optimizes the generated PDF files that is useful for mutation-based fuzzing of PDF.

7.2. Generation-based fuzzing and its applications on PDF

Generation-based fuzzing is another major type of fuzzing. Instead of deriving new test cases by mutating seed inputs, it directly generates inputs to test the target software (Sutton et al., 2007). In comparison to mutation-based fuzzing, generation-based fuzzing has the advantage of being able to produce highly structured inputs (i.e., inputs with satisfied grammars). To generate inputs, generation-based fuzzing tools mostly use two approaches.

The first approach is to pre-define grammar rules and then follow the grammar rules to generate structurally/syntactically correct test cases. Specifically, Domato (Fratric, 2017) and Freedom (Xu et al., 2020) follow manually created DOM rules to generate valid HTML files. In contrast, DOMFuzz (Mozilla Fuzzing Security, 2019) moves sub-components across the DOM trees of different HTML files using appending and insertion operations, to generate more HTML files. In principle, this approach is similar to our structure-aware object exchange. Park et al. (2020) and jsfunfuzz (Security, 2019) generate random, but syntactically correct JavaScript code based on manually-created grammar models and templates. LangFuzz (Holler et al., 2012) randomly combines the fragments of JavaScript code to generate new JavaScript code, under the restrictions of correct syntax. There are also many network-protocol fuzzers, such as Frankencerts (Brubaker et al., 2014), TLS-Attacker (Somorovsky, 2016), and LL-Fuzzer (Spensky and Hu, 2015). These fuzzers follow a model of the target network protocol to generate structurally correct test cases.

The second approach is to learn a grammar or a model using existing test cases and then follow the grammar or the model to generate new test cases. Skyfire (Wang et al., 2017) builds a probabilistic, context-sensitive grammar model for HTML and XSL via learning from valid inputs. It then uses the grammar to generate inputs that are accepted by target software. Similarly, DeepFuzz (Liu et al., 2019) trains a Sequence-to-Sequence model to automatically and continuously generate well-formed C programs. In contrast, Godefroid et al. (2008) train a grammar-based input generator to retrofit white-box fuzzing.

Past efforts have applied the learning-based approach to generate PDF files. Specifically, Godefroid et al. (2017) proposed a neural-network-based method to learn the structures of valid PDF files and generate test cases for fuzzing PDF software. As we described in §2.2, learning PDF is highly complex and there exist complex references between objects inside PDF. Models built with the approach presented in Godefroid et al. (2017) only generate less-complicated individual objects, which is less efficient than producing whole PDF files. Arcee reuses the generation rules of H2P generators that could generate valid and complex PDF files. Indeed, our evaluation demonstrates that Arcee, following the idea of generator-reuse, can generate more diverse target files.

8. Conclusion

We present the idea of generator reuse. It reuses existing generators (e.g., HTML generators) and grammar converters (e.g., HTML-to-PDF generators) to build new generators (e.g., PDF generators), bypassing the manual efforts required to summarise the grammar rules for the new generators. We showcase the feasibility of this idea by applying it to build a PDF generator. Through extensive evaluation, we demonstrate the utility of the PDF generator. Moreover, in the process of building and evaluating the PDF generators, we identified a set of limitations of our generator reuse idea and bring up a group of strategies to mitigate those limitations.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared link to our code in the manuscript.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. This project was supported in part by grants from the

Chinese National Natural Science Foundation (61272078, 62032010, 62172201) and Cisco. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

References

Adobe, 2020. How to convert websites to pdfs in 3 easy steps. https://acrobat.adobe.com/us/en/acrobat/how-to/convert-html-to-pdf.html.

AFL, 2015. Afl afl-tmin.c at master. https://github.com/google/AFL/blob/master/afl-tmin.c.

Allievi, A., Johnson, R., 2017. Harnessing intel processor trace on windows for fuzzing and dynamic analysis. https://recon.cx/2017/brussels/talks/intel_processor_trace.html.

Alon, Y., Ben-Simon, N., 2018. 50 cves in 50 days: fuzzing adobe reader. https://rb.gy/1w3veo.

Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., Teuchert, D., 2019. Nautilus: fishing for deep bugs with grammars. In: Proceedings of 2019 Network and Distributed System Security Symposium(NDSS).

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

Beautiful, 2004. soup: We called him tortoise because he taught US. https://www.crummy.com/software/BeautifulSoup/.

Berners-Lee, T., Connolly, D., 1995. Hypertext markup language-2.0.

Bidelman, E., 2019. Automated testing with headless chrome. https://developers.google.com/web/updates/2017/06/headless-karma-mocha-chai.

Bienz, T., Cohn, R., C. Adobe Systems (Mountain View, 1993. Portable Document Format Reference Manual. Citeseer.

Böhme, M., Pham, V.-T., Roychoudhury, A., 2016. Coverage-based greybox fuzzing as Markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, pp. 1032–1043. doi:10.1145/2976749.2978428.

Brubaker, C., Jana, S., Ray, B., Khurshid, S., Shmatikov, V., 2014. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 114–129. doi:10.1109/SP.2014.15.

Chen, P., Chen, H., 2018. Angora: efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 711–725. doi:10.1109/SP.2018.00046.

Chen, Y., Li, P., Xu, J., Guo, S., Zhou, R., Zhang, Y., Wei, T., Lu, L., 2020. Savior: towards bug-driven hybrid testing. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 1580–1596. doi:10.1109/SP40000.2020.00002.

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

Chromium, 2020. The chromium projects. https://www.chromium.org/.

Cog, G.,, 2020. Xpdfreader. https://dl.xpdfreader.com/xpdf-tools-linux-4.02.tar.gz.

Dor1s, 2020. Testcase of pdf. https://github.com/google/AFL/blob/master/testcases/others/pdf/small.pdf.

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

Fratric, I., 2017. domato: Dom fuzzer. https://github.com/googleprojectzero/domato.

freedesktop, 2020. Poppler. https://gitlab.freedesktop.org/poppler/poppler/-/commit/e1f562582db4ed902989e23f7a02645a51da8e88.

Gan, S., Zhang, C., Chen, P., Zhao, B., Qin, X., Wu, D., Chen, Z., 2020. GREYONE: data flow sensitive fuzzing. In: Proceedings of the 29th USENIX Security Symposium. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity20/presentation/gan.

Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., Chen, Z., 2018. Collafl: path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 679–696. doi:10.1109/SP.2018.00040.

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

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

Google, 2020. Chrome fuzzer program. https://www.google.com/about/appsecurity/chrome-rewards/#fuzzerprogram.

Google, 2020. Pdfium. https://pdfium.googlesource.com/pdfium/+/944a7850.

Harris, R., 2017. Html to pdf gets even easier with phantompdf 8.2. https://appdevelopermagazine.com/html-to-pdf-gets-even-easier-with-phantompdf-8.2/.

Holler, C., Herzig, K., Zeller, A., 2012. Fuzzing with code fragments. In: 21st USENIX Security Symposium (USENIX Security 12). USENIX Association, Bellevue, WA, pp. 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

Incorporated, A. S., 2006. Pdf reference (third edition). https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdf_reference_archive/pdf_reference_1-7.pdf.

Jay Berkenbilt, 2020. Qpdf. https://github.com/qpdf/qpdf/commit/78b9d6bfd4cb.

Jay Berkenbilt and Thorsten Schning, 2020. Wrapper of qpdf-fuzzer. https://github.com/qpdf/qpdf/blob/78b9d6bfd4cbd3e947b1c5ffe73eb97b040e312a/fuzz/qpdf_fuzzer.cc.

KDE, 2009. Eekonq: a web browser for KDE based on webkit.https://github.com/KDE/rekonq.

KDE, 2016. Konqueror. https://apps.kde.org/en/konqueror.

Kulkarni, A., Truelsen, J., 2020. wkhtmltopdf. https://wkhtmltopdf.org/.

LG, 2009. webos. https://en.wikipedia.org/wiki/WebOS.

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

Liu, X., Li, X., Prajapati, R., Wu, D., 2019. Deepfuzz: automatic generation of syntax valid C programs for fuzz testing. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, pp. 1044–1051.

Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.-H., Song, Y., Beyah, R., 2019. MOPT: optimized mutation scheduling for fuzzers. In: 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, CA, pp. 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu.

Majumdar, R., Sen, K., 2007. Hybrid concolic testing. In: Software Engineering, 2007. ICSE 2007. 29th International Conference on. IEEE, pp. 416–426.

Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M., 2019. The art, science, and engineering of fuzzing: A survey. IEEE Trans. Softw. Eng., vol. 47. pp. 2312-2331.

Misherghi, G., Su, Z., 2006. HDD: hierarchical delta debugging. In: Proceedings of the 28th International Conference on Software Engineering (ICSE), pp. 142–151.

Mozilla Fuzzing Security, 2019. Dom fuzzers. https://github.com/MozillaSecurity/domfuzz.

Pak, B.S., 2012. Hybrid Fuzz Testing: Discovering Software Bugs Via Fuzzing and Symbolic Execution. School of Computer Science Carnegie Mellon University.

Pandoc, 2020. Pandoc - a universal document converter. https://pandoc.org/.

Park, S., Xu, W., Yun, I., Jang, D., Kim, T., 2020. Fuzzing javascript engines with aspect-preserving mutation. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 1629–1642. doi:10.1109/SP40000.2020.00067.

Parr, T., 2016. Antlr - another tool for language recognition.

Preface By-Warnock, J., Preface By-Geschke, C., 2001. Pdf reference: adobe portable document format version 1.4 with cdrom.

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

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

Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D., 2012. Addresssanitizer: a fast address sanity checker. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12). USENIX Association, Boston, MA, pp. 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

She, D., Pei, K., Epstein, D., Yang, J., Ray, B., Jana, S., 2019. Neuzz: efficient fuzzing with neural program smoothing. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 803–817. doi:10.1109/SP.2019.00052.

Software, A., 2020. Mupdf. https://github.com/ArtifexSoftware/mupdf/commit/0e90241161.

Somorovsky, J., 2016. Systematic fuzzing and testing of TLS libraries. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1492–1504.

Sourceforge, 2020. Podofo. https://sourceforge.net/p/podofo/code/2016/.

Spensky, C., Hu, H., 2015. An automated NFC fuzzing framework for android devices. https://github.com/mit-ll/LL-Fuzzer.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2016. Driller: Augmenting fuzzing through selective symbolic execution. In: Proceedings of 2016 Network and Distributed System Security Symposium(NDSS), vol. 16, pp. 1–16.

Sutton, M., Greene, A., Amini, P., 2007. Fuzzing: Brute Force Vulnerability Discovery. Pearson Education.

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

Voyles, D., 2015. Getting started with emscripten: transpiling c/c++ to javascript/html5. https://www.sitepoint.com/getting-started-emscripten-transpiling-c-c-javascript-html5/.

Wang, J., Chen, B., Wei, L., Liu, Y., 2017. Skyfire: data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 579–594. doi:10.1109/SP.2017.23.

Wang, T., Wei, T., Gu, G., Zou, W., 2010. Taintscope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 497–512. doi:10.1109/SP.2010.37.

Wikipedia contributors, 2021. Pdf — Wikipedia, the free encyclopedia. [Online; accessed 20-January-2021]. https://en.wikipedia.org/w/index.php?title=PDF&oldid=999188580.

Xu, W., Park, S., Kim, T., 2020. Freedom: engineering a state-of-the-art dom fuzzer. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 971–986.

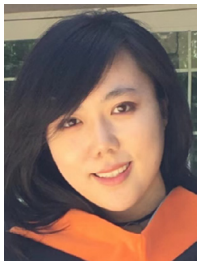
Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T., 2018. QSYM : a practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, MD, pp. 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

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

Zhao, L., Duan, Y., Yin, H., Xuan, J., 2019. Send hardest problems my way: probabilistic path prioritization for hybrid fuzzing. In: Proceedings of 2019 Network and Distributed System Security Symposium(NDSS).

Chengbin Pang received the B.E. degree in Computer Science and Technology from Shandong University in 2017. He is currently pursuing the Ph.D degree in the Department of Computer Science and Technology, Nanjing University. His current research interests include system security, software security and binary analysis.

Hongbin Liu received the B.E. degree in Information Science from the University of Science and Technology of China in 2020. He is currently a Ph.D. student in the Department of Electrical and Computer Engineering, Duke University. His current research interests include machine learning, security, and privacy.



Yifan Wang is a Ph.D. student in the Department of Computer Science at Stevens Institute of Technology. She received her M.S. in Computer Science from New York Institute of Technology in 2017. Her recent research focus is on software security. Specifically, she leverages fuzzing to find vulnerabilities in software.

Neil Zhenqiang Gong received the B.E. degree in computer science from the University of Science and Technology of China, in 2010, and the Ph.D. degree in computer science from the University of California Berkeley, in 2015. He is an assistant professor in the Department of Electrical and Computer Engineering and Department of Computer Science, Duke University. His research interests include computer security and privacy with a recent focus on security and privacy analytics. Specifically, he leverages data science techniques, including machine learning, network science, natural language processing, and optimization, to study security and privacy in various systems such as social systems, web services, mobile systems, and Internet of Things; and he also builds secure and privacy-preserving data analytics systems.

Bing Mao received the M.S. and Ph.D. degrees in computer science from Nanjing University, in 1994 and 1996, respectively. He is a professor with the Department of Computer Science and Technology, Nanjing University. His current research interests include system security, software security and binary analysis.

Jun Xu is an Assistant Professor in the School of Computing at University of Utah. He received his Ph.D. from the College of Information Sciences and Technology at the Pennsylvania State University. His research area covers software security, system security, malware and binary analysis.