COOPER: Testing the Binding Code of Scripting Languages with Cooperative Mutation

Peng Xu[†‡] Yanhao Wang[§] Hong Hu[¶] Purui Su[†‖✉]

[†]*TCA/SKLCS, Institute of Software, Chinese Academy of Sciences*
[‡]*University of Chinese Academy of Sciences*
[§]*QI-ANXIN Technology Research Institute*
[¶]*Pennsylvania State University*
[‖]*School of Cyber Security, University of Chinese Academy of Sciences*

***Abstract*—Scripting languages like JavaScript are being integrated into commercial software to support easy file modification. For example, Adobe Acrobat accepts JavaScript to dynamically manipulate PDF files. To bridge the gap between the high-level scripts and the low-level languages (like C/C++) used to implement the software, a binding layer is necessary to transfer data and transform representations. However, due to the complexity of two sides, the binding code is prone to inconsistent semantics and security holes, which lead to severe vulnerabilities. Existing efforts for testing binding code merely focus on the script side, and thus miss bugs that require special program native inputs.**

In this paper, we propose *cooperative mutation*, which modifies both the script code and the program native input to trigger bugs in binding code. Our insight is that many bugs are due to the interplay between the program initial state and the dynamic operations, which can only be triggered through two-dimensional mutations. We develop three novel techniques to enable practical cooperative mutation on popular scripting languages: we first cluster objects into semantics similar classes to reduce the mutation space of native inputs; then, we statistically infer the relationship between script code and object classes based on a large number of executions; at last, we use the inferred relationship to select proper objects and related script code for targeted mutation. We applied our tool, COOPER, on three popular systems that integrate scripting languages, including Adobe Acrobat, Foxit Reader and Microsoft Word. COOPER successfully found 134 previously unknown bugs. We have reported all of them to the developers. At the time of paper publishing, 59 bugs have been fixed and 33 of them are assigned CVE numbers. We are awarded totally 22K dollars bounty for 17 out of all reported bugs.

I. INTRODUCTION

Scripting languages, like JavaScript and Python, have been adopted in commercial software to provide convenient interfaces to the underly complicated systems. For example, Adobe Acrobat accepts JavaScript code to manipulate internal PDF objects [1], while IDA Pro provides a Python binding for third-party developers to inspect binary attributes [23]. As commercial software is usually written in low-level languages like C/C++, a *binding* layer is necessary for transferring the arguments and transforming the representation for the high-level scripting languages. However, since the software and the script are developed independently, the binding layer is prone to produce inconsistent representations or miss security checks, which lead to tons of severe security vulnerabilities [6, 11].

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24353
www.ndss-symposium.org

To mitigate the threat from the binding code, researchers have proposed multiple program analysis techniques to detect various vulnerabilities, like unhandled crashes [27, 30, 53], type-safety violations [15, 16, 53] and memory-safety violations [6, 31]. Most techniques statically analyze the binding code to detect specific violations. For example, Brown *et al*. develop a set of static checkers, each of which aims to locate a particular type of bugs, such as implicitly casted variables [6]. These methods are limited by the nature of static methods: unscalable to handle large code base; the inaccuracy leading to many false alarms; lack of concrete inputs to trigger the bug. Instead, Lee *et al*. build a dynamic analysis tool to detect specification violations at runtime [28]. However, it merely reports bugs triggered by the given inputs and thus has limited bug coverage.

Recently, *fuzzing* is widely used to test many programs and systems [20, 32, 61] and successfully found thousands of bugs [49]. The basic idea of fuzzing is to randomly generate a large number of inputs and run the program with these inputs to expose visible anomaly behaviors, like crashes or security violations [12, 33, 47, 48]. Because of the efficacy on finding bugs and the scalability to handle complicated systems, fuzzing is recently used to test the runtime and the binding code [11, 14, 21, 40, 45, 60] of JavaScript – the most popular scripting language [62]. Among them, Dinh *et al*. develop a tool, `favocado`, to generate semantically valid JavaScript code to solely test the JavaScript binding code and found lots of exploitable bugs in PDF readers and web browsers [11].

Although previous works demonstrate the feasibility of fuzzing binding code, we notice their limitation on exploring the program states due to *one-dimensional* mutation [59]. Specifically, `favocado` merely modifies JavaScript statements in order to trigger bugs in the binding layer. However, binding code is designed to connect the high-level scripting languages and the underlying systems written in low-level languages. Therefore, it accepts inputs from two dimensions: the code written in the scripting language (*i.e.*, the script) *and* the native input for the underlying system (*e.g.*, PDF files for Adobe Acrobat). Both dimensions can affect the execution of the binding code, and one cannot completely replace another. Mutating one dimension of the inputs cannot discover all bugs in the binding code. For example, although Adobe Acrobat accepts both PDF files and JavaScript code as its inputs, particular program features such as font are only mutable from the native input, i.e., the PDF file – merely mutating JavaScript code cannot trigger such bugs [1].

The solution to address this limitation seems straightforward: we mutate the inputs from two dimensions: modifying native inputs to prepare the program's initial states, and synthesizing valid scripts to trigger more diverse states. However, the simple method that randomly mutates two dimensions cannot explore the binding code effectively due to the large mutation space: the native input contains many objects but not all of them are related to the binding code, while the script code only accepts particular program states for further operations.

To reduce the mutation space from two dimensions, we propose *cooperative mutation*, which uses the relationship between native inputs and script code to guide the fuzzing process. Specifically, it first modifies native objects that are related to the binding code; then, it synthesizes script code that will manipulate the program state triggered by the current native input. In this way, all mutation energy is used towards affecting the execution of the binding code. However, we need to address two challenges before adopting cooperative mutation. First, it is nontrivial to infer the relationship between the native input and the binding code. Considering the complicated underlying system and diverse scripting languages, manual effort is unscalable for handling real-world binding code. Second, it is hard to tell whether each script code accepts the program state triggered by a given native input. This problem is even worse in commercial software, where the source code is usually unavailable and the document is not given in detail.

We design three novel techniques, object clustering, statistical relationship inference and relationship-guided mutation, to address the aforementioned challenges to enable practical cooperative mutation. First, we cluster objects in native inputs based on their semantics similarity. The clustering helps reduce the input mutation space significantly. Then, we rely on the large number of concrete executions to statistically infer the relationship between the native input and the script code. Specifically, we record the objects and script code in each file, and open it using the application to collect the execution result of the script code (*i.e.*, success or failure). If the success of the execution highly correlates with the inclusion of particular objects, we believe such objects are internally related to the script statement. Although such relationship is not formally confirmed, the large number of executions provide a statistical guarantee. Our evaluation in §V shows that the inferred relationship can help find a large number of severe bugs in the binding code. At last, we define a set of mutation policies that use the inferred relationship to guide the object modification and script generation. Specifically, we only modify objects related to script code, and update the corresponding script code to generate various runtime operations.

We implement COOPER, a prototype that uses cooperative mutation to test the binding code of scripting languages. COOPER contains 1,581 lines of Python code for object clustering and relationship inference, and 2,756 lines of Python code for relationship-guided mutation. Currently, it supports testing two binding types, JavaScript with PDF files and Visual Basic with Microsoft Word documents. Since the idea of cooperative mutation is general, it is straightforward to extend COOPER to other scripting languages, like Python and Perl.

To understand the effectiveness of COOPER, we applied it on three popular binding systems, Adobe Acrobat, Foxit Reader and Microsoft Word. COOPER successfully detected 134 unknown bugs, including 60 for Adobe Acrobat, 56 for Foxit Reader, and 18 for Microsoft Word. We have reported all these bugs to their developers. At the time of paper publishing, 103 bugs have been confirmed and 59 bugs of them have been fixed. Due to the severity of the reported bugs, we have received 33 CVE numbers and $22K bug bounties from various sources. Besides the bug detection, we also conducted unit tests to understand the contribution of each component of COOPER. The evaluation result shows that our relationship inference can deduce strong relationships between objects and script code, while the relationship-guided mutation improves the fuzzing efficacy by finding significantly more unique bugs in one week.

In summary, we make the following contributions.

- We propose cooperative mutation, which simultaneously modifies the native input and the *related* script code to test the binding code of scripting languages.
- We design COOPER, a prototype that infers relationships between the native input and the script code, and utilizes the relationship to guide the two-dimensional mutation.
- We applied COOPER on real-world popular commercial software and detected 134 bugs, which results in 33 CVEs and $22K bug bounties.

We plan to release the source code of COOPER at https://github.com/TCA-ISCAS/Cooper to help improve the security of binding layers of scripting languages.

II. BACKGROUND AND MOTIVATION

To explain the research problem, we first briefly introduce scripting languages and the binding code. Then, we use one example to demonstrate common vulnerabilities in the binding code. Next, we demonstrate why existing work fails to trigger or detect such bugs and how the cooperative mutation addresses the limitation. At last, we analyze the challenges of adopting cooperative mutation and provide an overview of our solution.

A. *Scripting Languages and Binding Code*

Scripting languages are designed to automate the execution of tasks. Each underlying complicated task is encapsulated as a high-level API call, while developers usually combine necessary API calls to achieve diverse functionalities. Scripting languages are intended to be easy to learn and use, and most of them get interpreted at runtime instead of statically compiled. While several scripting languages are designed for specific domains, like Bash for Unix operating system, many are general-purpose programming languages, like Python and Perl. With the high demand of quick development, scripting languages are getting more and more popular. For example, JavaScript, the scripting language designed for web browsers, have been the most popular programming language for many years [62].

```
1  %PDF-1.3
2  1 0 obj << /Pages 2 0 R >> endobj
3  2 0 obj << /Kids [ 3 0 R ] >> endobj
4  3 0 obj << /Resources << /Font << /TT1 4 0 R >> >>
5             /AA << /O << /S /JavaScript
6                         /JS 5 0 R >> >> >> endobj
7  4 0 obj << /FirstChar 0
8             /Widths [ 778 778 ... 556 500 ]    % 256 + 1 elements
9             /LastChar 255 >>  endobj
10 5 0 obj << /Length 539 >>
11         stream
12           this.zoomType=zoomtype.refW;        % Trigger the bug
13         endstream
14     endobj
15 trailer << /Root 1 0 R >>
```

Fig. 1: Simplified PDF file that triggers a heap buffer overflow of Adobe Acrobat. To trigger this bug, the PDF file must take two steps: overflowing the memory (line 8) and run JavaScript (line 12).

Many commercial software integrate the runtime of scripting languages so as to support unified, convenient and cross-platform programming interfaces. For example, PDF processing applications such as Adobe Acrobat and Foxit Reader allow the embedded JavaScript code in PDF documents to dynamically manipulate PDF objects or trigger dynamic actions [1, 13]. Binary decompilers like IDA Pro and Ghidra also provide Python bindings so that third-party developers can easily access the binary attributes and construct various extensions [18, 23].

Since scripting languages are designed in high-level and encapsulate the details, the commercial software that written in low-level languages (*e.g.*, C/C++) cannot directly communicate with the script code due to different memory models and type systems. A binding layer is necessary to bridge gaps by transforming the data from one representation to another. However, since the underlying commercial system and the high-level scripting languages are developed independently, even at different ages, it is challenging for the binding code to correctly handle all transformations. Missing security checks, unhandled exceptions and inconsistent semantics have led to a large number of severe security vulnerabilities [6, 11, 27, 30, 53].

B. Motivating Example

Figure 1 shows a simplified PDF file that triggers a heap-based buffer overflow of Adobe Acrobat. We found this bug using our tool and reported it to Adobe. At the time of paper writing, the bug has officially been fixed. PDF files are organized as trees of objects. Each object has a unique identifier and its content is defined within a pair of keywords `obj` and `endobj`. In the example, the PDF tree starts from the root object 1 defined at line 2. Object 1 contains one `Pages` object 2, which in turn has only one `Kids` object 3. Object 3 uses the font `TT1` that is defined in object 4. Object 4 includes an array `Widths` that specifies the width of each glyph in the character set (line 8). Object 3 further specifies the additional action `/AA` when opening `/O` this file, which runs JavaScript code defined in object 5 (line 10-14). The JavaScript statement at line 12 changes the zooming type of the file to `ReflowWidth`.

This bug can be triggered by an interplay between PDF objects and JavaScript statements, shown in Figure 2. Specifically, when Adobe Acrobat opens this file, it will follow the instruction in object 3 to execute the JavaScript statement at line 12. The JavaScript engine invokes the functions defined in the `reflow.api` module, which is implemented in C++. The `reflow.api` module first creates a fixed-length heap buffer

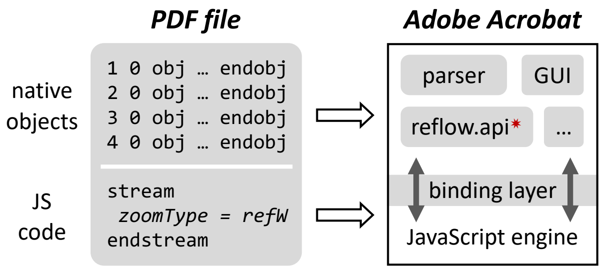

Fig. 2: Two-dimensional inputs to Adobe Acrobat. A PDF may define both native PDF objects and JavaScript code, where the former is processed by the native Acrobat modules (*e.g.*, reflow.api) while the latter is handled by the embedded JavaScript engine. JavaScript code invokes native APIs through the binding layer.

that occupies 256 four-bytes. Then, it copies the content of the `Widths` array in the object 4 to the fixed-length buffer. However, the array in this file contains 257 four-byte elements, and thus the copy will exceed the boundary of the fixed-length buffer and overwrite the following four-byte. Our manual analysis reveals that `reflow.api` checks the length of `Widths` before the memory copy, but it carelessly allows the marginal size 257 due to the confusion between $>$ and $\geq$. If we manually replace the `ja` instruction (*i.e.*, jump if above) with `jae` (*i.e.*, jump if above or equal), the program will work well without any crash. Our further analysis reveals that attackers can use any value between `0` and `0x7fffffff` to overwrite the following four-byte. Therefore, they can prepare special heap layouts [10, 52, 56] to either overwrite function pointers to execute arbitrary code [4, 46, 50], or modify critical data for similar malicious purposes [7, 25].

This bug received a CVE number CVE-2021-28638. We are awarded *$2,500* bounty for reporting this particular bug.

C. Necessity of Cooperative Mutation

To generate the bug-triggering PDF file in Figure 1 from normal inputs, we need to modify ***both*** PDF objects and JavaScript statements. On the one hand, we need to actively modify the `Widths` array to add extra elements. Based on the PDF format reference [2], in a simple font (*e.g.*, the `TT1` object in this example), each character code is represented by one byte. Therefore, the `Widths` array which specifies the glyph width of each supported character can have at most 256 elements. As a demonstration, among 16,000 PDF files collected from public, we did not find any one containing `Widths` objects with 257 or more elements. More importantly, Adobe Acrobat does not provide an interface for JavaScript code to modify the glyph width [1]. This means that merely updating JavaScript code can never trigger this bug. On the other hand, the action of changing the zoom type to `reflowW` will redraw the PDF file as single column whose width is the same as the current window (mainly for easy reading). This action cannot be specified in any PDF native objects, and we have to resort to the JavaScript code to dynamically trigger this action at runtime. Manually enabling the reflow mode is also possible by clicking the corresponding button on the Acrobat graphics interface, but that will disable the efficient automatic program testing.

Previous work cannot effectively detect this bug. Static analysis techniques either ignore this type of violations [15, 16, 27, 30, 31, 53] or cannot handle the code of Adobe Acrobat [6].

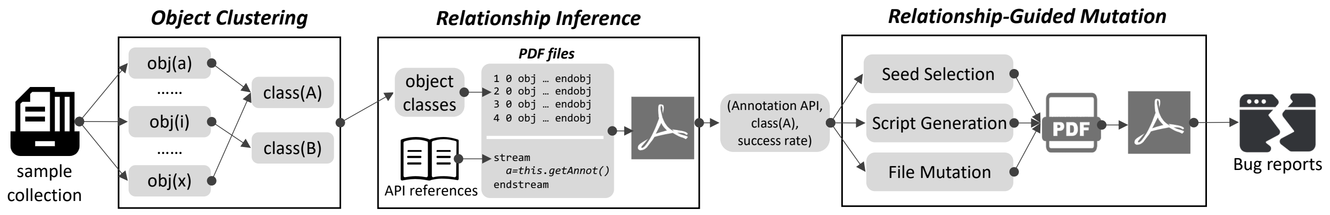

Fig. 3: Overview of COOPER. It takes the program binary, script manuals and sample documents as inputs, and reports memory-safety bugs. COOPER first clusters objects based on their high-level semantics. It then infers the relationship between objects and script APIs. Such relationships help COOPER cooperatively mutate objects and script code. The generated document is sent to the target program to trigger bugs.

It is worthwhile to note that the bug does not happen inside the binding layer, but in the native `reflow.api` module. Statically analyzing such a large code base without source code is known to be challenging [6], if not impossible. Dynamic program analysis [28] can neither detect this bug without a malformed PDF file that is generated from two-dimensional mutation.

The recent fuzzing work `favocado` tries to generate semantic-valid scripts to test the binding code of JavaScript in PDF readers and browsers [11]. However, `favocado` only modifies the JavaScript statements and does not make any changes to the PDF native objects. Considering that most of the normal PDF files do not have more than 256 elements in the `Widths` array, it is challenging for `favocado` to identify this bug using its current method. As demonstrated in [11], other JavaScript testing tools could not be used to test binding code in non-browser environments [22, 24, 29, 40, 54].

Our solution, COOPER, utilizes cooperative mutation to synthesize PDF files and successfully triggered this bug for the first time. Different from static analysis techniques, COOPER runs Adobe Acrobat with concrete PDF files and only watches for externally visible abnormal behaviors, like execution crashes. Therefore, it can test large programs regardless of the code size or the bug location and captures a wide range of violations. It actively and continuously generates diverse inputs to stress Adobe Acrobat, especially the binding layer, to run different code paths. Most importantly, it modifies both the JavaScript code and the PDF objects to trigger bugs as many as possible.

D. Challenges of Cooperative Mutation

Although the idea of cooperative mutation is easy to understand, it is nontrivial to implement the solution. The main challenge comes from the large mutation space of both dimensions. First, the native input file can have many objects, but not all objects are related to the binding layer [44]. In the case of Figure 1, the key step is to insert an extra element into the `Widths` array, whereas the concrete values of the existing 256 elements does not affect the execution of the binding layer. Second, the scripting code can combine any number of functions from thousands of APIs provided by the underlying system [11]. The synthesized script should accept the program state created by the underlying system by parsing the native input. Otherwise, the script either cannot trigger new program states, or even fails to execute. Previous work like `favocado` proposed API grouping to reduce the code search space. Specifically, they classify various APIs into several independent groups based on the high-level semantics. For each time of mutation, `favocado` only synthesizes code using APIs from one group. However, API grouping only partially addresses our challenges as the input space is still too large to be handled easily.

To reduce the mutation space, we propose to use the relationship between the input and the script code to guide the mutation. Specifically, we modify the input and the related script to divert the execution path of the binding layer. The guidance of input-script relation helps us focus on testing binding code. However, we need to address the following challenges.

C1. How to effectively infer relationships between the native input and the script code?
C2. How to use the inferred relationship to guide the input mutation from two dimensions?

To address challenge **C1**, we need a general solution to automatically infer the relationship for any given scripting languages and underlying systems. It is not scalable to use tedious human effort to understand the language specification, which usually contains hundreds of pages [1]. For challenge **C2**, the solution should be general while flexible so that users can adjust the policy when necessary. Other than these two challenges, we realize that the tremendous input space could prevent any effective solutions in the first place. We plan to follow the methodology used in `favocado` and classify input objects into different classes, which leads to the third challenge.

C3. How to cluster input objects to semantic-similar classes?

In the following sections, we will present our solutions to these challenges and assemble them into a system, COOPER, that can effectively test binding code of scripting languages.

III. COOPER DESGIN

COOPER leverage the relationship between the native objects and the script code to guide the mutation. Figure 3 provides an overview of our system, which takes API references, program binaries and sample documents as inputs and produces various bug reports of the tested program. In the beginning, COOPER parses the given sample documents to extract native objects. To reduce the object search space (*i.e.*, challenge **C3**), COOPER categorizes objects into different classes based on their attributes (§III-A). COOPER also adopts method proposed in previous work to group APIs [11]. Then, COOPER infers the relationship between object classes and API groups to tackle challenge **C1**. Specifically, it produces a large number of documents by combining different object classes and API groups, and records the execution results of the embedded scripts. Based on the success rate of the script execution and the distribution of object classes, COOPER infers the relationship between API groups and object classes (§III-B). At the end, COOPER leverages the inferred relationship to guide the object selection, script generation and object mutation (§III-C). We design several cooperative mutation strategies to address

```html
<img src="navbar.gif" border=0 usemap="#map1">
<map name="map1">
  <area href=search.html alt="Search" shape=rect coords="184,0,276,28">
  <area href=shortcut.html alt="Go" shape=rect coords="118,0,184,28">
</map>
```

(a) HTML file for graphical navigational toolbar

```xml
<w:rPr>
   <w:rFonts w:ascii="Preeti" w:hAnsi="Preeti" />
   <w:sz w:val="32" />
   <w:szCs w:val="32" />
</w:rPr>
```

(b) XML file for a DOCX document

Fig. 4: HTML and XML files commonly use attributes to define native objects.

challenge **C2**. The generated document is sent to the target program for execution, and any crash indicates a potential bug.

A. *Native Object Clustering*

Suppose one document contains M native objects and the binding layer supports N script APIs to manipulate objects, in theory we need to infer relationships of $M * N$ pairs of object-API combinations. Considering the large value of $M * N$, it is impractical to conduct the inference for each pair. Instead, relationships are actually defined between object classes and API groups. In other words, objects of the same type can be manipulated by the same set of APIs. Therefore, in the first step we cluster the large number of objects into type-based classes in order to simply the relationship inference task.

Object Attributes. It is common that an object in the document is described as a set of attributes, as shown in Equation 1. Each attribute A has a name and a value. An attribute name is usually a readable string that has high-level semantics, like `Length` for variable size, while an attribute value is either a constant, or another object (we call it *value object*) defined by more attributes. While Equation 1 shows an abstract representation, we can find different implementations of objects and attributes from popular document formats: in the PDF file of Figure 1, object 1 has one attribute `/Pages` whose value is object 2, and object 4 has three attributes where all of them are constant values; in the HTML file of Figure 4a, we can find the `img` object has three attributes, where the value of the `usemap` attribute is another `map` object; Figure 4b shows part of an XML file included in a DOCX document, where we can find the object `w:rPr` is defined with three attributes `w:rFonts`, `w:sz` and `w:szCs` and each attribute value is also an object defined by other attributes. After analyzing 16,000 randomly collected PDF files, we find that more than half of the PDF objects are `Dictionary`s, which are defined as lists of attributes. Therefore, object attributes are widely used in popular documents, and we can use it to reliably cluster objects.

$$O : object = \begin{Bmatrix} A_0 : name_0 = object_0, \\ A_1 : name_1 = object_1, \\ A_2 : name_2 = object_2, \\ ... = ... \end{Bmatrix} \tag{1}$$

Since attribute names deliver high-level semantics, we decide to use them to cluster objects. On the one hand, objects of the same type should share common attribute names by design. This is the theoretical foundation of our attribute-based clustering. On the other hand, due to optional attributes (i.e., objects may or may not have the attribute), even if two objects have the same type, they may have different sets of attributes. In this case, we decide to use the attribute similarity, instead of the exact attribute set, to cluster different objects. Our clustering process follows two steps: first, we create a new class for each attribute, and put all its value objects into the class; second, based on the similarity of attributes, we split classes into smaller ones or merge them into larger ones.

1) Clustering Value Objects: In the first step, we use the attribute name to cluster its `value` objects. Intuitively, if two objects can be used as the value of the same attribute, they must have similar high-level semantics. Specifically, we will check every (name,object) pair: if we find a new attribute name, we will create a new class, and put the object into the class; if we have seen the attribute name, we will find the existing class and put the object there. Based on our experiment, most objects are used as the `value` of other objects, and therefore this method can handle objects well. For objects that are not used as values, we simply put them into a special class. In Equation 2, four objects $\texttt{object}_x$, $\texttt{object}_y$, $\texttt{object}_z$ and $\texttt{object}_w$ are used as values of the same attribute $\texttt{name}_a$. Therefore, we will create a $\texttt{name}_a$ class, and put these four objects there.

$$\begin{aligned} O_i &:= \{..., name_a = object_x, ...\} \\ O_j &:= \{..., name_a = object_y, ...\} \\ O_k &:= \{..., name_a = object_z, name_a = object_w ...\} \end{aligned} \tag{2}$$

Although the attribute-based clustering is reasonable, we observe two issues that decrease the result accuracy. First, attribute names could have overly general meanings, which renders the created classes contain semantically different objects. We need to split these classes into fine-grained ones. For example in Figure 1, attribute `/AA` means additional action, whose value is a `Dictionary` object containing a set of concrete actions. Based on the value-object clustering, we will put all such `Dictionary` objects into one class, no matter they contain actions for `/Page`, `/Field` or other significantly different objects. Second, semantically similar objects could be put into multiple classes due to different attribute names. For example, attributes `/Parents` and `/Kids` play similar roles in organizing PDF documents and can be manipulated by the same set of script APIs. We should put them together. However, due to the different attribute names, they are split into two classes.

2) Clustering with Attribute Similarity: We rely on the common attributes between objects to split existing classes into smaller ones or merge them into larger ones. Note that our method does not distinguish large or small classes, but simply applies the algorithm to all existing classes for splitting or merging. Specifically, we use the Dice coefficient to gauge the similarity, shown in Equation 3. A and B are the attributes sets of two objects; $|A|$ is the element number in A while $|B|$ is the element number in B; $A \bigcap B$ produces the set of common attributes of A and B. Intuitively, a larger Dice coefficient means that A and B have more common attributes, indicating their stronger underlying connections. If $sim(A, B)$ is larger than a threshold θ, we will put them into the same class; otherwise, we will separate them into two classes. The threshold can be defined empirically to achieve the optimal fuzzing result. In our experiment, we use two different thresholds for splitting and merging to achieve fine-grained clustering results. Even if that brings in extra classes, the relationship inference algorithm should be able to capture them. Based our empirical evaluation, we set θ to 0.3 for splitting and use θ=0.7 for merging.

$$sim(A, B) = \frac{2\left(|A \bigcap B|\right)}{|A| + |B|} \tag{3}$$

Equation 3 could bring heavy calculation when we extend it to support classes: we have to calculate the Dice coefficient for each member of the Cartesian product of two classes, which may take a lot of time and resources to finish. To simplify the calculation, we decide to use the *high-frequency* attributes to represent all objects in a class. We identify a high-frequency attribute if it is used by at least half of all objects in the class. With the high-frequency attribute, we can still use Equation 3 for calculation: A and B now represents the set of high-frequency attributes of the classes. Note that, we will update the high-frequency attributes after adding a new object into the class, which means the result could depend on the order of object analysis. If we cannot find any high-frequency attribute from one class, we will not take any more action on the class.

Now, we consider the aforementioned examples where the simple attribute-based clustering fails. First, for value objects of `/AA`, since each `Dictionary` object is defined with different actions, we can use attribute similarity to further classify them. For example, `/Page` objects only accept open and close actions, so we can get a new class to hold them. `/Field` objects accept keystroke, format, validate and calculation actions, and we successfully split them into a new class. Second, regarding `/Parents` and `/Kids` objects, we find that most of them internally contain `/Parents` and `/Kids` attributes. Therefore, we combine these two classes as one.

Real-world Statistics. We applied our clustering method on 16,000 collected PDF files. The algorithm splits 12,374,420 objects into 901 classes. The value-object clustering introduces 169,973 classes; the high-frequency attribute-based method further splits them into 209,453 classes; we remove classes that have less than 64 objects; after merging, we have 901 classes ready for the next-step analysis.

B. Statistical Relationship Inference

To support cooperative mutation, we take a statistical method to infer the relationship between script APIs and various object classes. Note that our goal is to obtain the possible relationship through light-weight methods. Confirming inferred relations would take a large amount of reverse engineering efforts and thus is out of the scope of this paper. Our inference mechanism has three steps: interface recognition, execution logging and relationship inference.

1) Interface Recognition: In the first step, we manually analyze the official manual, specifically interface definitions and API references, to recognize APIs that can access objects defined in documents. Usually, these APIs are well organized in the language specification, and we can identify them quickly by scanning the specification. Figure 5a show an example simplified from the JavaScript for Acrobat API Reference [1]: API `getAnnot()` returns all `Annotation` objects in a PDF file. The argument is the document object (*i.e.*, `this`). If the file has no `Annotation` objects, this API will return `NULL`. Line 2 shows the way to modify the `strokeColor` property of the

```
1  var annots = this.getAnnot();
2  annot[0].strokeColor = color.blue;
```

(a) Original JS code

```
1 + try{
2     var annots = this.getAnnot();
3 +   app.alert(annots.length + " Annots Found");
4     annot[0].strokeColor = color.blue;
5 + } catch(e) {
6 +   app.alert("ERROR: " + e);
7 + }
```

(b) After instrumentation

Fig. 5: Example JavaScript code for accessing PDF annotations. (a) Line 1 gets the list of all available annotation objects, and line 2 updates the `strokeColor` attribute of the first annotation to `blue`. (b) We insert extra code to the original JS to inspect the execution results.

first `Annotation` object. JavaScript has both read and write permissions here.

Coverage of Mutable Properties. It is worthwhile to note that not all object properties accessible by script APIs can be modified — some are read-only to scripts. For example, based on the reference [1], Adobe totally exposes 578 properties to JavaScript and only 314 (*i.e.*, 54.2%) of them can be modified. Other exposed properties can only be modified through direct object mutation. Previous work that merely mutates scripting code and does not change any documents cannot explore binding code that interacts with such immutable objects.

To reduce the mutation space of script APIs, we adopt the method proposed in `favocado` [11] to group APIs based on their high-level semantics. The intuition is that functions and properties in the same section of the manual should work for similar high-level semantics, like updating annotations or checking form fields. Based on API groups and the language specification, we construct templates corpus for code generation.

2) Execution Logging: With API groups, we will construct simple testing scripts that use APIs to access native objects. Our system can automatically insert these scripts into each provided document. It opens each modified document with the tested application, and records the execution result of the embedded script. To record the execution result, our scripts must contain two logging operations. First, after one invocation of any API, we will use a proper statement to save the return value, like through printing functions or saving to log files. A successful call indicates that the document contains objects that are necessary for the current API to complete. Second, to record unexpected errors that throw exceptions, we insert fault handling code to hook exceptions and log the error message. This step produces a set of tuples in the form of `(document,API,result)`, where `document` identifies the tested file, `API` means the related APIs and `result` indicates the execution result. Figure 5b shows the JS code updated from the original version in Figure 5a. Code at line 3 prints the return value, while code at lines 1, 5, 6, and 7 hook exceptions and save the error message.

3) Relationship Inference: With object classes and execution results, we try to map each script API to related object classes. Our system takes Algorithm 1 to achieve this goal. First, for each API f we find all related execution results, including successes (denoted as S_f, line 4) and failures (denoted as F_f, line 5). Then, for each object class c, we find all the files that contain at least one object in class c (denoted as IN, line 7). Next, we calculate, among all successful files, the ratio of files

Algorithm 1: Relationship Inference Algorithm

```
Input: ExecResult = {(doc, API, res)},
       ObjClassSet = {ObjClass}
Output: RelationMap = {API → {(ObjClass, rate)}}
1  APIset = {API | (doc,API,res) ∈ ExecResult}
2  for API f ∈ APIset:
3      RelationMap[f] = ∅
4      $S_f$ = {(doc,API,res) ∈ ExecResult | API=$f$,res=success}
5      $F_f$ = {(doc,API,res) ∈ ExecResult | API=$f$,res=fail}
6      for class c ∈ ObjClassSet:
7          IN = {doc | ∃ object o ∈ doc, o ∈ c}
8          $S_{IN}$ = {(doc,API,res) ∈ $S_f$ | doc ∈ IN}
9          $F_{IN}$ = {(doc,API,res) ∈ $F_f$ | doc ∈ IN}
10         $rate_S = |S_{IN}| / |S_f|$
11         $rate_F = |F_{IN}| / |F_f|$
12         $rate = rate_{Succ} - rate_{Fail}$
13         if $rate > \delta$:
14             RelationMap[$f$].add((c,$rate$))
15 return RelationMap
```

that contain any objects in c (line 8, 10), denoted as $rate_S$; similarly, we also calculate among all failed files, the ratio of files that contain any objects in c (line 9, 11), denoted as $rate_F$. Finally, if the difference of $rate_s$ and $rate_F$ is larger than a threshold δ, we will add class c to the relationship set of API f, together with $rate$. We can tune δ to filter out less-related classes. In our experiment, we set it to 0 to keep all classes.

Handling Common Objects. $rate_S$ tells among all files supporting the current API, how many of them contain objects in the current class. A higher $rate_S$ indicates a stronger connection. However, this method may fail in the case of *common objects*. A common object exists in almost all normal files, like `/Pages` and `/Resources` in the PDF format. Therefore, $rate_S$ of common objects is usually very high (even reaches 100%), which will mislead our algorithm to conclude that these objects are strongly related to every API. To mitigate this problem, we further consider $rate_F$, which shows among all files not supporting the current API, how many of them contain any object in the current class. If the ratio is also high, it indicates that such objects are likely to be common objects. Therefore, we take the difference of $rate_S$ and $rate_F$ to rank the inferred relationship for each API.

C. Relationship-Guided Mutation

In the last step, we utilize the inferred relationship between object classes and API groups to guide the generation of new files. First, we choose one group of APIs as the testing target. If these APIs have entries in the `RelationMap` map (results of relationship inference in Algorithm 1), we will perform cooperative mutation on both scripts and objects. Otherwise, we randomly choose scripts and objects for mutation. We take a standard method to generate script code that invokes selected APIs. Second, we mutate the objects related to these APIs. Given APIs, we can find a set of object classes from `RelationMap`, where each class has a $rate$ value that represents its relationship with these APIs. We distribute our fuzzing energy based on the $rate$ value. Specifically, we add all $rate$ values of related object classes together, and normalize them by diving each $rate$ by the sum. Equation 4 defines the probability that the class c gets selected for mutation, where the API includes all selected APIs while the i covers all classes related to these APIs. Therefore, classes with higher $rate$s are likely to

```
1  !begin lines
2  <Annotation pointer>.alignment=<int_Annotation_alignment>;
3  <new int>=<Annotation pointer>.alignment;
4  !end lines
5
6  !begin block annot_handler array_name
7  if (<array_name>.length <lt> n){ //<lt> represents '<' character
8    :expand add_annots
9    <array_name>=this.getAnnots();
10 }
11 for(var i = 0;i <lt> <array_name>.length;i++){
12   :fuzzall Annotation <array_name>[i]
13 }
14 !end block
```

Fig. 6: Templates for creating new script codes. Statements between `lines` (*e.g.*, line 2 and 3) represent individual lines, while statements between `blocks` (*e.g.*, line 7-13) should be used as a whole; `new int` indicates a variable of type `int`; `<A>` represents a symbol `A` that can be expanded with the user-defined rules.

be selected for mutation. After selecting the class, we randomly choose several objects from the class for mutation.

$$P(class\ c) = \frac{rate(c)}{\sum_{API}\sum_{i} RelationMap[API][i].rate} \quad (4)$$

1) Script Code Generation: We takes a traditional method to generate script code, similar to the grammar-based fuzzer `domato` [14]. It takes language grammars, statement formats, predefined templates and other information such as type and value scope as inputs, and creates a script that uses the selected APIs. Figure 6 shows one simple code template we use to generate JavaScript code for manipulating PDF's `Annotation` objects. To create the corpus and templates, we initially scan the API declarations from the manual of binding code and then manually edit the API operation templates to make them more likely to generate valid behavior sequences. Additionally, we expand several strategies to create statement blocks and make the operation logic more complex, such as `expand` and `fuzzall` operations. Lines 7-13 of Figure 6 show an example of code blocks, which we will use as a whole for the code generation. We leave the details of these strategies to §IV.

In this paper we focus on cooperative mutation, and do not try to improve the script generation process. Therefore, we use any other advanced generators here as long as the generated code contains selected APIs. For example, the state-of-the-art work, `favocado`, focuses on generating valid, semantically correct JavaScript code for testing the binding code [11]. We can plug this tool into our system to improve the quality of generated scripts. Our cooperative mutation techniques are general enough to collaborate with contemporary or even future script generators to handle different bindings code.

2) Object Mutation: Our system modifies document objects based on the generated script. Specifically, for objects related to current APIs, we insert, delete and modify properties to trigger different behaviors. We also randomly update common objects in the document to change the global states of the program.

Attribute Mutation. Our basic mutation strategy modifies each attribute (i.e., one name-value pair). First, we randomly choose one attribute of one object and *replace* it with another attribute from other objects within the same class. The large number of samples will provide diverse attributes for each class of objects. Second, we randomly choose one attribute and *insert* it into other objects within the same class. Since attributes could be optional, the insertion operation may increase the number of object properties. Third, we *delete* optional attributes from the object to reduce its properties. At last, we *replace the value* of one attribute with other values from other attributes.

Whole-object Mutation. Other than the fine-grained attribute mutation, we also modify whole objects to improve the mutation efficiency. This is particularly necessary when an object is never used as a value. The whole-object mutation also includes three operations: object replacement, insertion and deletion. For replacement, we randomly select another object within the same class to replace the current one. We retain the identifier of the object so that all references to the old object now refer to the new one. For deletion, we randomly choose one object and replace its content with random bytes. Meanwhile, we locate all references to the old object and replace them with `Null`. For insertion, we only add new elements into `Array` objects, where the new objects come from the classes of existing objects.

Universal Mutation. We randomly update API-unrelated, common objects to change the global program states. We identify common objects during the relationship inference §III-B. Specifically, if one class of objects have a high value in both $rate_S$ and $rate_f$, we treat them as common objects. Our mutator accordingly performs two mutations. First, we insert well-known interesting values in order to trigger boundary conditions. For example, for integers, the mutator replaces them with infamous values such as the maximum and the minimum values of integers; for strings, we replace them with `Null` objects and randomly insert `0` to the sequence of characters. Second, we modify the size of particular objects. For strings and arrays, we set their length to zero or an extremely large value. In object level, we delete all attributes in them or duplicate existing attributes to the maximum number.

IV. Implementation

We implement COOPER, our prototype of cooperative mutation, with 4.3K lines of code in Python. In particular, we modified `PyPDF2` [42], an open-source PDF parser, to parse PDF files into objects and construct a graph of mutual references between objects. For DOCX file, we use the python library `zipfile` to decompress the file into hierarchical directory files. Then, we use the python library `xml` to modify the file structure.

Insert JavaScript Code into PDF Files. We modify `PyPDF2` python library to parse the original PDF file and find its first page object. Then, we create an `opening` JavaScript action to hold the generated JavaScript code, and insert the action into the first page. When the file is opened, the PDF viewer (*e.g.*, Adobe Acrobat) will display the first page, which will trigger the embedded `opening` event and execute the JavaScript code.

Insert VBA Code into DOCX Files. Microsoft provides the Automation mechanism for one application to modify the objects implemented in another application [36]. With this mechanism, we can use the `win32com` module in `pywin32` [43] to insert VBA code to a DOCX file and save the file in the DOCM format (only the DOCM format accepts VBA code). However, if we directly save the updated file, Microsoft Word will rewrite the content based on its internal settings, like adding new tags or changing the file structure. The rewriting could reduce the diversity of the seed inputs, which are generated by various tools in different versions. Even worse, it modifies the particular file structure that we intentionally create during the cooperative mutation. To avoid such unexpected modifications, we first generate the VBA code, and use `win32com` to insert it into an empty DOCX file, named `a.docx`. Next, we extract the VBA-related components from `a.docx`, like `word/vbaData.xml`, `word/vbaProject.bin`, and `word/_rels/vbaProject.bin.rels`. At last, we insert these modules into the original DOCX file, modify VBA-related configuration and save it as a new DOCM file. When the DOCM file is opened, the embedded VBA code will get executed. In this way, all the mutation steps are under our control and the resulting files are expected.

Get Script Execution Result. Our relationship inference module in §III-B relies on the execution result of the script code to correlate objects and script. However, we did not find a general interface to retrieve the execution results of JavaScript and VBA. Instead, we use the popup-window APIs to obtain such information. For the PDF format, API `app.alert(string)` pops up a window to present the message of `string`. We insert the `app.alert` call to JavaScript code of PDF files to show the result in the window (see Figure 5b for an example). Meanwhile, We use another process to detect the creation of the window and parse the window's content to get the script execution result. For Microsoft Word and VBA, the similar API is *MsgBox*, while for HTML and JavaScript, the counterpart is `alert`.

Block-level Template for Script Generation. From the previous experience, we observe that existing single-line template for code generation [11, 14] is relatively monotonous, and difficult to form a complicated script sample. We introduce block-level template consisting of multiple lines, which allows us to easily construct code with complicated loops and condition statements. We further design two extension commands to extend the block template. The first one is "`expand subblock_name`", which inserts another block into the current block template, like the `include` primitive in C language. The second command is "`fuzzall objtag objname`", which covers all single-line operations related to `objtag` in the current place. `objname` is a name that already defined in the current context. In the `objtag`-related template, it will replace `objtag` tag with the existing variable name. The purpose is to fully test all properties and methods of the object in a specific context (*e.g.*, loop).

These extensions help generate complicated code. For example in Figure 6, we use the block template `annot_handler` to throughly test each annotation in the document. We check whether the number of annotations is less than a threshold. If so, it will use the `add_annots` block to add more annotations into the sample. After that, it will iterate every annotation and use all line-templates related to `Annotation` to test it. The complexity finally helps find more bugs. Like we will show in §V-B1, the vulnerability in Figure 8 can be easily triggered by this block, through iteratively accessing the first two annotations, and sequentially modifying their attributes and invoke their methods. Without the block template, it is unlike for single-line template to form such an API access sequence.

Supporting New Languages. At the early stage of COOPER development, it takes one of our authors about two weeks to support JavaScript in PDF documents. Lately, the same author spent two days to support VBA for DOCX files. We believe supporting more scripting languages for other programs will

Program	Format	Language	Script	Source	Free
Adobe Acrobat	PDF	C++	JavaScript	✗	✗
Foxit Reader	PDF	C++	JavaScript	✗	✓
Microsoft Word	DOCX	C++	Visual Basic	✗	✗

TABLE I: Programs used for evaluation. The "language" means the programming language for implementing the program.

take less than one-week effort. Here we list the necessary manual efforts to apply COOPER to a new language. (1) For object clustering, users can reuse existing parsers to parse sample documents to get object attributes. Then, COOPER could group native objects automatically. It is usually easy to find open-source parsers for widely used file formats. For example, we use `PyPDF2` to parse PDF files, and use `zipfile` and `xml` to parse DOCX files. All of them are open-source tools and can be found online. (2) For statistical relationship inference, users need to prepare a piece of API-testing scripts for COOPER to infer the relationship. To prepare API-testing scripts, users can find object-related APIs from the language specification. These APIs are well organized in the specification, and users can even find example code snippets. (3) For relationship-guided mutation, users need to prepare some templates and can also reuse the templates of existing tools or specifications. To collect templates, we can use a crawler to retrieve API sample code from the specification.

V. EVALUATION

In this section, we evaluate COOPER on real-world programs to understand its strength regarding the following aspects:

Q1. Can COOPER find new vulnerabilities from real-world programs that adopt scripting languages? (§V-B)
Q2. How accurate can COOPER cluster different objects into semantically similar classes? (§V-C1)
Q3. Can COOPER infer reasonable relationships between document objects and script APIs? (§V-C2)
Q4. Does the cooperative mutation find more unique bugs than other configurations and existing tools? (§V-D)
Q5. Can COOPER explore more unique code coverage than other configurations and existing tools? (§V-E)

A. Evaluation Setup

Target Programs. We select three widely used programs that adopt scripting languages as the testing targets. Table I shows the program details. Adobe Acrobat and Foxit Reader are PDF viewers and editors; both are developed with the unsafe language C++, and embed JavaScript engines to support JS code. Microsoft Word can process DOCX files, and permits scripts written in Visual Basic for Applications (VBA) to modify files. All programs are proprietary, where the source code is not available and users may need to purchase them with expensive cost. As they are popular targets of advanced attacks, most of them have bounty programs to reward bug reporters.

Sample Collection. We collect a large number of documents from public resources as the native inputs to the tested programs. To test Adobe Acrobat and Foxit Reader, we use the Google search engine to fetch 16,000 normal PDF files, which contain a total of 12,374,420 `Dictionary` objects. Based on our analysis, 90% of the samples contain about 1,290 `Dictionary` objects, but in the extreme case, one PDF sample contains 151,460 `Dictionary` objects. To test Microsoft Word, we search in Google and collect 18,000 normal DOCX files, which contain 288,617,886 *`Tag` objects* — the basic elements in XML files. 90% of the samples contain about 30,046 `Tag` objects, but in the extreme case, one DOCX file contains 3,606,234 `Tag` objects.

Experiment	Relation Guidance	Object Mutation	Script Generation
COOPER-full	●	●	●
COOPER-random	○	●	●
COOPER-object	●	●	○
COOPER-script	○	○	●
`Domato`	○	○	◐

TABLE II: Experiment configurations. ● indicates feature enabled; ○ means feature disabled; ◐ means non-default feature.

Experiments Design. To answer questions `Q4` and `Q5`, we design five experiments that test the selected programs with different configurations, shown in Table II. COOPER-full is the full-featured COOPER, which utilizes the inferred relationship to select objects and APIs, and modifies both through cooperative mutation. COOPER-random does not use the relationship-based guidance, but just randomly chooses and mutates objects and APIs. COOPER-object only modifies objects that are related to binding code, while COOPER-script merely changes the script and does not mutate any object. The last experiment is conducted using the existing JavaScript fuzzer `Domato`, which only modifies the JavaScript code in a slightly different way from COOPER-script. Since `Domato` is design to test general JavaScript code, we add our JS templates for it to generate PDF documents and VBA templates to generate DOCX files.

Evaluation Setup. We conducted our experiments on three servers, each with 32 Intel(R) Xeon(R) CPU E5-2630 V3@2.40GHZ cores, 64GB RAM and 64-bits Ubuntu 14.04 TLS. To fully utilize all resources, we create multiple virtual machines (VMs) where each has two cores and 4GB RAM. For the long-term bug-finding experiment in §V-B, we have used eight VMs to test Adobe Acrobat and Foxit Reader for four months, and got eight VMs to test Microsoft Word for one week. To compare different configurations and tools on bug detection (§V-D) and code coverage (§V-E), we tested each program for one week and repeated the experiment five times. All experiments start with the same initial seed set. To catch all heap corruptions, we enabled full PageHeap [35].

B. Summary of Bug Finding

COOPER successfully found 134 unique, previous-unknown bugs during the four-month testing. Table III shows the bug details and lists script APIs that trigger these bugs. We have responsibly reported all of them to their developers. At the time of paper publishing, 103 of them have been confirmed and 59 of them have been fixed. 33 fixed bugs are assigned CVE numbers due to severe security impacts. From column `Type` we can see that COOPER can identify bugs in common types of memory issues, including stack buffer overflow, heap buffer overflow, use-after-free, null pointer dereference, and so on. Attackers can use these bugs to run arbitrary code or steal sensitive information (shown in column `Impact`). 53 bugs have "High" severity, indicating that they are highly likely exploitable. Other "Moderate" bugs may lead to denial-of-service attacks.

	ID	Type	Impact	Severity	Status	APIs
Adobe Acrobat	1	use-after-free	arbitrary code execution	High	CVE-2020-3748	Annot.page
	2	use-after-free	arbitrary code execution	High	CVE-2021-21035	Annot.popupOpen ...
	3	use-after-free	arbitrary code execution	High	CVE-2021-21033	Annot.setProps
	4	use-after-free	arbitrary code execution	High	CVE-2021-21028	Annot.getProps ...
	5	use-after-free	arbitrary code execution	High	CVE-2021-21021	Doc.getAnnots
	6	use-after-free	arbitrary code execution	High	CVE-2021-35981	App.LanchURL
	7	use-after-free	arbitrary code execution	High	CVE-2021-28635	Doc.addField
	8	heap buffer overflow	arbitrary code execution	High	CVE-2021-28638	Doc.zoomType
	9	stack buffer overflow	arbitrary code execution	High	CVE-2020-3799	Doc.getNthFieldName ...
	10	buffer error	arbitrary code execution	High	CVE-2020-9698	-
	11	buffer error	arbitrary code execution	High	CVE-2020-9699	-
	12	buffer error	arbitrary code execution	High	CVE-2020-9700	-
	13	buffer error	arbitrary code execution	High	CVE-2020-9701	Doc.getLegalWarnings
	14	buffer error	arbitrary code execution	High	CVE-2020-9704	Doc.exportAsFDFStr
	15	heap buffer overflow	arbitrary code execution	High	CVE-2021-28561	Doc.zoomType
	16	null pointer deference	denial-of-service	Moderate	CVE-2021-39849	Annot.stateModel
	17	null pointer deference	denial-of-service	Moderate	CVE-2021-39850	Annot.setProps ...
	18	null pointer deference	denial-of-service	Moderate	CVE-2021-39851	Annot.popupOpen
	19	null pointer deference	denial-of-service	Moderate	CVE-2021-39852	Field.getItemAt ...
	20	null pointer deference	denial-of-service	Moderate	CVE-2021-39853	-
	21	null pointer deference	denial-of-service	Moderate	CVE-2021-39854	Doc.zoomType
	22	stack exhaustion	denial-of-service	Moderate	CVE-2020-9702	Doc.getLegalWarnings
	23	stack exhaustion	denial-of-service	Moderate	CVE-2020-9703	Doc.layout ...
	...	stack exhaustion	denial-of-service	Moderate	Confirmed	Doc.exportAsFDFStr ...
	52	use-after-free	arbitrary code execution	High	Confirmed	-
	...	use-after-free	arbitrary code execution	High	Confirmed	Annot.vertices ...
	56	use-after-free	arbitrary code execution	High	Reported	Doc.removeField ...
	57	buffer error	arbitrary code execution	High	Reported	-
	58	heap overread	memory leakage	High	Reported	Doc.zoomType
	59	instruction acces violation	arbitrary code execution	High	Reported	Annot.popupOpen
	60	instruction acces violation	arbitrary code execution	High	Reported	Annot.transitionToState ...
Foxit Reader	1	use-after-free	arbitrary code execution	High	CVE-2021-31441	Annot.destroy
	2	use-after-free	arbitrary code execution	High	CVE-2021-31451	Annot.destroy
	3	use-after-free	arbitrary code execution	High	CVE-2021-31456	Annot.popupOpen ...
	4	use-after-free	arbitrary code execution	High	CVE-2021-31457	Annot.destroy
	5	use-after-free	arbitrary code execution	High	CVE-2021-31458	Annot.destroy
	6	use-after-free	arbitrary code execution	High	CVE-2021-34831	Field.richText ...
	7	use-after-free	arbitrary code execution	High	CVE-2021-34832	Annot.readonly ...
	8	use-after-free	arbitrary code execution	High	CVE-2021-34852	Field.delay ...
	9	use-after-free	arbitrary code execution	High	CVE-2021-34974	Annot.delay ...
	10	use-after-free	arbitrary code execution	High	CVE-2021-34975	Annot.trasitionToStat ...
	...	use-after-free	arbitrary code execution	High	Confirmed	Doc.pageNum ...
	19	heap buffer overflow	arbitrary code execution	High	Confirmed	Doc.deletePages
	20	heap overread	memory leakage	High	Confirmed	Bookmark.createChild
	...	heap overread	memory leakage	High	Fixed	Doc.getField ...
	27	buffer error	arbitrary code execution	High	Fixed	Field.signatureValidate
	28	stack exhaustion	denial-of-service	Moderate	Fixed	Field.textColor
	...	stack exhaustion	denial-of-service	Moderate	Fixed	Doc.deletePages ...
	32	null pointer deference	denial-of-service	Moderate	Fixed	Annot.popupOpen ...
	...	null pointer deference	denial-of-service	Moderate	Fixed	Doc.getPageLabel ...
	46	null pointer deference	denial-of-service	Moderate	Reported	Doc.deletePages ...
	...	null pointer deference	denial-of-service	Moderate	Reported	Doc.doNotScroll ...
	54	instruction acces violation	arbitrary code execution	High	Fixed	Annot.fillColor ...
	55	instruction acces violation	arbitrary code execution	High	Confirmed	Annot.readonly ...
	56	security check failure	denial-of-service	Moderate	Confirmed	Doc.addAnnot ...
Microsoft Word	1	use-after-free	arbitrary code execution	Moderate	Reported	Range.TCSCConverter ...
	...	use-after-free	arbitrary code execution	Moderate	Reported	Paragraph.Style ...
	4	heap overread	memory leakage	Moderate	Reported	Range.InsertXML ...
	...	heap overread	memory leakage	Moderate	Reported	ActiveWindow.Panes ...
	9	null pointer deference	denial-of-service	Moderate	Reported	Range.FormattedText ...
	...	null pointer deference	denial-of-service	Moderate	Reported	Pane.NewFrameset ...
	17	memory error	denial-of-service	Moderate	Reported	Range.Duplicate ...
	18	instruction acces violation	arbitrary code execution	Moderate	Reported	Range.Previous ...

TABLE III: Deteced bugs: 134 bugs from 3 applications with 2 scripting languages. These bugs fall into 10 categories and 40% have high impact on the application security. We have reported all of them to their developers. 59 bugs have been fixed and 33 of them were assigned CVEs. ... means additional APIs are needed to trigger the bug (Table VIII has the complete API list).

All these bugs are triggered using 90 script APIs in 11 object classes. Table VIII in Appendix provides more details of the bug-triggering APIs for each detect bug.

Adobe Acrobat. We detected 60 new bugs from Adobe Acrobat, including 12 use-after-free, one heap buffer overflow and one stack buffer overflow. Among them, 23 vulnerabilities

```
1  %PDF-1.3
2  1 0 obj << /Pages 2 0 R >> endobj
3  2 0 obj << /Kids [ 3 0 R 4 0 R ] >> endobj
4  3 0 obj << /AA << /O << /S /JavaScript
5                          /JS 7 0 R>> >>
6             /Annots [ 5 0 R ] >> endobj
7  4 0 obj << /Parent 2 0 R >> endobj
8  5 0 obj << /Popup 6 0 R /NM ()
9             /Subtype /Circle >> endobj
10 6 0 obj << /NM () >> endobj
11 7 0 obj << /Length 237 >>
12         stream
13           var annot=this.getAnnots()[0];
14           annot.setProps(annot.getProps());
15           annot.page=1;
16         endstream
17     endobj
18 trailer << /Root 1 0 R >>
```

Fig. 7: Simplified PoC of CVE-2021-21028, a use-after-free vulnerability in Adobe Acrobat Reader DC Version-2020.012.20048.

have been assigned CVEs and 15 of them are marked as *critical* by the vendor, indicating that this vulnerability allows attackers to execute arbitrary malicious code. Such vulnerabilities include use-after-free, heap overflow, untrusted point dereference, stack-based overflow, and buffer error. We are awarded $18K bounty due to seven of these exploitable vulnerabilities.

Foxit Reader. COOPER identified 56 vulnerabilities in Foxit Reader, including 18 use-after-free, one heap overflow and seven heap overread. We have reported all these bugs. 10 use-after-free vulnerabilities have been assigned CVEs and marked as *critical* by the vendor. We received $4K bug bounty.

Microsoft Word. Microsoft Word has many inter-content elements. Due to the time limit, we only tested APIs `Paragraph`, `Table`, `Range` and `Pane` for one week. The quick test revealed 18 unique bugs, including three use-after-free, five heap overread, eight null pointer reference, one memory error and one access violation bug. We have reported all of them to Microsoft.

1) Case Studies: We inspect several vulnerabilities to help understand their root causes and security consequences.

Bug Triggered with Empty Names. Figure 7 shows a simplified PoC of CVE-2021-21028, a use-after-free vulnerability in Adobe Acrobat. This PDF file contains two pages represented by object 3 and object 4. The first page has an annotation object 5, which we call the main annotation. Object 5 refers to another annotation object 6 through attribute `/Popup`, which we call the popup annotation. The first page has an additional action, which will be executed once the page is opened. The vulnerability is triggered by the JavaScript code at line 15. To generate such a bug-triggering input from normal PDF files, we need to conduct two-dimensional mutation: the main annotation and the popup annotation should both have an empty name (`/NM ()` at line 8 and 10); the script should reset the properties of the main annotation (lines 13 to 15) and move the annotation to the second page (line 15). Our manual debugging and analysis reveal the root cause: a heap block was allocated when Acrobat parses the PDF objects; at line 15, JavaScript engine uses the binding code to invoke the native `Annots.api` module, which frees the heap block and accesses it with a dangling pointer, leading to the use-after-free vulnerability.

Bug Triggered with Abnormal Actions. Figure 8 shows the simplified PoC of CVE-2021-21035, another use-after-free vulnerability in Adobe Acrobat. This PDF file has two pages and the first page contains three annotation objects 5, 6 and 7.

```
1  %PDF-1.3
2  1 0 obj << /Pages 2 0 R >> endobj
3  2 0 obj << /Kids [ 3 0 R 4 0 R ] >> endobj
4  3 0 obj << /AA <</O <</S /JavaScript
5                          /JS 8 0 R >> >>
6             /Annots [ 5 0 R 6 0 R 7 0 R ] >> endobj
7  4 0 obj << /Parent 2 0 R >> endobj
8  5 0 obj << /Subtype /Caret >> endobj
9  6 0 obj << /T (Total Improvement area Y)
10            /Subtype /FreeText >> endobj
11 7 0 obj << /Action /GoTo/GoTo >> endobj
12 8 0 obj << /Length 401 >>
13         stream
14           var a0 = this.getAnnots()[0];
15           var a1 = this.getAnnots()[1];
16           a0.setProps({type:"Polygon",page:1,});
17           a0.popupOpen=true; a0.popupOpen=false;
18           a1.setProps({type:"Polygon",page:1,popupRect:[ ... ]});
19           a1.popupOpen=true; a1.popupOpen=false;
20         endstream
21     endobj
22 trailer << /Root 1 0 R >>
```

Fig. 8: Simplified PoC of CVE-2021-21035, a use-after-free vulnerability in Adobe Acrobat Version-2020.012.20048.

Metadata	class id: 449; object count: 335482 leadattributes: `/Pg`, `/Kids`, `/O`, `/Dest`, `/D`, `/Names`, `/OpenAction`
High-frequency Attributes	**Values**
`/Type`: 335482	`/Page`: 335482
`/Parent`: 335480	`id_504`: 335480
`/Contents`: 335296	`id_10`: 298364, `<array of id_10/440>`: 36923
`/Resources`: 334701	`id_527/524/525/...`: 334701
`/MediaBox`: 329356	`<array of 0/1/2/...>`: 329356
`/Rotate`: 232937	0: 221864, 90: 8776, 270: 1337, ...
`/CropBox`: 226735	`<array of 0/1/2/...>`: 226735
`/StructParents`: 124785	`<array of 0/1/2/...`: 124785
`/Tabs`: 89893	`/S`: 85482, `/W`: 4364, `/R`: 42, `/A`: 5
`/Annots`: 58027	`<array of id 456/742/127/506/...>`
Other high-frequency attributes: `/Group`: 84705, `/BleedBox`: 75765, `/TrimBox`: 70210, `/ArtBox`: 57511, `/Thumb`: 20773, `/B`: 9670, `/Trans`: 5669, `/PieceInfo`: 4864, `/ID`: 3273, `/LastModified`: 2306, ...	

TABLE IV: Details of object class 449, which contains 335,482 objects, commonly used as values of attribute in `leadattributes`. `Attributes` and `Values` show the popular attributes within the objects. Digits are the number of objects containing such attributes.

Similarly, triggering this bug requires to mutate both objects and script code: in the object dimension, the third annotation object 7 contains an `/Action` attribute with an abnormal value `/GoTo/GoTo`; in the script dimension, we need to repeatedly set the properties of the first two annotations, and change their `popupOpen` property twice (lines 17 and 19). Finally, the bug is triggered by the native module `AcorRd32.dll`, invoked by the JavaScript engine through the binding layer.

C. Qualitative Analysis of Clustering and Inference

Object clustering and relationship inference are prerequisites to cooperative mutation. However, due to the large number of objects and script APIs, it is impractical to quantitatively measure the clustering and inference results (we do not have the ground truth). To understand the rationality of our method, we manually inspect some intermediate results to demonstrate the consistency between our insight and the outcome. Specifically, we check one object class produced by the clustering process, inspect its high-frequency attributes, and try to compare them with the object definition in the format specification. We also analyze the `RelationMap` of one concrete script API, and verify its quality by cross-checking the API manual.

Attribute	Type	Value
Type	name	required, must be Page
Parent	dictionary	required, parent of this page
LastModified	date	required if PieceInfo is present
Resources	dictionary	required, resources required by page
MediaBox	rectangle	required
CroBox	rectangle	optional
BleedBox	rectangle	optional
TrimBox	rectangle	optional
ArtBox	rectangle	optional
BoxColorInfo	rectangle	optional
...	...	...

TABLE V: Entries of `Page` objects. The PDF format defines 30 entries (attributes). We show the first 10 here due to the space limit.

1) Object Clustering: The clustering process on 12,374,420 PDF objects finally leads to 901 object classes. We pick one class with a large number of objects and compare its content with the corresponding description in the PDF specification. Table IV shows the details of this class, which has ID 449 and contains 335,482 objects. In the collected PDF files, these objects are used as values of several attributes (`leadattributes` in the table), like `/Pg` and `Kids`. The table also presents the high-frequency attributes. For example, all objects have attribute `/Type`, whose values are always `/Page`. All but two objects have attribute `/Parent`, and the value objects belong to another class with ID 504. 98.2% of objects have attribute `/MediaBox`, and the value objects are arrays of numbers, like `[0 0 10 9]`.

Based on our knowledge of the PDF format, this class should be related to `Page` objects. Table V shows the first 10 attributes of `Page` objects defined in the official document (another 20 skipped). By cross-checking with the specification, we confirm that all high-frequency attributes shown in Table IV are legitimate for the `Page` object. Values of different attributes are also consistent with the specification. For example, `Type` is required for each `Page` object, and its value must be `/Page`. All objects in class 449 satisfy this requirement.

We also identify non-`Page` objects from this class (false positives) and found that their percentage is very low. For example, among all attributes that do not belong to `page`, `eCpyResolution` has the maximum number (744) of objects. However, based on the PDF specification, we did not find any object should have this attribute. It is likely an attribute used by third-party applications. By default, Acrobat will ignore such unknown attributes and thus having them in the class will not affect the program execution. From another perspective, `MediaBox` is a required attribute for each `Page` object. Table IV shows that 98.2% of all objects in the class have this attribute, meaning invalid objects only account for 1.8%.

2) Relationship Inference: `RelationMap` maps each API to a set of object classes, where each class has an associated mutation probability. We inspect the APIs related to annotation objects in Acrobat and Foxit Reader to understand the rationality of our inference algorithm. Table VI shows the inference details. Based on the attribute of each related class, we manually search the PDF documentation and JavaScript reference to confirm whether they are related to Annotations or not. Since two results are similar, we focus on analyzing the one for Adobe Acrobat. In this result, the most relevant class has attribute `/AP`. According to the PDF specification, `/AP` is a valid attribute in the annotation dictionary, referring to the appearance object.

Prog	Succeed (s)		Failed (f)		Diff	Attributes
	in	rate	in	rate		
Adobe Acrobat (s 108, f 15611)	102	0.944	420	0.026	0.917	/AP
	103	0.953	3676	0.235	0.718	/Fm45, /Fm44, ...
	66	0.611	46	0.002	0.608	/Annots
	83	0.768	3703	0.237	0.531	/Resources
	59	0.546	1693	0.108	0.437	/AcroForm
	58	0.537	1677	0.107	0.429	/DR
	...	...	...	...	...	...
Foxit Reader (s 174, f 15778)	156	0.897	388	0.025	0.872	/AP
	158	0.908	3673	0.232	0.675	/Fm45, /Fm44, ...
	107	0.614	5	0.000	0.614	/Annots
	130	0.747	3732	0.236	0.511	/Resources
	93	0.534	1694	0.107	0.427	/AcroForm
	92	0.528	1676	0.106	0.422	/DR
	...	...	...	...	...	...

TABLE VI: Relation between object classes and annotation APIs. `Succeed` and `Failed` are the execution result of injected scripts. `in` means files that contain objects in the current class.

Although `/AP` can also be used in other objects (*e.g.*, `field`), our further analysis on 12,374,420 objects reveals that other uses are very rare. The second class with attribute `/FMxx` are names of `formXObject` objects, which can participate in forming the appearance of annotations. The third class with attribute `Annots` refers to an array of annotations in a page object — the main form to include annotation objects in one PDF file.

Summary. Our manual analysis reveals that the results of object clustering and relationship inference match our intuition. Although the analysis is not complete, we will use them to guide the testing process. As we will show in §V-D, the relationship-guided COOPER outperforms all other configurations.

D. Unique Bug Finding

To understand the contribution of COOPER components, we tested Adobe Acrobat and Foxit Reader using five different configurations shown in Table II. We ran each experiment for one week (i.e., seven days, 168 hours). For reported crashes, we combine call stacks and manual efforts to remove duplicated bugs. Figure 9 shows the number of real bugs, while Table VII shows the bug types and bug IDs (defined in Table III).

Benefit of Relationship-guided Mutation. The full-featured COOPER outperforms all other configurations in detecting unique bugs. Specifically, it triggers 18 unique bugs in Adobe Acrobat and 14 unique bugs in Foxit Reader within one week. COOPER-random that randomly but simultaneously mutates objects and JavaScript code detects 12 unique bugs in Adobe Acrobat, 9 bugs in Foxit Reader. Therefore, the relationship guidance helps find 50% more unique bugs for Adobe Acrobat, and helps detect 55.6% more unique bugs for Foxit Reader.

Benefit of Two-dimensional Mutation. COOPER-object uses inferred relationship to guide the object mutation, but it does not change any JavaScript code. COOPER-script does not modify any object and just mutates JavaScript code. After one-week testing, the former finally reports 4 bugs in Adobe Acrobat and 3 bugs in Foxit Reader, while the latter identifies 8 bugs in Adobe Acrobat and 5 bugs in Foxit Reader. Compared with results from COOPER-full (18 bugs from Acrobat, 14 bugs from Foxit), such results demonstrate the necessity of the two-dimensional mutation: over object-only testing, cooperative

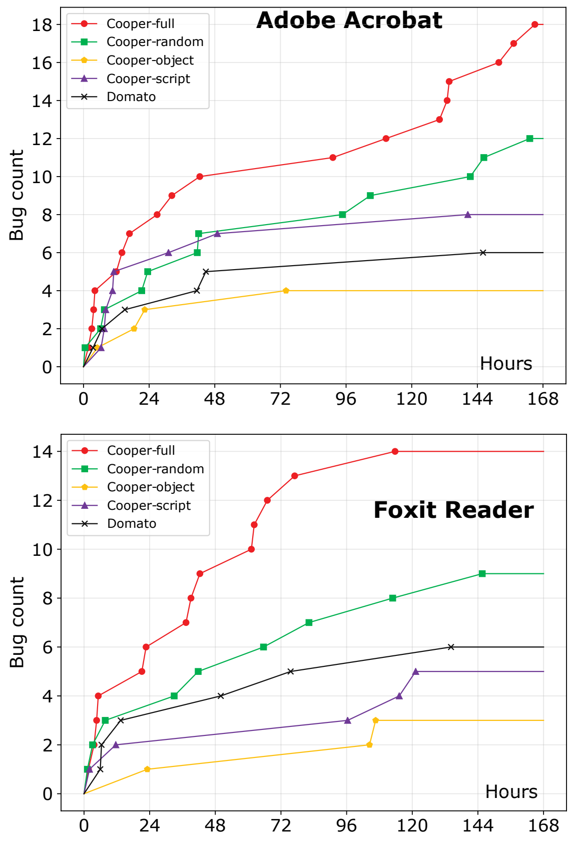

Fig. 9: Unique bugs with different configurations. We tested Adobe Acrobat and Foxit Reader with five different settings shown in Table II. Each experiment is conducted for one week.

mutation helps find 3.5× more unique bugs from Acrobat, and helps find 3.7× more real bugs from Foxit; over script-only mutation, cooperative mutation helps find 1.3× more unique bugs from Acrobat and 1.8× more unique bugs from Foxit.

An interesting observation is that even with the relationship-based guidance, object-only mutation COOPER-object has the worst performance. Our understanding is that (1) Adobe Acrobat has been extensively tested [20, 32, 61] and thus it is challenging to find bugs in the native code; (2) Object-only mutation can hardly trigger binding code as the interfaces are invoked through scripting languages. Therefore, we should mutate both objects and scripts to test the binding code.

Comparison with Domato. The existing JavaScript fuzzer, `Domato`, detected 6 bugs from Adobe Acrobat and 6 bugs from Foxit Reader. It is not surprising that `Domato` found fewer bugs than the full-featured COOPER as it merely modifies JavaScript code and never touches native objects. `Domato` even fails to compete with COOPER-random, which does not use any relationship to guide object and API mutation. This result is consistent with our observations among different COOPER configurations: both two-dimensional mutation and relationship guidance are necessary to improve the bug-finding efficiency.

Venn Diagram. Since it is difficult to draw Venn Digram for five sets, we use Table VII to show the relationships between bugs detected by different configurations and tools. For both Adobe Acrobat and Foxit Reader, the full-featured COOPER covered almost all bugs found by other configurations and

Target	ID†	Type	full	random	object	script	Domato
Adobe Acrobat	26	Stack exhaustion	✔	✗	✔	✗	✗
	37	Stack exhaustion	✔	✔	✔	✗	✗
	55	Use-After-Free	✔	✔	✗	✔	✗
	32	Stack exhaustion	✔	✔	✗	✔	✗
	33	Stack exhaustion	✔	✔	✗	✔	✗
	56	Use-After-Free	✔	✔	✗	✔	✔
	39	Stack exhaustion	✔	✔	✗	✔	✔
	40	Stack exhaustion	✔	✔	✗	✔	✔
	41	Stack exhaustion	✔	✔	✗	✗	✔
	21	Null pointer deref	✔	✔	✗	✗	✔
	43	Stack exhaustion	✔	✔	✗	✗	✗
	45	Stack exhaustion	✔	✗	✔	✗	✗
	16	Null pointer deref	✔	✗	✗	✔	✔
	49	Stack exhaustion	✔	✗	✗	✔	✗
	53	Use-After-Free	✔	✗	✗	✗	✗
	54	Use-After-Free	✔	✗	✗	✗	✗
	58	Heap overread	✔	✗	✗	✗	✗
	47	Stack exhaustion	✔	✗	✗	✗	✗
	24	Stack exhaustion	✗	✔	✗	✗	✗
	18	Null pointer deref	✗	✔	✗	✗	✗
	48	Stack exhaustion	✗	✗	✔	✗	✗
Foxit Reader	36	Null pointer deref	✔	✔	✔	✗	✗
	38	Null pointer deref	✔	✔	✔	✗	✗
	32	Null pointer deref	✔	✔	✗	✔	✗
	11	Use-After-Free	✔	✔	✗	✔	✔
	39	Null pointer deref	✔	✔	✗	✔	✔
	40	Null pointer deref	✔	✔	✗	✗	✔
	24	Heap overread	✔	✔	✗	✗	✗
	17	Use-After-Free	✔	✗	✗	✔	✔
	30	Stack overflow	✔	✔	✗	✗	✔
	45	Null pointer deref	✔	✔	✗	✔	✗
	46	Null pointer deref	✔	✗	✗	✗	✗
	49	Null pointer deref	✔	✗	✗	✗	✗
	18	Use-After-Free	✔	✗	✗	✗	✗
	19	Heap overflow	✔	✗	✗	✗	✗
	42	Null pointer deref	✗	✗	✔	✗	✗
	43	Null pointer deref	✗	✗	✗	✗	✔

TABLE VII: Distribution of unique bugs found by different testing configurations. ID† means the bug ID in Table III.

tools, showing its advantage over others. Interestingly, COOPER-object always found one bug that is not covered by COOPER-full within one weak. We believe this is because the object-only mutation may reach some program states that cooperative mutation can hardly trigger. For example, if we infer that an object is not related to any API, we will mutate it with the lowest priority and could miss bugs related to this object. However, from the table we can see that such cases are rare (only one for each program). This observation also applies to all other configurations, as bugs found by COOPER-object have the minimal overlap with those of other tools, including `Domato` – the one having no relation with COOPER.

E. Code Coverage

Experiment Setup. Code coverage is commonly used to evaluate greybox fuzzing techniques. COOPER does not take code coverage as a feedback, but we can use it to understand the capability of COOPER in exploring program states. However, measuring the whole-program coverage does not make much sense, as we intentionally prioritize inputs that can explore the binding code. We should focus on the coverage of the binding code. Unfortunately, due to the close-source nature of the tested programs, we cannot directly distinguish binding code from others. To address this issue, we rely on the public interfaces of the script engine and call stacks to identify binding layer-related code. Specifically, we run each program using Intel

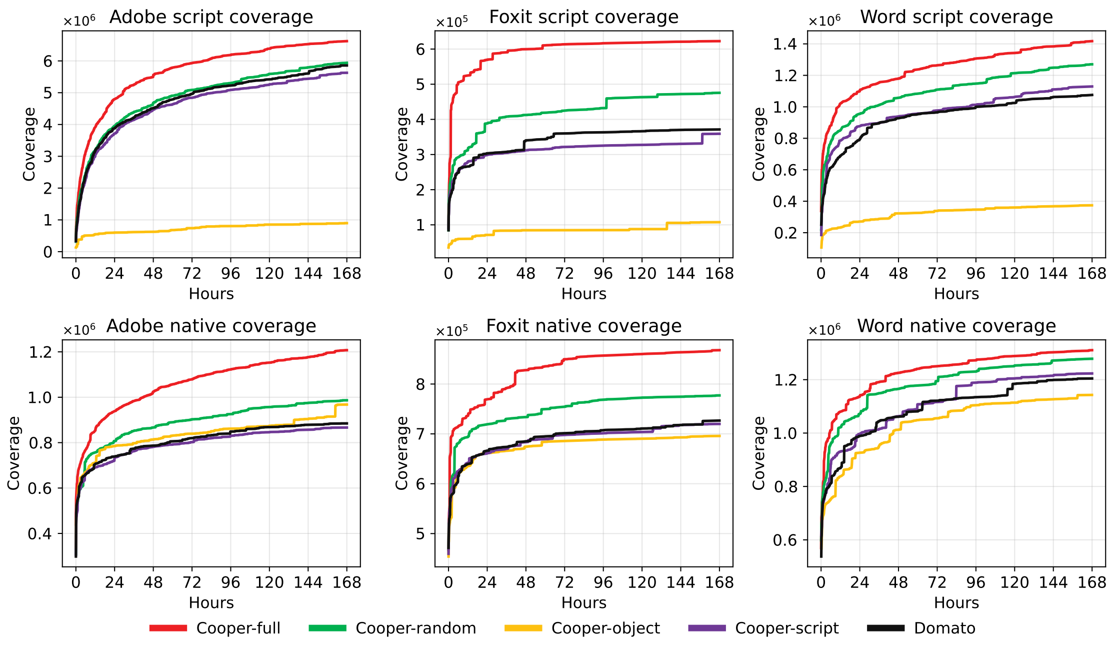

Fig. 10: Unique edges discovered by different configurations and tools. "script" means the code related to the script engine and the binding layer; "native" means code for other components. We record all documents generated by COOPER in one week, and count coverage offline.

PIN [34], a dynamic instrumentation tool, and for each branch, check whether the current call stack contains public interfaces of script engines. If such interfaces (i.e., `FXJSE_ExecuteScript` for Foxit, any function in module `EScript.api` for Acrobat, any function in module `VBE7.dll` for Word) are in the call stack, we treat the branch as related to the binding layer. Otherwise, the branch is merely for the native code. We allocate a unique ID to each branch to avoid the collision issue [17]. We first test each program with COOPER for one week and collect all generated documents. Then, we run the program with these documents again in PIN to collect the code coverage.

Figure 10 shows the results of the coverage measurement for three programs, including both script coverage and native coverage. We can see that the full-featured COOPER consistently outperforms other configurations and tools for both script code and native code. For example, when testing Adobe Acrobat, COOPER achieves 11.5%, 637.2%, 17.7% and 13.0% more edge coverage on script code, compared to COOPER-random, COOPER-object, COOPER-script, and `Domato`. Meanwhile, it triggers 22.4%, 24.8%, 39.3% and 36.5% more edge coverage on native code compared to others. This result demonstrates that COOPER can improve the code coverage when testing programs that adopt binding layers, which could explain why it can detect so many severe vulnerabilities.

VI. RELATED WORK

We have explained `favocado`, the most related work and compared it with our system COOPER in previous sections. In this section, we discuss other works related to finding binding bugs using program analysis and fuzzing.

A. Program Analyses for Binding Bugs

Static program analyses. Static analyses are widely used to detect various bugs in the boundaries of *multilingual* programs (*i.e.*, programs written in more than one languages). For example, several works are proposed to statically analyze Java JNI code to detect mishandled exceptions [27, 30, 53]. Tan *et al.* utilize static analyses to detect type safety issues where C pointers are used in Java as integer [53]. To detect type safety issues, Furr *et al.* develop a system to automatically infer types for OCaml and Java foreign function interfaces (FFI) [15, 16]. Li *et al.* design static checks to identify reference counting issues in Python/C interfaces [31]. Brown *et al.* investigate bugs in JavaScript bindings of Chrome and Node.js and develop a set of static checkers to detect crashes, type-violations and memory-violations [6]. Although these checkers successfully found many exploitable bugs, they are limited in the analysis scope and bug types due to the embedded specific logics. For example, one of their memory-safety checkers focus on detecting dangerous uses of implicitly casted variables, which may permit time-of-check-to-time-of-use attacks to invalidate security checks. They cannot cover large code beyond the binding layer due to the limited scalability nor produce inputs to trigger the bug.

As a dynamic bug detector, COOPER differs from static analysis techniques in three ways. First, COOPER detect a large set of bugs that finally triggers program crashes or can be detected by security sanitizers [35]. Second, static analyses suffer from the scalability issues as they usually require inter-functional analysis. COOPER can handle large complicated systems, like browsers and commercial document processors. Third, COOPER does not report false alarms as every bug is detected at runtime with a concrete input. In contrast, static analyses have a higher false positive rate (*e.g.*, checkers [6] reported 30 false positives and 81 true bugs), and rely on manual efforts or other external tools to generate bug-triggering inputs.

Dynamic program analysis. `Jinn` detects bugs in foreign language interfaces (FFI) like Java Native Interface (JNI) and Python/C at runtime [28]. It first constructs state machines based on the deep understanding of the interface specification. Then, it instruments the program to automatically insert checks, which will validate the invocations of foreign language interfaces at runtime. COOPER differs from Jinn in two ways. First, it does not require interface-specific checks, but instead relies on the system protection (*i.e.*, crashes) and secure allocators [35] to detect memory errors. Second, it actively mutates the input of the program to trigger as many bugs as possible, while Jinn just passively detects bugs triggered by the given inputs.

B. Fuzzing for JavaScript Bugs

Fuzzing repeatedly generated random inputs to stress the tested program [20, 32, 37, 61]. It has been widely used to test and successfully found thousands of bugs from various systems, like operating systems [9, 26, 39, 58, 59], compilers and interpreters [8, 22, 40], web browsers [11, 19, 49, 60], network protocols [5, 41], smart contracts [38, 57] and so on. As one of the most hacked systems, JavaScript engines have been extensively fuzzed both in industry and academia.

The most straightforward way to test JavaScript engines is to generate JavaScript code from scratch based on the context-free grammar [14, 21, 45, 51]. However, this simple method can hardly create completely valid code (syntactically correct and semantically correct) nor reach deep program logic. Even worse, it requires tedious manual efforts to write grammar rules. One way to improve the testing efficacy is to use the code coverage to guide the input generation [3, 55]: a newly generated code is kept for future mutation only if it triggers fresh code paths of the JavaScript engine. In this way, the fuzzer will spend more efforts on the code-discovering inputs.

The second direction is to improve the syntax-correctness and semantic correctness of the generated code to avoid early termination. For example, LangFuzz combines fragments of given valid code to improve the syntax correctness [24]; Skyfire learns context-sensitive grammar from existing code samples [54]; CodeAlchemist maintains code context to make sure that variables are defined before being used [22]. Several works lift the JavaScript code into an intermediate presentation (IR) so as to comprehensive code analysis to generate high-quality test cases [8, 21, 60]. Among them, FreeDOM detects browser bugs by mutating all elements of web pages, including CSS styles, DOM tree and JavaScript statements [60]; PolyGlot develops a general mutation method towards generating valid test cases for different programming languages [8].

The third method obtains insights from old JavaScript bugs so as to produce high-quality code to trigger new vulnerabilities. For example, DIE inspects the proof-of-concepts (PoCs) of known bugs and preserves the type and structure in order to retain critical aspects [40]. Montage leverages neural network to train a language model to guide the test case generation [29].

Different from these tools, COOPER is designed to test the binding code where JavaScript runs inside another complicated commercial software. Due to the diversity of the underlying systems, especially in the non-browser environments, existing JavaScript engines cannot effectively test binding layer [11]. Despite the differences in design goals, we can further improve COOPER with existing JavaScript fuzzers. Specifically, the current implementation of COOPER relies on the code templates of Domato [14] to manipulate JavaScript and can still find a lot of severe bugs thanks to the cooperative mutation. We plan to adopt the coverage-based feedback and high-quality JavaScript mutator to improve the fuzzing efficacy.

VII. CONCLUSION

We propose cooperative mutation, a novel approach that tests the binding code of scripting languages to find memory-safety issues. Cooperative mutation simultaneously modifies the script code and the related native input to explore various code paths of the binding code. To support cooperative mutation, we infer the relationship between native inputs and script APIs, and use the relationship to guide the two-dimensional mutation. We applied our tool COOPER on three popular commercial software, Adobe Acrobat, Foxit Reader and Microsoft Word. COOPER detected 134 previously unknown bugs, which resulted in 33 CVE numbers and $22K bug bounty.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful feedback. We also thank Xiangkun Jia, Zheng Huang and Zhenkai Liang for their comments on earlier drafts of this paper. This research was supported, in part, by National Natural Science Foundation of China (Grand No. U1936211, U1836117, U1836113, 61902384 and 62102406), the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDC02020300.

REFERENCES

[1] Adobe Inc, "Developing Acrobat Applications Using JavaScript (version 02/01/2021)," https://opensource.adobe.com/dc-acrobat-sdk-docs/acrobatsdk/pdfs/acrobatsdk_jsdevguide.pdf, (accessed July 10, 2021).

[2] ——, "Document Management - Portable Document Format - Part 1: PDF 1.7," https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf, 2008.

[3] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for Deep Bugs with Grammars," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-Oriented Programming: A New Class of Code-reuse Attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, Hong Kong, China, Mar. 2011.

[5] boofuzz Developers, "boofuzz: Network Protocol Fuzzing for Humans," https://github.com/jtpereyda/boofuzz, (accessed July 10, 2021).

[6] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan, "Finding and Preventing Bugs in JavaScript Bindings," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[7] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-Control-Data Attacks Are Realistic Threats," in *Proceedings of the 14th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2005.

[8] 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," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, Virtual, May 2021.

[9] J. Choi, K. Kim, D. Lee, and S. K. Cha, "NTFUZZ: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, Virtual, May 2021.

[10] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou, "Heap Taichi: Exploiting Memory Allocation Granularity in Heap-spraying Attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, Austin, TX, Dec. 2010.

[11] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé *et al.*, "Favocado: Fuzzing the Binding

Code of JavaScript Engines Using Semantically Correct Test Cases," in *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*, Virtual, Feb. 2021.

[12] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing Binaries for Memory Safety Errors with QASan," in *Proceedings of 2020 IEEE Secure Development (SecDev)*, Virtual, 2020.

[13] Foxit, "Working with JavaScript using Foxit PDF SDK (Java)," https://developers.foxit.com/developer-hub/document/javascript-pdf-sdk-java/, (accessed July 11, 2021).

[14] I. Fratric, "Domato: A DOM Fuzzer," https://github.com/googleprojectzero/domato, (accessed July 10, 2021).

[15] M. Furr and J. S. Foster, "Checking Type Safety of Foreign Function Calls," *ACM SIGPLAN Notices*, 2005.

[16] ——, "Polymorphic Type Inference for the JNI," in *European Symposium on Programming*. Springer, 2006.

[17] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[18] Ghidra, "Ghidra Scripting," https://ghidra.re/courses/GhidraClass/Intermediate/Scripting_withNotes.html#Scripting.html, (accessed July 11, 2021).

[19] Google, "ClusterFuzz," https://google.github.io/clusterfuzz, (accessed July 10, 2021).

[20] Google, "Honggfuzz," https://google.github.io/honggfuzz/, (accessed July 10, 2021).

[21] S. Groß, "Fuzzil: Coverage Guided Fuzzing for JavaScript Engines," *Master's thesis, TU Braunschweig*, 2018.

[22] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[23] Hex Rays, "IDAPython Project for Hex-Ray's IDA Pro," https://github.com/idapython/src, (accessed July 10, 2021).

[24] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[25] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[26] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid Fuzzing on the Linux Kernel," in *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[27] G. Kondoh and T. Onodera, "Finding bugs in java native interface programs," in *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA)*, Amsterdam, The Netherlands, Jul. 2008.

[28] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley, "Jinn: Synthesizing Dynamic Bug Detectors for Foreign Language Interfaces," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Toronto, ON, Canada, Jun. 2010.

[29] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A Neural Network Language Model-guided Javascript Engine Fuzzer," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual, Aug. 2020.

[30] S. Li and G. Tan, "Finding Bugs in Exceptional Situations of JNI Programs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Nov. 2009.

[31] ——, "Finding Reference-counting Errors in Python/C Programs with Affine Analysis," in *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP)*, Uppsala, Sweden, Jul.–Aug. 2014.

[32] LLVM, "LibFuzzer - A Library For Coverage-guided Fuzz Testing," http://llvm.org/docs/LibFuzzer.html, (accessed July 10, 2021).

[33] LLVM, "Undefined Behavior Sanitizer (UBSan)," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, (accessed July 10, 2021).

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[35] D. Marshall and N. Schonning, "GFlags and PageHeap," https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap, (accessed July 11, 2021).

[36] Microsoft, Inc, "OLE Automation," https://docs.microsoft.com/en-us/cpp/mfc/automation?view=msvc-160, (accessed July 23, 2021).

[37] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, Dec. 1990.

[38] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Virtual, Jun.–Jul. 2020.

[39] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[40] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing JavaScript Engines with Aspect-preserving Mutation (to appear)," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[41] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNet: a Greybox Fuzzer for Network Protocols," in *Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, Virtual, Sep. 2020.

[42] PyPDF2, "PyPDF Documentation," https://pythonhosted.org/PyPDF2/, (accessed July 23, 2021).

[43] pywin32 Developers, "pywin32:Python for windows(pywin32)," https://github.com/mhammond/pywin32, (accessed July 23, 2021).

[44] M. Rajpal, W. Blum, and R. Singh, "Not All Bytes are Equal: Neural Byte Sieve for Fuzzing," *arXiv preprint arXiv:1711.04596*, 2017.

[45] J. Ruderman, "Introducing jsfunfuzz," https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, (accessed July 11, 2021).

[46] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proceedings of the 26th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2005.

[47] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jun. 2012.

[48] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data Race Detection in Practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, New York, NY, Dec. 2009.

[49] K. Serebryany, "Sanitize, Fuzz, and Harden Your C++ Code." San Francisco, CA: USENIX Association, 2016.

[50] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct.–Nov. 2007.

[51] W. Snyder and M. Shaver, "Building and Breaking the Browser," *Black Hat USA*, Aug. 2007.

[52] A. Sotirov, "Heap Feng Shui in JavaScript," *Black Hat Europe*, Mar. 2007.

[53] G. Tan and J. Croft, "An Empirical Security Study of the Native Code in the JDK," in *Proceedings of the 17th USENIX Security Symposium (Security)*, San Jose, CA, Jul.–Aug. 2008.

[54] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven Seed Generation for Fuzzing," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[55] ——, "Superion: Grammar-aware Greybox Fuzzing," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Surabaya, Indonesia, May–Jun. 2019.

[56] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "MAZE: Towards Automated Heap Feng Shui," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual, Aug. 2021.

[57] V. Wüstholz and M. Christakis, "Harvey: A Greybox Fuzzer for Smart

Contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, Virtual, Nov. 2020.

[58] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data Race Fuzzing for Kernel File Systems," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[59] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing File Systems via Two-Dimensional Input Space Exploration," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[60] W. Xu, S. Park, and T. Kim, "FREEDOM: Engineering a State-of-the-Art DOM Fuzzer," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, FL, Nov. 2020.

[61] M. Zalewski, "American Fuzzy Lop (2.52b)," http://lcamtuf.coredump.cx/afl, (accessed July 10, 2021).

[62] C. Zapponi, "GitHut: a Small Place to Discover Languages in GitHub," https://githut.info/, (accessed July 11, 2021).

Appendix

APIs	Bug IDs	APIs	Bug IDs	APIs	Bug IDs
Adobe Acrobat		**Foxit Reader**		**Microsoft Word**	
Annot.page	1,4,55	Annot.destroy	1,2,3,4,5,7,9,10,14, 16,17,30,32,54,55	Paragraph.Range	1,2,3,4,5,6,7,9,10, 11,12,17,18,15
Annot.popupOpen	2,17,18,24,53,54,59	Annot.popupOpen	3,32	Paragraph.LineSpacingRule	10
Annot.setProps	2,3,4,17,43,53	Annot.readOnly	7,16,55	Paragraph.TextboxTightWrap	8
Annot.getProps	4,53	Annot.delay	9	Paragraph.InsertAlignmentTab	2
Annot.vertices	53	Annot.quads	14	Paragraph.Alignment	7
Annot.noView	53	Annot.trasitionToState	10	Paragraph.SelectNumber	11
Annot.intent	54	Annot.borderEffectIntensity	15	Paragraph.RightIndent	13
Annot.rect	55	Annot.fillColor	54	Paragraph.Style	2,12
Annot.stateModel	16	Annot.hidden	9	Range.TCSCConverter	1,5
Annot.popupRect	23	Field.richText	6	Range.SortAscending	4,15
Annot.points	43	Field.value	6,8,11,26	Range.GoToNext	5
Annot.repeat	51	Field.signatureValidate	27	Range.FormattedText	7,9,10
Annot.destroy	53,54	Field.setFocus	18,35,41,50,51	Range.PhoneticGuide	10,17
Annot.delay	53	Field.exportValues	47	Range.WordOpenXML	10,17
Annot.richContents	55	Field.rotation	50	Range.GetSpellingSuggestions	2,12
Annot.transitionToState	60	Field.delay	8	Range.ImportFragment	17
Doc.getAnnots	5	Field.textFont	11	Range.Previous	18
Doc.addField	7	Field.textColor	28,49	Range.CheckSynonyms	3
Doc.zoomType	8,15,21,33,40,41,58	Field.doNotScroll	44	Range.TwoLinesInOne	14
Doc.getNthFieldName	9	Field.comb	50	Range.InsertXML	4
Doc.layout	23,41,54,55	Field.readonly	18	Range.Next	5,6,18
Doc.addAnnot	53,54,60	Doc.embedDocAsDataObject	6,11,12,13,47	Range.Relocate	6,10,11
Doc.removeField	56	Doc.zoomType	45	Range.Text	9
Doc.exportAsFDFStr	14,44	Doc.removeDataObject	13,24,53	Range.HorizontalInVertical	10
Doc.pageNum	41,55	Doc.removeField	26	Range.InsertAfter	10
Doc.resetForm	19,31,46	Doc.resetForm	26,43,46	Range.Duplicate	2,12,17
Doc.getField	9	Doc.selectPageNthWord	48	Range.SortByHeadings	18
Doc.zoom	54,55	Doc.zoom	12,45	Range.AutoFormat	7
Doc.getLegalWarnings	13,22,29,32,39,42,47,49	Doc.pageNum	13	Range.InsertParagraphAfter	7
App.LaunchURL	6	Doc.getField	25	ActiveWindow.Panes	8,16
Field.page	9	Doc.getPageLabel	37	Pane.Previous	8,16
Field.getItemAt	19,44,56	Doc.getAnnots	54	Pane.Next	8,16
Collab.documentToStream	25	Doc.addAnnot	39,40,56	Pane.NewFrameset	8,16
AcroForm.AFSimple_Calculate	34	Bookmark.createChild	20		
		AcroForm.AFNumber_Keystroke	18		
		Doc.deletePages	18,19,24,25,31,45, 46,48,49,51,52		

TABLE VIII: Objects and APIs relevant to bugs found by COOPER.