Operand-Variation-Oriented Differential Analysis for Fuzzing Binding Calls in PDF Readers

***Abstract*—Binding calls of embedded scripting engines introduce a serious attack surface in PDF readers. To effectively test binding calls, the knowledge of parameter types is necessary. Unfortunately, due to the absence or incompleteness of documentation and the lack of sufficient samples, automatic type reasoning for binding call parameters is a big challenge. In this paper, we propose a novel operand-variation-oriented differential analysis approach, which automatically extracts features from execution traces as oracles for inferring parameter types. In particular, the parameter types of a binding call are inferred by executing the binding call with different values of different types and investigating which types cause an expected effect on the instruction operands. The inferred type information is used to guide the test generation in fuzzing. Through the evaluation on two popular PDF readers (Adobe Reader and Foxit Reader), we demonstrated the accuracy of our type reasoning method and the effectiveness of the inferred type information for improving fuzzing in both code coverage and vulnerability discovery. We found 38 previously unknown security vulnerabilities, 26 of which were certified with CVE numbers.**

***Index Terms*—binding call, PDF reader, type reasoning, fuzzing**

I. Introduction

PDF (Portable Document Format) is the de-facto standard for electronic document exchange [1]. It has become an appealing target for the adversaries. Many attacks have been found that exploit vulnerabilities hidden in different components of famous PDF readers [2]. Finding vulnerabilities in a PDF reader requires comprehensive testing. Besides the underlying parsers and renders, the embedded scripting engines are also critical test subjects. According to Vulners, 17% of the PDF vulnerabilities submitted to the Zero Day Initiative (ZDI) program in recent three years are related to the embedded scripting engines [3].

The embedded scripting engine provides the access to the native functionalities via a set of specialized programming interfaces (named *binding calls* [4]), which are implemented in low-level, unsafe languages such as C and C++. A widely-used technique for testing scripting engine is fuzzing, which runs strategically generated test cases for finding vulnerabilities. To effectively fuzz the embedded scripting engine, both the knowledge of grammar rules and binding call parameter types are necessary. While the grammar rules can be automatically extracted from the standard language specification [5], [6] or learned from the vast number of existing samples [7], [8], automatically obtaining the parameter types of binding calls is not an easy task.

On the one hand, *there is no standard specification for PDF binding calls*. Different vendors have different implementations and some of them do not even release public documentation. To the best of our knowledge, only Adobe provides freely available API reference manual [9]. Although well-organized and human-readable, the manual is incomplete. More than 47% of the binding calls are entirely absent. Even for the documented binding calls, the manual does not enumerate all possible parameters and their various types.

On the other hand, *the existing samples are inadequate for learning the parameter types of PDF binding calls*. Most PDF test suites [10], [11] mainly focus on testing the parsers and renders, with little attention paid to the embedded scripting engines. Even in Mozilla's PDF.js test suite [12], which contains over 600 PDF files that embed script code, less than 10% of the binding calls are covered.

A natural way to infer binding call parameter types is to perform differential analysis [13] on execution traces. Specifically, the target binding call is executed for multiple times, each with different values of different types. The expected parameter types are inferred by comparing the observable features extracted from execution traces. Previous studies use some easy-to-obtain features, such as error message [14] and path length [15].

However, *shallow execution features are not suitable for type reasoning of PDF binding calls*. In fact, for the consideration of robustness, embedded scripting engines always provide a great degree of fault tolerance. About 60% of Adobe binding calls and 70% of Foxit ones implicitly convert type-incorrect parameters to the expected types with default values assigned. This makes it very difficult, if not impossible, to find significant distinction between type-correct executions and type-incorrect executions in terms of shallow features.

To address the problem, we propose a novel differential analysis approach based on the execution features with sufficient discrimination that can be used as oracles for type reasoning. First, we present a concept of *type indicators*, which capture how instruction operands vary when passing different parameter values of expected types. Compared with error message and path length, type indicators carry more execution semantics. Then, we develop an algorithm to identify type indicators from various execution traces. The algorithm aggregates the differential results of each individual binding call parameters, merging the ones summarized from the *may-type-correct* executions and excluding those from the *must-type-incorrect* executions. Finally, the possible types of a binding call parameter are inferred by checking the occurrence of the expected type indicators in the corresponding differential results. The inferred type information is used as fuzzing guidance for generating more effective test cases.

We implemented a prototype system called TypeOracle and evaluated it on two popular PDF readers (i.e., Adobe Reader [16] and Foxit Reader [17]). The experimental results

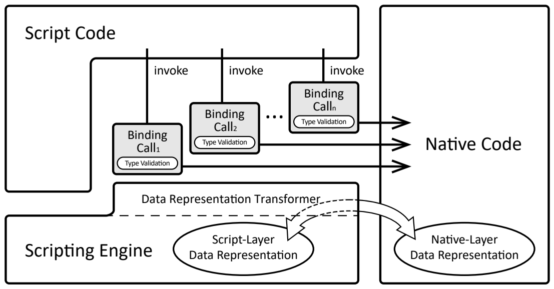
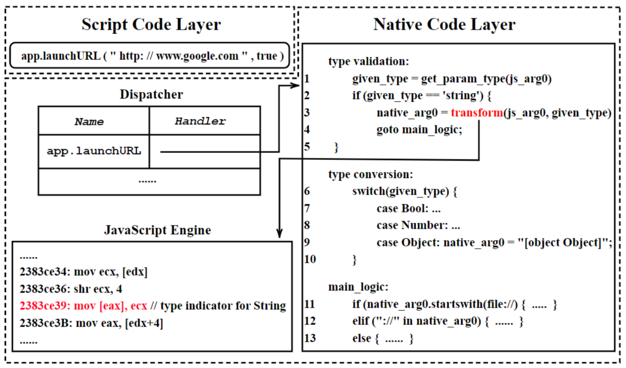


Fig. 1: Binding calls bridge script code and native code.



```
app.launchURL ( " http: // www.google.com " , true )
```

```
......
2383ce34: mov ecx, [edx]
2383ce36: shr ecx, 4
2383ce39: mov [eax], ecx // type indicator for String
2383ce3B: mov eax, [edx+4]
......
```

```
   type validation:
1      given_type = get_param_type(js_arg0)
2      if (given_type == 'string') {
3          native_arg0 = transform(js_arg0, given_type)
4          goto main_logic;
5   }

   type conversion:
6      switch(given_type) {
7          case Bool: ...
8          case Number: ...
9          case Object: native_arg0 = "[object Object]";
10     }

   main_logic:
11     if (native_arg0.startswith(file://) { ..... }
12     elif ("://" in native_arg0) { ...... }
13     else { ...... }
```

Fig. 2: Invocation processing logic of `app.launchURL`.

show that TYPEORACLE can effectively infer the parameter types of binding calls with high accuracy (99% precision and 96% recall). The inferred type information leads to significant improvement on code coverage (40% and 35% increment compared with random testing and fuzzing with API documentation respectively) and the discovery of 38 zero-day vulnerabilities (26 of which were certified with CVE numbers).

This paper makes the following contributions.

- We proposed a novel approach to extract observable features from the execution traces for inferring the parameter types of binding calls. We will open-source the prototype system to the community upon publication [1].
- We systematically studied the differences in parameter types of binding calls between documentation and implementation, as well as between different PDF readers. We found the undocumented and inconsistent binding calls are likely vulnerable since they may not get fully tested.
- We responsibly analyzed and disclosed all vulnerabilities found by TYPEORACLE and helped the vendors with the patch development. All the submitted vulnerabilities have been confirmed and got fixed.

II. BACKGROUND

PDF readers embed scripting engine to allow the document creators to programmatically invoke advanced functionalities when the document readers are viewing the PDF file. The advanced functionalities include constructing interactive forms, adjusting document appearance, etc. These functionalities are provided in native code. To enable the invocation from the script level to the native level, PDF readers add new internal objects and their accompanying methods and accessors (getters and setters) to the scripting engine. Calling these methods or getting/setting these accessors are dispatched to the underlying native handlers. These methods and accessors are named as binding calls. Specifically, a `getter` is analogous to a function with no parameter and non-void return value, and a `setter` is analogous to a function with a single parameter and void return value.

Figure 1 shows how binding calls are used as the bridge to connect script code and native code. When invoking a binding call via script code, the scripting engine is responsible for translating the data representation of the parameters and the native code is responsible for validating the type correctness. Binding calls share the same translation logic but differ in the type validation logic. Different binding calls may have different fault tolerance hence perform different actions for type-incorrect parameters (e.g. type conversion, using default value or throwing an exception).

The mainstream PDF readers use Javascript as the scripting language. The parameter types of binding calls are a subset of the Javascript data types. These data types can be divided into two categories: primitive (including `Boolean`, `Number` and `String`) and composite (including `Array` and `Object`). The composite types are created by using primitive types.

As an example, Listing 1 shows a simplified two-page PDF file that contains script code, which will be triggered once the file is opened. Line 5 invokes the `launchURL` method (that requires a `String` parameter and a `Boolean` parameter) to open a website or a local file. Line 6 sets 200% zoom level by writing the `zoom` accessor (that requires the `Number` value) of the `doc` internal object (*this*). If a binding call is vulnerable, an attack could embed the exploit script in a PDF file and deceive the users to open the file to trigger the vulnerability.

Listing 1: Simplified PDF file that contains script code.

```
1  1 0 obj <</Pages 2 0 R /OpenAction 4 0 R>> endobj
2  2 0 obj << /Kids [3 0 R 5 0 R] /Count 2 >> endobj
3  3 0 obj << /Type /Page /Parent 2 0 R >> endobj
4  4 0 obj << /Type /Action /S /JavaScript /JS (
5      app.launchURL("http://www.google.com", true);
6      this.zoom = 200; ) >>
7  5 0 obj << /Type /Page /Parent 2 0 R >> endobj
8  trailer << /Size 5 /Root 1 0 R >>
```

Figure 2 presents the invocation of `app.launchURL`. When the binding call is invoked at the script code layer, the control is dispatched to the corresponding native handler. The handler contains three parts of functionalities: type validation, type conversion and main logic. Type validation checks whether the type of the given parameter is as required. If so, it invokes the utility function provided by the scripting engine to translate data representation (Line 3). Otherwise, it adopts various type conversion strategies to implicitly convert the given parameter

[1]The experiment data and the instructions to access the data is available at https://github.com/TypeOracle/TypeOracle. Please refer to the README.md in the github repository.

to the required type (Lines 6-10). We can see that if we provide the 0th parameter of `app.launchURL` with a non-string value, some code in the main logic (e.g., Lines 11-12) will never get executed.

III. Motivation

To see why inferring binding call parameter types is important for testing, we present the discovery of a zero-day vulnerability in Adobe Reader as an example. This vulnerability was awarded $900 bounty. Listing 2 shows the minimized triggering code. The first invocation of the binding call `Collab.registerReview` (Line 1) allocates an 8-byte memory region for an internal structure, which is pointed to by a global pointer. The second invocation (Line 2) allocates another piece of memory region and releases the previous allocated one. However, the global pointer is not updated in time to point to the new memory region. The released memory region is accessed via the global dangling pointer during the invocation of the binding call `this.getAnnot` (Line 3), causing a use-after-free violation.

Listing 2: Motivating example.

```
1 Collab.registerReview(this,{},"","","","","","",
      false,"",false,"123","","",false,"",1.2);
2 Collab.registerReview(this,{},"","","","","","",
      false,"",true,"123","","",true,"",1.2);
3 this.getAnnot(1,"abc");
```

As we can see, the key to triggering the vulnerability is to feed crafted yet type-correct parameter values to the `registerReview` and `getAnnot` binding calls. We should note that `registerReview` is undocumented, neither could we find its invocation instance in the existing test suites [10]–[12]. Randomly selecting parameter values without any type guidance will cause combinatorial explosion, making the testing very ineffective. The search space will be dramatically reduced if the parameter types can be inferred.

Through manually reverse-engineering the implementation of `registerReview`, we found that it takes more than 17 parameters, 3 of which are critical to the vulnerability. In particular, it expects a `Boolean` value for the 8th parameter, a `String` value for the 11th parameter, and a `Number` value for the 16th parameter. It has different fault tolerance strategies for individual parameters. Specifically, it tries to stringify the non-string value passed to the 12th parameter (e.g., converting to "true"/"false" for `Boolean` and "[object Object]" for `Object`) and numerify the non-number values passed to the 16th parameters (e.g., converting to 1.0 for `true`, 123.45 for "123.45" and `NaN` for the values that cannot be converted to numbers).

Fault tolerance strategies are binding-call-specific. Different binding calls have different implicit type conversion rules even for the same type. For example, while `registerReview` implicitly converts non-number values to floats, `getAnnot` converts them to integers. The implicit type conversion does not generate any hint and it does not guarantee that type conversion (for type-incorrect parameters) has longer or shorter execution path than data transformation (for type-correct parameters). As a result, there is no significant distinction between type-correct executions and type-incorrect executions in terms of error message and path length. In other words, we cannot use these two shallow features for type reasoning.

Given the large number of binding call parameters, it is impractical to manually reverse-engineer their conversion or extraction methods. There is a need for automatic approach to type reasoning. The key is to extract execution features with sufficient discrimination for type reasoning.

Observations. Our automatic solution is inspired by three observations. First, *although different binding calls may have different processing logic for type-incorrect parameters, they share the same processing logic for type-correct parameters*. In fact, once the type validation of parameters is passed or tolerated, the binding calls will invoke the utility functions provided by the scripting engine to transform data representation. The utility functions are unique for each type. Second, *the data transformation of each type can be perceived via differential analysis on multiple executions with different values of the given type*. Essentially, differential analysis provides a light-weight method to understand how the input is processed by the program. The instructions whose operand values vary in multiple executions are related to the input processing. Third, *the type-consistency feature of scripting language provides valuable type-related knowledge*. For an accessor of a built-in object, the return type of its `getter` function should be consistent with the parameter type of its `setter` function, since they operate on the same virtual property. Such a feature can be used for distinguishing between may-type-correct and must-type-incorrect differential results.

Our Technique. Following the above observations, we propose an *operand-variation-oriented differential analysis* approach to identify binding call parameter types. Specifically, we extract the variation patterns between operands and parameters for each data type from the differential results on the accessors of the type under consideration, and infer the parameter type of a given method by checking the occurrence of the expected variation patterns in the corresponding differential results. The variation patterns are formally defined as *type indicators* as detailed in the next section.

As an example, the type indicator of `String` is related to the `mov` instruction at address $0x2383ce39$ (Figure 2). When we change the value of a `String` parameter from "*AAA*" to "*bbbbb*", the value of the *ecx* operand in the `mov` instruction will change from $0x3$ to $0x5$ (related to string length). For the 12th parameter of `registerReview`, we found the `String` type indicator appears in the differential result where we feed the binding call with different `String` values. Hence it is inferred as a `String` parameter. The types of the 9th and 17th parameters are identified in a similar way. During fuzzing, when generating the test case that involves the `registerReview` binding call, we will assign its parameters with values of the inferred types. The aforementioned vulnerability is finally triggered.

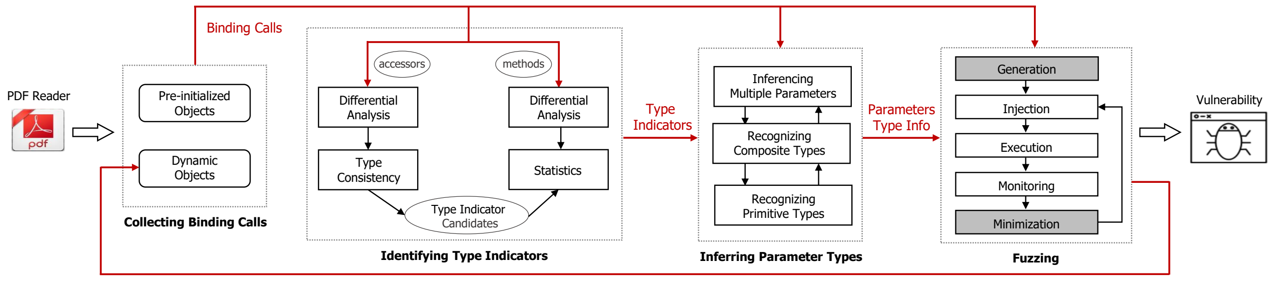


Fig. 3: Workflow of TypeOracle.

IV. Methodology

Definition. As shown in Figure 4, a type indicator is formally defined as an implication relation between parameter variation of binding calls and operand variation of program instructions. The former (denoted as `ParVary`) is a triple $\langle par\_type, par\_val_1, par\_val_2\rangle$ indicating the value change from par_val_1 to par_val_2 on a binding call parameter whose type is par_type. The latter (denoted as `OpdVary`) is a quadruple $\langle insn, opd, opd\_val_1, opd\_val_2\rangle$ indicating the value change from opd_val_1 to opd_val_2 on the opd operand of the $insn$ instruction.

```
TypeIndicator ::= ParVary → OpdVary
      ParVary ::= ParType × ParVal × ParVal
      OpdVary ::= Insn × Opd × OpdVal × OpdVal

     ExeTrace ::= { Insn × Opd × OpdVal }
```

Fig. 4: Definitions.

Workflow. The workflow of our methodology is shown in Figure 3. Given a PDF reader, TypeOracle first runs script within the embedded scripting engine to collect the binding calls by traversing the internal objects (Section IV-A). Then, a two-stage operand-variation-oriented differential analysis is performed on the binary level to identify the most common type indicator for each primitive data type (Section IV-B). The identified type indicator is used for inferring the parameter type of each binding call (Section IV-C). The type information helps the fuzzer to generate more effective test cases that can reach deep code (Section IV-D).

A. Collecting Binding Calls

Collecting binding calls is the first step. The more binding calls we collected, the more comprehensive the fuzz testing could be. It should be noted that we cannot rely on Adobe's API reference manual to obtain the complete binding call list. As we can see in Section V-B, around 40% of binding calls in Foxit Reader are not implemented in Adobe Reader and more than 47% of binding calls in Adobe Reader are undocumented.

Noticing that binding calls are registered and attached to built-in objects, we try to collect them by iterating the properties (accessors and methods) of built-in objects. There are two kinds of built-in objects: pre-initialized ones created during program initialization and dynamically-generated ones created during program execution. The traverse on pre-initialized objects is straight-forward. It starts from the root element of document (i.e., *this*) and recursively uses the `for-in` statement to iterate over properties of a given object.

For dynamically-generated objects, we should create them before iterating their properties. The creation requires invocation of certain binding calls with appropriate parameters. We postpone it to the fuzzing period where the parameter type information is ready. During fuzzing, if the invocation of a binding call (with type-correct parameters) returns a previously unseen built-in object, we will iterate through its properties.

B. Identifying Type Indicators

We leverage differential analysis to identify type indicators. Differential analysis is a technique that studies how the perturbations of program inputs have impact on the perturbations of program outputs. In this work, we treat the parameters of binding calls as inputs and the execution traces as outputs. For simplicity, execution trace (denoted as `ExeTrace`) is defined as a set of triples $\langle insn, opd, opd\_val\rangle$, recording the invoked instructions along with operand values. If an instruction has multiple operands or is invoked multiple times in an execution, it will correlate to multiple items in `ExeTrace`.

Given a binding call and one of its parameters under consideration, we invoke the binding call two times, each time assigning a different value to the parameter. Let $ExeTrace_1$ and $ExeTrace_2$ be the traces of the two executions. The differential result $Diff$ is calculated according to Equation 1. It is a set of operand variations involving those instructions that are invoked in both executions but have different operand values as response to the different parameter values assigned.

$$\begin{aligned}
&Diff = \{\langle insn, opd, opd\_val_1, opd\_val_2\rangle \mid cond_a \wedge cond_b \wedge cond_c\}\\
&cond_a : \langle insn, opd, opd\_val_1\rangle \in ExeTrace_1\\
&cond_b : \langle insn, opd, opd\_val_2\rangle \in ExeTrace_2\\
&cond_c : opd\_val_1 \neq opd\_val_2
\end{aligned} \tag{1}$$

We perform a two-stage process to identify type indicators. In the first stage, we leverage the type-consistency of accessors to select type indicator candidates. In the second stage, we resort to statistics to determine the most common type indicator for each data type from the candidates. The approach relies on two assumptions: (1) the accessors cover all primitive types and (2) the vast majority of binding calls share the same utility functions to translate data. Section VI discusses the generality of these assumptions.

#	Type	Differential Results	Occurrence
1	String	<0x2383ce39,ecx,0x3,0x5>	105
2	String	<0x2383da69,eax,0x6,0xa>	14
3	~~String~~	~~<0x2386788c,[ebp-0x14],0x10000,0x1000>~~	~~54~~
4	~~String~~	~~<0x238652cd,edx,0x10,0x20>~~	~~25~~
…		…	

ⓐ: this.zoomType as String √

Type	Differential Results
String	<0x2383ce39,ecx,0x3,0x5>
String	<0x2383da69,eax,0x6,0xa>
~~String~~	~~<0x2386788c,[ebp-0x14],0x10000,0x1000>~~
~~String~~	~~<0x238069e3,ecx,0x6,0x8>~~
	…

ⓑ: this.layout as String √

Type	Differential Results
Number	<0x2383da0d,[eax],0x17,0xb3>
Number	<0x2380315b,ecx,0x2,0x3>
Number	<0x2386788c,[ebp-0x14],0x10000,0x1000>
Number	<0x238639f7,eax,0x8,0x9>
	…

ⓒ: this.zoomType as Number ×

Fig. 5: Example of type indicator identification.

Algorithm 1: Select type indicator candidates.

```
Input  : a list of accessors
Output : type indicator candidates
1  $Candidates \leftarrow \{\}$
2  for $accessor$ in $accessor\_list$ do
3      $(getter, setter) \leftarrow ObtainGetterAndSetter(accessor)$
4      $correct\_type = typeof(getter())$
5      for $par\_type$ in $primitive\_types$ do
6          $\langle par\_type, par\_val_1, par\_val_2\rangle \leftarrow GetParVary(par\_type)$
7          $ExeTrace_1 \leftarrow Track(setter(par\_val_1)))$
8          $ExeTrace_2 \leftarrow Track(setter(par\_val_2)))$
9          $Diff \leftarrow GetDiff(ExeTrace_1, ExeTrace_2)$
10         if $par\_type == correct\_type$ then
11             if $Candidates[par\_type] == NULL$ then
12                 $Candidates[par\_type] \leftarrow Diff$
13             else
14                 $Candidates[par\_type] \leftarrow Candidates[par\_type] \cap Diff$
15         else
16             $Candidates[par\_type] \leftarrow Candidates[par\_type] \setminus Diff$
```

Algorithm 2: Determine the most common type indicator.

```
Input  : a list of binding calls, type indicator candidates
Output : type indicator for each data type
1  $TypeIndicator \leftarrow \{\}$
2  for $binding\_call$ in $binding\_call\_list$ do
3      for $par\_type$ in $primitive\_types$ do
4          $\langle par\_type, par\_val_1, par\_val_2\rangle \leftarrow GetParVary(par\_type)$
5          $ExeTrace_1 \leftarrow Track(binding\_call(par\_val_1)))$
6          $ExeTrace_2 \leftarrow Track(binding\_call(par\_val_2)))$
7          $Diff \leftarrow GetDiff(ExecuteTrace_1, ExecuteTrace_2)$
8          for $item$ in $(Diff \cap Candidates[par\_type])$ do
9              $occurrence[par\_type][item] + +$
10 for $par\_type$ in $primitive\_types$ do
11     $ParVary \leftarrow GetParVary(par\_type)$
12     $OpdVary \leftarrow \underset{item}{argmax}(occurrence[par\_type][item])$
13     $TypeIndicator[par\_type] = (ParVary \rightarrow OpdVary)$
```

Stage 1: Select Type Indicator Candidates. Algorithm 1 describes the selection process. For each accessor, we first obtain its `getter` and `setter` functions (Line 3) and acquire the correct type of the `getter` function's return value via the `typeof` operator (Line 4). Then we perform differential analysis for each primitive type (Lines 5-16). In particular, we apply the pre-defined parameter variation policy for the specified type (Line 6) [2], invoking the `setter` function with two different values and computing the difference of the two execution traces (Lines 7-9). The candidates should include the differential results of type-correct executions (Lines 11-14) and exclude those of type-incorrect executions (Line 16).

Stage 2: Determine the Most Common Type Indicator. The selected candidates from Stage 1 are the over-approximation of the type indicators. We would like to determine the most common one for each primitive type, such that it works for the majority of binding calls. As shown in Algorithm 2, we iterate the binding call list and attempt to match the parameter with various primitive types (Lines 2-9). For each combination of binding call and parameter type, we perform the differential analysis (Lines 4-7) and count the frequency of occurrence of the candidates (Lines 8-9). Finally, we choose the most common type indicator for each primitive type based on the frequency of occurrence (Lines 10-13). If a type has multiple indicator candidates sharing the same most frequent occurrence, we choose an arbitrary one as the final indicator.

[2]Currently, we apply simple yet effective policies: change the value from *true* to *false* for `Boolean`, change the value from a random number to another random number for `Number`, change the value from a random string to another random string with different lengths for `String`.

Example. Figure 5 illustrates how the type indicator of `String` is identified. In the first stage, we apply each primitive type on each accessor. The left and middle parts of Figure 5 show the type-correct differential results that apply `String` on `this.zoomType` and `this.layout`. The right part shows the type-incorrect differential result that applies `Number` on `this.zoomType`. For brevity, we only list four items in the figure for each differential result. According to Algorithm 1, Items #1 and #2 of the differential result ⓐ are treated as the type indicator candidates for `String`. Item #3 is excluded since it is contained in a type-incorrect differential result ⓒ. Item #4 is also excluded since it is not contained in all type-correct differential results ⓐ and ⓑ. In the second stage, Algorithm 2 determines Item #1 as the most common type indicator for `String`, since it is the most frequently occurring item (105 occurrences). The type indicator $\langle$`String`, "AAA", "*bbbbb*"$\rangle \rightarrow \langle 0x2383ce39, ecx, 0x3, 0x5\rangle$ means if the value of a `String` parameter is changed from "AAA" to "*bbbbb*", the ecx operand of the instruction at address $0x2383ce39$ will be changed from $0x3$ to $0x5$.

C. Inferring Parameter Types

Given a binding call, we iterate over its parameters. For each parameter, we first try to check it with primitive types, then try to check it with composite types.

Primitive Types. With the help of type indicator, we can easily judge whether a binding call parameter is of a certain primitive type. Basically, we apply the parameter variation (i.e., change the parameter values in two executions) specified by the type indicator and monitor whether the expected operand variation (i.e., change of operand values in two executions) occurs in the differential results. Take the `String` indicator ($\langle$`String`, "AAA", "*bbbbb*"$\rangle \rightarrow$

$\langle 0x2383ce39, ecx, 0x3, 0x5 \rangle$) as an example. To judge whether a binding call parameter is of the `String` type, we execute the binding call twice with the parameter value set to be "AAA" and "*bbbbb*" respectively and monitor at the break point set on the address $0x2383ce39$ to see whether the value of the ecx operand is $0x3$ and $0x5$ accordingly.

Composite Types. Unlike primitive types, we do not have direct type indicator for composite types. Fortunately, the embedded scripting engine relies on the underlying primitive type handlers to resolve composite types. We can use type indicator of primitive types to recognize composite types. Note that for a composite value, we should not only identify that it is an array or an object, but also identify its element types (for array) or property names and types (for object).

Array. An array can hold many different types of values, and one can access the values by referring to an index number. To judge whether a binding call parameter is an array, we construct an array object for the parameter and try to check its first element against each primitive type. If we find the operand variation (as the response to the parameter variation) matches the type indicator of a certain primitive type, we know that the parameter is an array and its first element is of the corresponding primitive type. We repeat type checking for other elements of the array until there is no type indicator found or the attempt limit is reached.

Object. An object contains `key-value` pairs, and one can access the values by referring to the keys. The keys can not be obtained by blind guessing. Observe that the binding calls generally get the desired key of an object from the data segment of the program. Hence the strings read from the data segment during the binding call execution are considered as possible keys. For each of these keys, we try to identify the type of the corresponding value. Similarly as done for array, we construct an object, fill it with the specified key and try different values of different types to compute the differential results. If the parameter-operand variation pattern of a certain type indicator appears in the differential results, we can conclude the object property with the given key is of the corresponding type.

Multiple Parameters. Binding calls usually have more than one parameters. By default, the type reasoning is conducted on the parameters one by one according to their occurrence order. Although a binding call may process the parameters in a unique order, it is not the common situation. There is a special case where certain binding calls have a requirement for the number of the provided parameters. If the requirement is not satisfied, these binding calls do not process any parameter. We handle such a situation by increasing the number of parameters one by one until the type of the first parameter gets identified or the attempt limit is reached. If the type of the first argument can be identified finally, the minimum number of parameters required by the binding call is also obtained.

D. Fuzzing

We develop a simple yet effective generation-based fuzzer that leverages the inferred type information as guidance. The fuzzer repeats the following steps. (1) Generates script code containing binding calls. (2) Injects the generated script into base PDF files. (3) Executes the target PDF reader with the injected PDF files. (4) Monitors the execution state for crash detection. (5) Once a vulnerability is discovered, minimizes the triggering test case. The injection, execution and monitoring are standard. We focus on the generation and minimization.

Generation. The generation process produces a bunch of binding calls. For each invoked binding call, the fuzzer feeds its parameters with values of the inferred types. Such values can be randomly generated or dictionary items extracted from the target program. Given the considerable number of binding calls, it is impossible to test all of their combinations. We notice that testing binding calls that are semantically related together may have better performance than the blind mixture. Since the name of a binding call reflects its semantics, we treat binding calls whose names share non-trivial subwords as being semantically related. For two semantically related binding calls, the fuzzer will try with probability to assign the same value to their parameters and/or return values that have the same type. For example, `getNthFieldName` and `getField` are semantically related, since they share the subword "`Field`". The fuzzer may try to use the return value of `getNthFieldName` as the value of the first parameter in `getField`, since both of them are of the `String` type.

Minimization. The purpose of minimization is to reduce the complexity of triggering code. It is achieved by removing the invocations of binding calls from the test case as much as possible while keeping the vulnerability triggerable. We adopt two modes of minimization: eager and incremental. The eager mode is used by default. It repeatedly reduces by half the number of binding call invocations in the test case until the vulnerability could not be triggered. The incremental mode is activated when the eager mode no longer works. It randomly selects one binding call invocation to be removed along with its semantically related counterparts. By removing the invocations of semantically related binding calls as a whole, we can decrease the possibility of invalid reference to the removed items (i.e., reference errors). For example, if the invocation of `getNthFieldName` is removed from the test case, the invocation of `getField` will be removed either.

E. Implementation

We implement TYPEORACLE with 9.6K lines of code in Python, in which 7.6K lines for type reasoning and 2K lines for fuzzing. The type reasoning component uses Pin [18] to collect execution traces for differential analysis. The fuzzing component uses Pywinauto [19] to automate the whole testing process, WerFault [20] to capture application crash, and DynamoRIO [21] to collect instruction coverage.

V. EVALUATION

We investigate the following research questions in order to evaluate the effectiveness of TYPEORACLE:

RQ1: How accurate is the identified type information? (Section V-B)

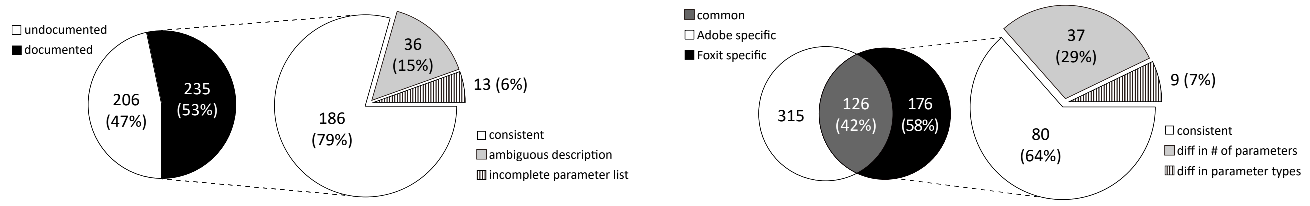
TABLE I: Accuracy of type reasoning.

	Inferred by TYPEORACLE						Inferred by Error Message						Inferred by Path Length					
	Boolean	Number	String	Array	Object	Total	Boolean	Number	String	Array	Object	Total	Boolean	Number	String	Array	Object	Total
# of correctly reported	112	80	329	16	6	543	3	19	6	2	0	30	8	35	21	4	0	68
# of reported cases	112	80	330	16	6	544	6	19	6	2	0	33	27	189	34	258	31	539
# of actual cases	115	81	337	19	6	558	115	81	337	19	6	558	115	81	337	19	6	558
precision	100.0%	100.0%	99.7%	100.0%	100.0%	99.8%	50.0%	100.0%	100.0%	100.0%	0.0%	90.9%	29.6%	18.5%	61.8%	1.6%	0.0%	12.6%
recall	97.4%	98.8%	97.6%	84.2%	100.0%	97.3%	2.6%	23.5%	1.8%	10.5%	0.0%	5.4%	7.0%	43.2%	6.2%	21.1%	0.0%	12.2%

(a) on Adobe Reader

	Inferred by TYPEORACLE						Inferred by Error Message						Inferred by Path Length					
	Boolean	Number	String	Array	Object	Total	Boolean	Number	String	Array	Object	Total	Boolean	Number	String	Array	Object	Total
# of correctly reported	77	53	140	17	3	290	6	26	48	6	1	87	12	31	61	12	2	118
# of reported cases	77	53	140	17	3	290	6	26	48	6	1	87	21	81	67	104	31	304
# of actual cases	78	60	144	18	4	304	78	60	144	18	4	304	78	60	144	18	4	304
precision	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	57.1%	38.3%	91.0%	11.5%	6.5%	38.8%
recall	98.7%	88.3%	97.2%	94.4%	75.0%	95.4%	7.7%	43.3%	33.3%	33.3%	25.0%	28.6%	15.4%	51.7%	42.4%	66.7%	50.0%	38.8%

(b) on Foxit Reader



(a) between documentation and implementation (b) between Adobe Reader and Foxit Reader

Fig. 6: Systematic study of binding call inconsistency.

RQ2: How inconsistent is between documentation and implementation and between different implementations? (Section V-C)

RQ3: How effective is the identified type information in improving fuzzing performance? (Section V-D)

RQ4: How good is the ability of TYPEORACLE for discovering zero-day vulnerabilities? (Section V-E)

A. Experimental Setup

Target Programs. We choose two widely used PDF readers (Adobe Reader v2021.011.20039 and Foxit Reader v11.2.1.53537) as target programs.

Computing Resources. The target programs are run in virtual machines (VMs), using VMware as the hypervisor. The host machine has 8-core CPU (Intel i7-6700 @ 4.00GHz) and 32GB memory. The guest VM has one CPU core and 4GB memory, and uses Windows 8.1 as the operating system.

B. Accuracy of Type Reasoning

Since there is no complete ground truth for the parameter type of binding calls in different PDF readers, we resort to the API reference manual provided by Adobe [9] and reverse engineering to measure the accuracy of type reasoning. The inferred type is considered to be correct if it is consistent with that obtained from the documentation. Otherwise, we manually identify the parameter type by inspecting which type-specific utility function is invoked by the binding call to translate data representation of the parameter.

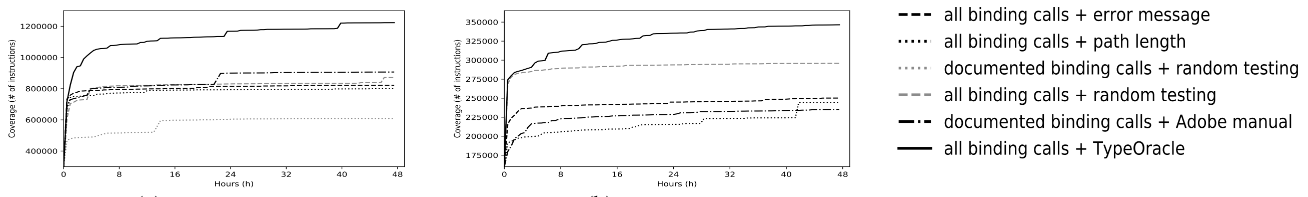
For Adobe Reader, we analyze all documented binding calls and randomly selected 10% of undocumented ones. For Foxit Reader, we analyze all common binding calls that are also implemented in Adobe Reader and randomly selected 10% of those specific to Foxit Reader. In total, we have 251 binding calls analyzed for Adobe Reader and 193 for Foxit Reader. In this dataset, the number of parameters ranges from 1 to 40 (1.9 per binding call on average), and the number of parameter types ranges from 1 to 5 (1.3 per binding call on average).

Table I shows the analysis result. For each data type, we record the number of parameters that actually have this type (# of actual cases), that reports as having this type (# of reported cases), and that are correctly reported (# of correctly reported). As we can see, TYPEORACLE has almost 100% precision and around 96% recall. As a comparison, type reasoning by using error message provides moderate precision but poor recall (<30%), while using path length provides both low precision and recall (<40%). This demonstrates the need for sophisticated feature (rather than the shallow ones) in type reasoning.

C. Inconsistency of Binding Calls

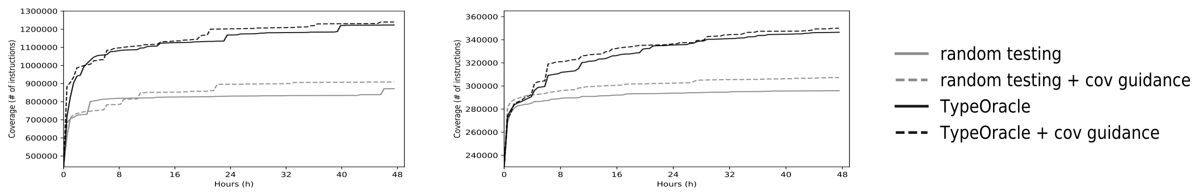
To further demonstrate the advantage of TYPEORACLE over the API documentation, we systematically study the binding call inconsistency between documentation and implementation, and between Adobe Reader and Foxit Reader.

We can see from Figure 6a that only 53% of Adobe Reader's binding calls are documented, 21% of which have type inconsistency with the documentation. Further inspection reveals that the inconsistency comes from ambiguous description and incomplete parameter list in the documentation. Figure 6b shows that Foxit Reader shares 42% of binding calls with Adobe Reader, 36% of which differ in either the number of parameters or the type of certain parameters.



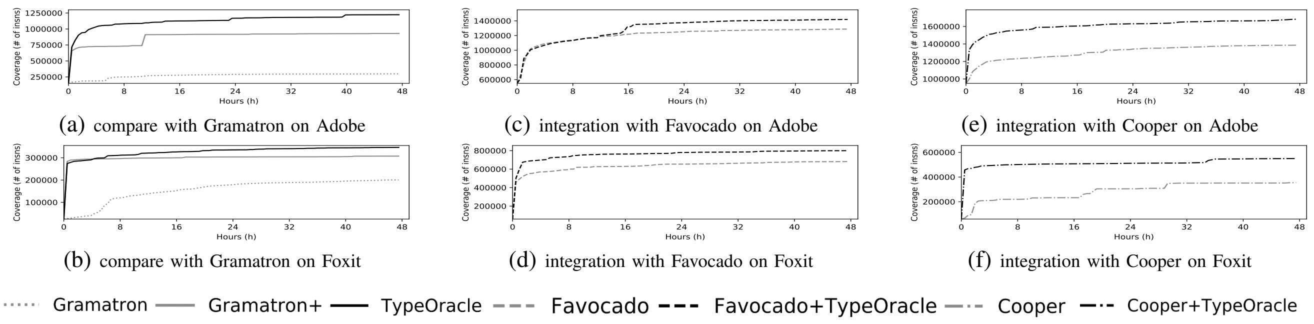
(a) Adobe Reader (b) Foxit Reader

Fig. 7: Fuzzing performance of type guidance.



(a) Adobe Reader (b) Foxit Reader

Fig. 8: Fuzzing performance of coverage guidance.



(a) compare with Gramatron on Adobe (c) integration with Favocado on Adobe (e) integration with Cooper on Adobe

(b) compare with Gramatron on Foxit (d) integration with Favocado on Foxit (f) integration with Cooper on Foxit

Fig. 9: Comparison and integration with state-of-art fuzzers.

This indicates that Adobe's API documentation is not suitable for obtaining the parameter type of Foxit Reader's binding calls, neither is it fully dependable for Adobe Reader. TYPEORACLE performs better than the API documentation in type identification. Among the vulnerabilities discovered by TYPEORACLE, 19 involve undocumented binding calls and 2 involve inconsistent ones.

D. Fuzzing Performance

We conduct three groups of experiments to evaluate the fuzzing performance of different configurations on both Adobe Reader and Foxit Reader. Each experiment is run for five times, and each time lasts for 48 hours. Figure 7, Figure 8 and Figure 9 present the experiment results, in which X-axis represents time, Y-axis represents the number of covered instructions, the lines represent mean of the five runs. Note that we do not count the instructions covered by base PDF files themselves, and only concentrate on the instructions covered by binding calls.

Group 1: Type Guidance. This group of experiments is to assess the effect of using type information as fuzzing guidance. Figure 7a presents the experiment results on Adobe Reader, from which we have two conclusions. First, undocumented binding calls incur 43.09% coverage increment (comparing the gray dashed line with the gray dotted line). Second, type information provided by TYPEORACLE incurs 40.38% coverage increment than random testing (comparing the black solid line with the gray dashed line) and 34.88% coverage increment than that provided by Adobe's API documentation (comparing the black solid line with the black dash-dot line). The experiment results on Foxit Reader are shown in Figure 7b. As we can see, using the type information extracted by TYPEORACLE incurs 17.09% and 47.28% coverage increment compared with random testing and using Adobe's API documentation respectively. From Figures 7a and 7b, we can also see that using type information inferred by error message or path length yields even lower coverage than random testing.

Group 2: Coverage Guidance. We also assess the effect of using coverage feedback as fuzzing guidance. The fuzzing process is refracted to leverage the coverage feedback. Specifically, if a test case covers new paths, it will be preserved for further mutations in the subsequent iterations. In this group of experiments, we use the same base PDF file as in Group 1. The experiment results indicate that coverage guidance has limited impact on improving fuzzing performance. On Adobe Reader, it increases the coverage by 4.27% for random testing and by 1.36% for fuzzing with TYPEORACLE (Figure 8a). On Foxit Reader, it increases the coverage by 3.86% for random testing and by 1.00% for fuzzing with TYPEORACLE (Figure 8b). This indicates that type guidance is better than coverage guidance for fuzzing binding calls in PDF readers. The possible reason is that the versatile fault tolerance reduces the sensitivity of code coverage towards the parameter values of different types.

TABLE II: Discovered Zero-Day Vulnerabilities.

Product	Type	Severity	Discovered By					
			G$^+$	F	C	T	F+T	C+T
Adobe Reader	Buffer Error *	High				√⃝	√⃝	
	Buffer Error *	High				√⃝		
	Buffer Error *	High				√⃝		
	Buffer Error *	High				√		
	Buffer Error *	High			√	√⃝		
	Buffer Overflow *	High				√⃝		
	Buffer Overflow *	High				√⃝		√
	Buffer Overflow	High		√⃝			√⃝	
	Buffer Overflow	High						√⃝
	Memory Corruption *	High				√		
	Memory Corruption *	High				√		
	Out-of-bounds Write *	High				√		
	Untrusted Pointer Dereference *	High	√	√	√	√⃝	√	√⃝
	Untrusted Pointer Dereference *	High				√		
	Use After Free *	High				√⃝		
	Invalid Memory Access *	Moderate				√⃝		√⃝
	Invalid Memory Access *	Moderate				√⃝		√⃝
	Invalid Memory Access *	Moderate				√⃝		√⃝
	Memory Corruption *	Moderate				√		
	Out-of-bounds Read *	Moderate				√		
	Out-of-bounds Read *	Moderate				√		
	Out-of-bounds Read *	Moderate				√		
	Null Pointer Dereference *	Low				√⃝	√	
	Null Pointer Dereference *	Low				√⃝		√
	Stack Exhaustion *	Low		√		√		
	Stack Exhaustion	Low				√⃝		
	Stack Exhaustion	Low				√		
Foxit Reader	Buffer Overflow	High				√⃝		
	Buffer Overflow	High						√⃝
	Invalid Memory Access	High				√		
	Invalid Memory Access	High						√⃝
	Out-of-bounds Read	Moderate				√⃝		
	Out-of-bounds Read	Moderate				√⃝		
	Out-of-bounds Read *	Moderate					√⃝	
	Null Pointer Dereference	Low				√⃝		
	Null Pointer Dereference *	Low				√⃝		
	Null Pointer Dereference *	Low				√		
	Stack Exhaustion	Low	√			√		

G$^+$: Gramatron$^+$, F: Favocado, C: Cooper, T: TYPEORACLE
F+T: Favocado+TYPEORACLE, C+T: Cooper+TYPEORACLE
√: discovered within 2 weeks, √⃝: discovered within the first 48 hours
* in the second column indicates the vulnerability got CVE assigned

Group 3: Comparison and Integration with State-of-Art Fuzzers. Gramatron [22] is a state-of-art grammar-aware fuzzer that uses grammar automatons and aggressive mutation operators to synthesize complex syntactically-valid test cases. We extend Gramatron (denoted as Gramatron$^+$) to support binding call invocations. We use Gramatron and Gramatron$^+$ for the generation of scripts injected into the base PDF file and treat them as standalone baselines. By comparison, TYPEORACLE achieves 309.18% and 72.56% more coverage than Gramatron, and 31.56% and 12.65% more coverage than Gramatron$^+$ (Figures 9a and 9b). This indicates that binding call invocations (especially with type-correct parameter values) plays more important role than syntax complexity for fuzzing embedded scripting engines.

Favocado [23] and Cooper [24] are two state-of-art fuzzers that consider binding calls during fuzzing. Favocado mixes binding calls with various Javascript templates. Cooper simultaneously mutates both binding calls and PDF page elements. They extract parameter types from Adobe's API documentation, hence cannot handle undocumented and inconsistent binding calls. TYPEORACLE can be complementary with them by providing more complete and accurate parameter type information. Compared with the original Favocado, the integration of Favocado and TYPEORACLE (denoted as Favocado+TYPEORACLE) increases coverage by 10.19% on Adobe Reader and by 17.30% on Foxit Reader (Figures 9c and 9d). Compared with the original Cooper, the integration of Cooper and TYPEORACLE (denoted as Cooper+TYPEORACLE) increases coverage by 21.37% on Adobe Reader and by 55.12% on Foxit Reader (Figures 9e and 9f). The increased coverage raises the chance of triggering deep vulnerabilities.

E. Vulnerability Discovery

To evaluate the vulnerability discovery capabilities of different fuzzers in the wild, we deploy 2-week fuzzing campaign for each fuzzer. Gramatron$^+$, Favocado, Cooper, TYPEORACLE, Favocado+TYPEORACLE and Cooper+TYPEORACLE report 2, 6, 5, 33, 10, 18 unique crashes respectively. Further inspection on the crash reports result in 38 zero-day vulnerabilities, as detailed in Table II. Even within the first 48 hours, TYPEORACLE exposed 18 vulnerabilities, which is 2.57 times that of the best of other fuzzers. We should note that Favocado+TYPEORACLE and Cooper+TYPEORACLE split time between the mutations of binding calls and the mutations of Javascript templates or PDF page elements, hence expose fewer vulnerabilities than vanilla TYPEORACLE within the same time budget.

Case 1: Vulnerability Involving Foxit-Specific Binding Call. It is a stack exhaustion vulnerability in Foxit Reader, which involves the binding call `App.sendEmail` that is specific to Foxit Reader. This binding call is not presented in Adobe's API documentation, neither is it implemented by Adobe Reader. Fortunately, TYPEORACLE can correctly identify its parameter is of the `String` type and guide the fuzzer to effectively generate crafted yet type-valid test cases. Listing 3 shows the minimized triggering code. As we can see, if this binding call is fed with a string containing a considerable number of open square brackets (i.e., "["), the vulnerability will be triggered.

Listing 3: Stack exhaustion vulnerability in Foxit Reader.

```
var str = Array(6000).join("[");
app.sendEmail(str);
```

Case 2: Vulnerability Discovered by Favocado + TYPEORACLE. By integrating Favocado with TYPEORACLE, we found an out-of-bounds read vulnerability in Foxit Reader. As shown in Listing 4, the key to triggering the vulnerability is the use of the `toString` template (provided by Favocado) and the type-correct parameter values fed to the `Util.scand` binding call (responsible by TYPEORACLE). Both Adobe Reader and Foxit Reader implement this binding call, however Favocado does not correctly extract the type information from the Adobe's API documentation. Through manual check, we found the possible reason could be the ambiguous description of the documentation. It uses "the rules" and "the date" to describe the semantics of the two parameters, which cannot be intuitively mapped to data types. On the contrary, TYPEORACLE can correctly identify the two parameters are of the `String` type with the help of type indicator. We should note that during fuzzing, the selection of Javascript templates and binding calls are completely automatic without any manual efforts.

Listing 4: Out-of-bounds read vulnerability in Foxit Reader.

```
var obj = {}
obj.toString = function() {
    try { this.color.convert(); return "a"; }
    catch (e) {}
}
util.scand("oyt", obj);
```

VI. Discussion

Generality. In addition to the PDF readers, we also investigate the generality of the research problem and our methodology on other host environments, including PDF-XChange [25], Node.js [26] and Chromium [27]. We found the prevalence of fault tolerance for type-incorrect binding call parameters (recall in Section III) and the wide satisfaction of the two assumptions on accessors and utility functions (recall in Section IV-B).

PDF-XChange is a famous full-featured PDF editor, which allows users to modify and transform PDF files. It implements binding calls for both reading and editing PDF files. By using the parameter type information extracted by TypeOracle, we found 6 vulnerabilities in the latest version of PDF-XChange, which have been confirmed and awarded \$250 bounty.

Node.js and Chromium are the desktop and browser environments that embed scripting engines. TypeOracle can successfully identify the parameter types of their binding calls. The inspection on randomly selected 10% of the inferred results shows the precision is 100% and the recall is around 98%. We leave the comprehensive research on testing binding calls in other host environments as future work.

Threats to Validity. There are two threats to validity in this work. First, our fuzzer currently relies on function names to deduce semantic relationship between binding calls. Although simple and effective, it is possible that two semantically-related binding calls do not share same subwords in their names. Second, although type-correct invocations of binding calls help the execution reach deep code and trigger most vulnerabilities, it is possible that type-incorrect parameter values may also trigger certain vulnerabilities at special program states. To tackle this problem, we can adapt the fuzzer to provide type-incorrect parameter values with a small probability.

VII. Related Work

Type Recovery. The most relevant work is type recovery of program variables on binary executable or bytecode [28]. Traditional approaches infer types by examining the values stored in registers and memory (i.e., value-based [29]–[31]), by propagating types from instructions and function invocations whose operands/parameters types are known (i.e., flow-based [32]–[34]), or by combining both [35], [36]. The inference can be static [37]–[39], dynamic [40]–[42] or hybrid [43], [44]. The traditional approaches heavily rely domain-expert-provided rules, hence are brittle and require extensive efforts to maintain.

Recently, researchers propose data-driven approaches that leverage machine learning for type recovery [45]–[48]. The type-related instructions are used for learning. StateFormer [47] pretrains a model to statically approximate the operational semantics of assembly instructions and transfers the knowledge to learn type inference rules. SigRec [48] uses type-aware symbolic execution to explore the instructions of self-generated smart contract functions and learns the access patterns of each type to summarize rules for type recovery of parameters.

Conceptually, TypeOracle is a light-weight learning-based dynamic approach to type recovery. Different from the existing studies, TypeOracle tries to infer the type of binding call parameters that are wrapped and passed through different language layers. Due to the semantic gap, it is very difficult to manually summarize type inference rules for script values from native instructions. In addition, we do not have sufficient data for pretraining as required by StateFormer, neither can we generate binding calls on-demand as done by SigRec. Indeed, TypeOracle proposes a new approach to learning type inference rules from differential analysis.

Fuzzing. Fuzzing is a promising technique for vulnerability discovery. There exist a large number of studies exploring different ways to improve the effectiveness of fuzzing [49]–[56]. File format fuzzers solve complex path conditions by symbolic execution [57]–[62] or gradient-based search [63]–[68]. Language fuzzers usually enhance the diversity of the syntax structure and semantic operations with the help of grammar specifications [69], [70], historical proof-of-concept exploits [71], [72], or grammar-aware mutations on the generated test cases [22], [73]–[76].

API fuzzers manage to extract the interface specifications, mine the interface association rules and select the appropriate content as feedback. The interface specifications are usually obtained from source code [77]–[79], documentation [80], [81], test suites [82]–[84] or easy-to-obtain features of runtime executions [14]. In the scenario of fuzzing binding calls in commercial-of-the-shelf PDF readers, no source code is available, neither documentation, test suites and shallow execution features are adequate for type reasoning of parameter types.

Favocado [23] and Cooper [24] are state-of-art approaches that use fuzzing to detect flaws in binding code. They heavily rely on Adobe's API documentation to extract parameter type, hence cannot handle undocumented or inconsistent binding calls. TypeOracle can be complementary with them.

VIII. Conclusion

This paper proposes TypeOracle, an approach to infer the parameter types of binding calls for improving fuzzing of the embedded scripting engine in PDF readers. TypeOracle first performs operand-variation-oriented differential analysis to extract type indicators that can be used as oracles for type reasoning. Then, TypeOracle generates and executes test cases based on the inferred parameter types. Through the evaluation on two popular PDF readers (Adobe Reader and Foxit Reader), we demonstrated the accuracy of our type reasoning method and the effectiveness of the inferred type for improving fuzzing in both code coverage and vulnerability discovery. TypeOracle found 38 zero-day vulnerabilities, involving undocumented or inconsistent binding calls whose correct parameter types are previously unknown.

References

[1] Wikipedia, "Pdf," https://en.wikipedia.org/wiki/PDF, 2021.

[2] C. P. S. T. LTD, "The 2020 cyber security report," https://www.ntsc.org/assets/pdfs/cyber-security-report-2020.pdf, 2020.

[3] Vulners, "Vulnerability assessment platform," https://vulners.com/search?query=type:zdi\%20acrobat, 2022.

[4] F. Brown, S. Narayan, R. S. Wahby, D. R. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in javascript bindings," in *2017 IEEE Symposium on Security and Privacy, SP*. IEEE Computer Society, 2017, pp. 559–578.

[5] J. Park, J. Park, S. An, and S. Ryu, "JISET: javascript ir-based semantics extraction toolchain," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 2020, pp. 647–658.

[6] J. Park, S. An, D. Youn, G. Kim, and S. Ryu, "JEST: N +1-version differential testing of both javascript engines and specification," *CoRR*, vol. abs/2102.07498, 2021.

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

[8] F. M. Kifetew, R. Tiella, and P. Tonella, "Generating valid grammar-based test inputs by means of genetic programming and annotated grammars," *Empirical Software Engineering*, vol. 22, no. 2, pp. 928–961, 2017.

[9] Adobe, "Javascript for acrobat api reference," https://opensource.adobe.com/dc-acrobat-sdk-docs/acrobatsdk/pdfs/acrobatsdk_jsapiref.pdf, 2021.

[10] "veraPDF test suite," https://www.pdfa.org/resource/verapdf-test-suite/, 2021.

[11] "PDF/A test suite," https://github.com/bfosupport/pdfa-testsuite, 2021.

[12] Mozilla, "Pdf.js test suite," https://github.com/mozilla/pdf.js/tree/master/test/pdfs, 2021.

[13] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno, "Privacy oracle: a system for finding application leaks with black box differential testing," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS*, 2008, pp. 279–288.

[14] Y. Aafer, W. You, Y. Sun, Y. Shi, X. Zhang, and H. Yin, "Android smarttvs vulnerability discovery via log-guided fuzzing," in *30th USENIX Security Symposium (USENIX Security)*. USENIX Association, Aug. 2021.

[15] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *2019 IEEE Symposium on Security and Privacy, SP*. IEEE, 2019, pp. 769–786.

[16] Adobe, "Adobe reader," https://acrobat.adobe.com/us/en/acrobat/pdf-reader.html, 2022.

[17] Foxit, "Foxit reader," https://www.foxitsoftware.com/pdf-reader/, 2022.

[18] O. Levi, "Pin - a dynamic binary instrumentation tool," https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html, 2018.

[19] M. M. Mahon, "Pywinauto," https://github.com/pywinauto/pywinauto, 2022.

[20] WikiPedia, "Windows error reporting," https://en.wikipedia.org/wiki/Windows_Error_Reporting, 2022.

[21] DynamoRIO, "Dynamorio," https://dynamorio.org/, 2022.

[22] P. Srivastava and M. Payer, "Gramatron: effective grammar-aware fuzzing," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2021, pp. 244–256.

[23] 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," 2021.

[24] P. Xu, Y. Wang, H. Hu, and P. Su, "Cooper: Testing the binding code of scripting languages with cooperative mutation," in *Network and Distributed Systems Security NDSS Symposium 2022*. NDSS, 2022.

[25] "Pdf-xchange," https://pdf-xchange.eu/pdf-xchange-editor/, 2022.

[26] "Node.js," https://nodejs.org/en/, 2022.

[27] "Chromium," https://www.chromium.org/chromium-projects/, 2022.

[28] J. Caballero and Z. Lin, "Type inference on executables," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 65:1–65:35, 2016.

[29] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. USENIX Association, 2008, pp. 255–266.

[30] I. Haller, A. Slowinska, and H. Bos, "Mempick: High-level data structure detection in C/C++ binaries," in *20th Working Conference on Reverse Engineering, WCRE*. IEEE Computer Society, 2013, pp. 32–41.

[31] V. Srinivasan and T. W. Reps, "Recovery of class hierarchies and composition relationships from machine code," in *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*, ser. Lecture Notes in Computer Science, vol. 8409. Springer, 2014, pp. 61–84.

[32] G. Ramalingam, J. Field, and F. Tip, "Aggregate structure identification and its application to program analysis," in *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1999, pp. 119–132.

[33] I. Guilfanov, "A simple type system for program reengineering," in *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01*. IEEE Computer Society, 2001, pp. 357–361.

[34] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irún-Briz, "Tupni: automatic reverse engineering of input formats," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS*. ACM, 2008, pp. 391–402.

[35] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2010.

[36] A. Prakash, X. Hu, and H. Yin, "vfguard: Strict protection for virtual function calls in COTS C++ binaries," in *22nd Annual Network and Distributed System Security Symposium, NDSS*, 2015.

[37] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "Vtint: Protecting virtual function tables' integrity," in *22nd Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2015.

[38] K. Elwazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*. ACM, 2013, pp. 51–60.

[39] D. Dewey and J. T. Giffin, "Static detection of C++ vtable escape vulnerabilities in binary code," in *19th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2012.

[40] J. Caballero, H. Yin, Z. Liang, and D. X. Song, "Polyglot: automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS*. ACM, 2007, pp. 317–329.

[41] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2010.

[42] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Proceedings of the Network and Distributed System Security Symposium, NDSS, publisher = The Internet Society, year = 2011.*

[43] J. Lee, T. Avgerinos, and D. Brumley, "TIE: principled reverse engineering of types in binary programs," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, 6th February - 9th February 2011*. The Internet Society, 2011.

[44] G. Wondracek, P. M. Comparetti, C. Krügel, and E. Kirda, "Automatic network protocol analysis," in *Proceedings of the Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2008.

[45] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, "Typeminer: Recovering types in binary programs using machine learning," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*, ser. Lecture Notes in Computer Science, vol. 11543. Springer, 2019, pp. 288–308.

[46] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. T. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. ACM, 2018, pp. 1667–1680.

[47] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana, "Stateformer: fine-grained type recovery from binaries using generative state modeling," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 2021, pp. 690–702.

[48] T. Chen, Z. Li, X. Luo, X. Wang, T. Wang, Z. He, K. Fang, Y. Zhang, H. Zhu, H. Li, Y. Cheng, and X. Zhang, "Sigrec: Automatic recovery of function signatures in smart contracts," *IEEE Trans. Software Eng.*, vol. 48, no. 8, pp. 3066–3086, 2022.

[49] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. USENIX Association, 2008, pp. 209–224.

[50] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: differential fuzzing for side-channel analysis," in *Proceedings of the 41st International Conference on Software Engineering, ICSE*. IEEE / ACM, 2019, pp. 176–187.

[51] R. Kersten, K. S. Luckow, and C. S. Pasareanu, "POSTER: afl-based fuzzing for java with kelinci," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*. ACM, 2017, pp. 2511–2513.

[52] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "Just fuzz it: Solving floating-point constraints using coverage-guided fuzzing," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, 2019, pp. 521–532.

[53] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu, "Apicraft: Fuzz driver generation for closed-source SDK libraries," in *30th USENIX Security Symposium, USENIX Security*. USENIX Association, 2021, pp. 2811–2828.

[54] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *ICSE '20: 42nd International Conference on Software Engineering*. ACM, 2020, pp. 999–1010.

[55] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*. ACM, 2017, pp. 2329–2344.

[56] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *Proceedings of the 31st USENIX Security Symposium*, ser. USENIX SEC, 2022.

[57] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *23rd Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2016.

[58] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium, USENIX Security*. USENIX Association, 2018, pp. 745–761.

[59] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. ACM, 2019, pp. 515–530.

[60] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: fuzzing with input-to-state correspondence," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019*, 2019.

[61] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation with java pathfinder," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*. ACM, 2004, pp. 97–107.

[62] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011*. ACM, 2011, pp. 1066–1071.

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

[64] W. You, X. Liu, S. Ma, D. M. Perry, X. Zhang, and B. Liang, "SLF: fuzzing without valid seed inputs," in *Proceedings of the 41st International Conference on Software Engineering, ICSE*. IEEE / ACM, 2019, pp. 712–723.

[65] X. Liu, W. You, Z. Zhang, and X. Zhang, "Tensilefuzz: facilitating seed input generation in fuzzing via string constraint solving," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2022, pp. 391–403.

[66] P. Chen, J. Liu, and H. Chen, "Matryoshka: Fuzzing deeply nested branches," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. ACM, 2019, pp. 499–513.

[67] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy, SP*. IEEE, 2019, pp. 803–817.

[68] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proceedings of the 41st International Conference on Software Engineering, ICSE*. IEEE / ACM, 2019, pp. 736–747.

[69] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi, and D. Teuchert, "NAUTILUS: fishing for deep bugs with grammars," in *26th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2019.

[70] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "SQUIRREL: testing database management systems with language validity and coverage feedback," in *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2020, pp. 955–970.

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

[72] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided javascript engine fuzzer," in *29th USENIX Security Symposium, USENIX Security*. USENIX Association, 2020, pp. 2613–2630.

[73] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: grammar-aware greybox fuzzing," in *Proceedings of the 41st International Conference on Software Engineering, ICSE*. IEEE / ACM, 2019, pp. 724–735.

[74] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, vol. 9878. Springer, 2016, pp. 581–601.

[75] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21th USENIX Security Symposium*. USENIX Association, 2012, pp. 445–458.

[76] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines," in *26th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2019.

[77] Google, "syzkaller - kernel fuzzer," https://github.com/google/syzkaller, 2021.

[78] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *29th USENIX Security Symposium, USENIX Security*. USENIX Association, 2020, pp. 2271–2287.

[79] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "FANS: fuzzing android native system services via automated interface analysis," in *29th USENIX Security Symposium, USENIX Security*. USENIX Association, 2020, pp. 307–323.

[80] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: stateful REST API fuzzing," in *Proceedings of the 41st International Conference on Software Engineering, ICSE*. IEEE / ACM, 2019, pp. 748–758.

[81] P. Godefroid, B. Huang, and M. Polishchuk, "Intelligent REST API data fuzzing," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2020, pp. 725–736.

[82] H. Han and S. K. Cha, "IMF: inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*. ACM, 2017, pp. 2345–2358.

[83] K. Yang, H. Zhao, C. Zhang, J. Zhuge, and H. Duan, "Fuzzing IPC with knowledge inference," in *38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019*. IEEE, 2019, pp. 11–20.

[84] E. Arteca, M. Schaefer, and F. Tip, "Learning how to listen: Automatically finding bug patterns in event-driven javascript apis," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.