From Documentation to Zero-day Vulnerabilities: LLM-Driven Fuzzing of JavaScript Engines in PDF Readers

Suyue Guo
University of California, Santa Barbara
Santa Barbara, California, USA
sguo568@ucsb.edu

Stijn Pletinckx
University of California, Santa Barbara
Santa Barbara, California, USA
stijn@ucsb.edu

Tianle Yu
University of California, Santa Barbara
Santa Barbara, California, USA
tianleyu@stanford.edu

Yigitcan Kaya
University of California, Santa Barbara
Santa Barbara, California, USA
yigitcan@ucsb.edu

Saad Ullah
Boston University
Boston, Massachusetts, USA
saadu@bu.edu

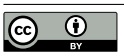
Wenbo Guo
University of California, Santa Barbara
Santa Barbara, California, USA
henrygwb@ucsb.edu

Christopher Kruegel
University of California, Santa Barbara
Santa Barbara, California, USA
chris@cs.ucsb.edu

Giovanni Vigna
University of California, Santa Barbara
Santa Barbara, California, USA
vigna@ucsb.edu

Abstract

Existing fuzzers for PDF readers rely on simple test cases that involve only *individual* API calls, leading to limited coverage and potentially missing vulnerabilities that require *sequences* of API calls. To address these limitations, we propose PDFuzzer, a novel PDF engine fuzzer that automatically generates complex and meaningful API call sequences. PDFuzzer first uses a Large Language Model (LLM) to construct context-free grammars and infer the relationships between individual API calls from specifications extracted from JavaScript API manuals and execution traces. Based on the grammars and relationships, PDFuzzer employs a constraint solver to generate concrete API call sequences for fuzzing. Our experiments show that PDFuzzer significantly outperforms state-of-the-art PDF fuzzers (TypeOracle, Favocado, and Cooper) and LLM-based fuzzers (Fuzz4All, naive LLM) on three mainstream PDF readers: Adobe Acrobat Reader, Foxit PDF Reader, and PDF-XChange Editor. PDFuzzer achieves up to 48% higher coverage than existing tools and identifies 31 zero-day vulnerabilities in these readers, from information leakage to arbitrary code execution. Our ablation study validates the necessity of each component, including LLMs, which achieve high accuracy across all pipeline stages (93-98%). We disclosed all vulnerabilities to the vendors via a coordinated vulnerability disclosure process and received bug bounties.



CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832609

CCS Concepts

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**.

Keywords

API Security, API Relation, Fuzzing, LLM

ACM Reference Format:
Suyue Guo, Stijn Pletinckx, Tianle Yu, Yigitcan Kaya, Saad Ullah, Wenbo Guo, Christopher Kruegel, and Giovanni Vigna. 2026. From Documentation to Zero-day Vulnerabilities: LLM-Driven Fuzzing of JavaScript Engines in PDF Readers. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832609

1 Introduction

The Portable Document Format (PDF)'s support for JavaScript code offers interactive capabilities, where the code is parsed and executed by a dedicated JavaScript engine embedded in the PDF reader. Because this engine runs user-provided code on a host system, it is crucial to identify the vulnerabilities that could lead to malicious code execution. To achieve this, existing efforts [16, 24, 63] generate JavaScript-based test cases to fuzz PDF readers. However, these approaches suffer from significant limitations in both automation and relationship inference. Specifically, Favocado [16] and Cooper [63] require substantial manual effort to analyze natural-language API documentation and construct machine-readable test-generation rules. TypeOracle [24], although automated, relies on simple name-based similarity for relationship inference, missing complex semantic dependencies between API calls. The challenge is compounded by the fact that over 30% of PDF reader functions

are undocumented [24], creating substantial gaps in API coverage. Even tools that attempt to model API relationships [16] only capture basic producer-consumer dependencies, in which one API function's return value directly serves as another's input parameter.

Specifically, by systematically examining Adobe's JavaScript API documentation [3, 4], we identify two relationship types that producer-consumer-only approaches cannot represent. The first is value-constraint relationships, where parameters of two API calls must satisfy equality/inequality constraints (the parameter names need not match; e.g., *addField* and *removeField* require the same *cName* with a specific ordering, while *popUpMenuEx*'s *cName* and *search.query*'s *cQuery* must hold the same string). The second is implicit relationships, in which APIs interact through shared state rather than explicit parameters (e.g., *setAction* registers handlers that are later triggered by *setFocus*). The combination of incomplete API coverage and limited relationship inference creates a substantial gap in testing effectiveness, as both undocumented APIs and complex interaction patterns that could reveal vulnerabilities remain largely unexplored.

In this work, we propose PDFuzzer, which leverages LLMs and constraint solvers to automatically infer comprehensive specifications for both documented and undocumented APIs and to capture intricate relationships among API calls, thereby generating high-quality (sequential) inputs. PDFuzzer identifies and models three types of API relationships—producer-consumer, value-constraint, and implicit—while automatically inferring detailed specifications for the substantial portion of undocumented functionality, enabling comprehensive testing that existing approaches cannot achieve. The key distinction is that PDFuzzer does not stop at producer-consumer relationships. It turns natural-language API semantics into symbolic constraints that capture value coupling, ordering requirements, and state-dependent interactions between API calls. Technically, we first extract comprehensive API specifications through a dual-path approach that handles both documented and undocumented APIs. For documented APIs, we extract structured information from official manuals using regular expressions. For undocumented APIs, we extract their signatures via differential analysis of execution traces [24] and leverage LLMs to infer comprehensive specifications. Here, LLMs capture API design patterns, naming conventions, and parameter semantics to transform signature information into detailed specifications that match the quality of documented APIs, outperforming signature-based approaches. Using these specifications, we create context-free grammars (CFGs) for each API function and its parameters, providing a structured syntax for generating syntactically valid API function invocations. We then use LLMs to infer inter-API relationships, leveraging their ability to perform multi-step reasoning over diverse documentation and previously inferred specifications [59]. This allows us to move beyond simple type matches to identify the dependencies that govern realistic API call sequences. To ensure precision, we express the relationship extracted by LLMs as constraints for pairs of API calls, and use SMT solvers (Z3) to solve the constraints to generate concrete test cases with *sequences* of API calls.

We compared PDFuzzer with three state-of-the-art PDF reader fuzzers (TypeOracle [24], Favocado [16], and Cooper [63]) and two LLM-based methods for general-purpose fuzzing (Fuzz4All [60] and a naive LLM method based on TitanFuzz [14]). We evaluated these methods on three popular PDF readers, Adobe Acrobat Reader, Foxit PDF Reader, and PDF-XChange Editor, using two experiments: (1) tracking coverage over 24 hours and (2) recording vulnerabilities found over two weeks. PDFuzzer reaches up to 48% higher coverage and discovers 31 zero-day vulnerabilities across three targets, compared to at most 6 found by competing tools. These vulnerabilities include null pointer dereferences, buffer overflows, and use-after-free bugs, with potential for arbitrary code execution. We coordinated disclosure of all vulnerabilities with the vendors; 26 have been confirmed or fixed, yielding $2,450 in bug bounties.

We validated our design choices through a comprehensive ablation study on each key component of PDFuzzer, by either turning it off or replacing it with the appropriate baseline technique, such as TypeOracle [24]. Our evaluation demonstrates that each component contributes significantly to the overall effectiveness: undocumented API specification inference achieves up to 28% higher coverage than type-only approaches, parameter-level grammar generation provides up to 18% higher coverage than API function-level grammar generation, strong symbolic relationship modeling improves coverage by up to 8% compared to candidate (co-occurrence-only) relationships, and value-constraint and implicit relationships specifically add up to 15.5% over a producer-consumer-only configuration, and PDF objects integration enhances coverage by up to 8% beyond JavaScript-only approaches. Additionally, spot-checks of the LLM-driven components across our pipeline showed consistent reliability, with 93–98% accuracy in API specification extraction, grammar generation, and relationship inference.

In summary, our main contributions are as follows:

- We propose PDFuzzer, an LLM-based fuzzer designed to achieve deeper program logic coverage in PDF readers. It generates test inputs by extracting detailed specifications for undocumented API calls and leveraging LLMs to infer symbolic relationships between JavaScript functions. Unlike producer-consumer-only approaches, PDFuzzer also infers value-constraint and implicit relationships from natural-language API specifications, enforcing them through SMT solving during test generation.
- We experimentally demonstrate that PDFuzzer achieves up to 48% higher code coverage and discovers 31 zero-day vulnerabilities in three popular PDF readers (Adobe Acrobat Reader, Foxit PDF Reader, and PDF-XChange Editor), significantly outperforming state-of-the-art traditional and LLM-based fuzzers.
- We validate our design through detailed ablation studies on key components, including specification inference, CFG generation, and relationship inference, showing how each component contributes to the overall effectiveness.

2 Background and Existing Work

2.1 JavaScript Engines in PDF Readers

PDF documents can embed JavaScript code to enable interactive features such as spell checking and form submission. To support this functionality, PDF readers include scripting engines that parse and execute the embedded code. Beyond standard JavaScript, these engines expose specialized APIs that invoke native code within the reader, altering how documents are rendered. The APIs supported by PDF readers are documented in vendor API manuals [3, 4]. While powerful, this functionality also introduces risks: adversaries

have exploited vulnerable API functions, such as Adobe's *util.printf* (buffer overflow [1]) and Foxit's *app.launchURL* (remote code execution [2]) to compromise systems. Thorough testing of PDF readers is therefore critical. In particular, dynamic testing requires generating valid PDF documents with realistic JavaScript API calls, a task made challenging by the need to interpret API manuals and produce valid, mutated API calls that are embedded into executable PDF files. Throughout this paper, we use *API specification* to denote the information needed to generate and validate such an API call: its owning object (the JavaScript object that exposes the API, e.g., *app* in *app.popUpMenuEx*), function or property name, parameters, constraints, return value, and behavior.

2.2 Threat Model

We consider a attacker who crafts a PDF document with embedded JavaScript in order to trigger vulnerabilities in the PDF reader's JavaScript engine. The attack is realized when a victim opens the malicious PDF, causing the embedded script to execute inside the reader and potentially leading to memory corruption, information leakage, denial of service, or arbitrary code execution on the victim's system. This threat is realistic and consistent with in-the-wild cases of JavaScript-based PDF exploitation, such as CVE-2024-28888 [52] and CVE-2024-34099 [53]. Accordingly, our goal is to generate high-quality JavaScript-based PDF test cases that can expose vulnerability-triggering behaviors in mainstream PDF readers.

2.3 Existing PDF Fuzzers and Limitations

Due to the closed-source nature of mainstream PDF readers, static analysis is not commonly used for the security evaluation of JavaScript engines. As such, existing work on analyzing the security of JavaScript engines in PDF readers has focused mostly on dynamic testing techniques, more specifically, fuzzing [16, 24, 63]. To perform fuzzing on PDF readers, one needs to embed in PDF documents JavaScript code that invokes a large and diverse set of JavaScript API calls (this is called the fuzzing corpus). One can then input these PDF documents into the PDF reader to observe their behavior. This basic approach is used by all prior work [16, 24, 63], but it suffers from three key limitations:

① **Missing undocumented API specifications.** Favocado [16] and Cooper [63] miss a wide range of potential test cases because they rely solely on official API manuals. TypeOracle [24] can identify undocumented API functions via differential analysis but fails to extract details about their functionality, preventing the generation of high-quality (valid) invocations for these undocumented API functions.

② **Failure to capture complex inter-API relationships.** TypeOracle [24] uses differential analysis on execution traces to infer parameter type information for API functions and finds inter-API relationships through name similarity. API fuzzing approaches, including Favocado [16], focus on producer-consumer relationships, where the return value of one API is used as an input parameter to another API. Both name-based matching and producer-consumer matching are useful, but they are insufficient for relationships whose validity depends on concrete parameter values or shared program state. For example, while *addField* and *removeField* both have a *cName* parameter type, they are related only if the *cName* value is the same. Additionally, the order of these API calls is also important - *addField* needs to be called before *removeField* (and with the same parameter) for them to have a relationship. This illustrates a value-constraint relationship, while implicit relationships arise from shared reader state; Section 3 defines both cases in detail.

③ **Lack of automated specification extraction.** While existing PDF reader fuzzers automate the test case generation and execution process, they still require substantial manual effort to convert natural language API documentation into machine-readable test generation rules. For instance, Favocado [16] demonstrated that even with automated parsing pipelines, significant human expertise is still necessary to manually review hundreds of pages of API documentation, interpret semantic relationships between API functions, and construct comprehensive specification rules for JavaScript APIs. In practice, human experts still need to fill gaps left by parsers. They supplement missing fields, correct extraction errors, and add initialization code for binding objects (internal PDF objects that bridge JavaScript APIs to native reader functions) when environment-specific data is required [16]. Cooper [63] similarly depends on manual analysis of API documentation to create test generation templates. TypeOracle [24] automates API signature extraction via differential analysis, but cannot infer semantic descriptions and behavioral constraints from documentation, limiting its ability to generate high-quality test cases. With Adobe's official API manual [3, 4] exceeding 700 pages of natural language descriptions, manual specification extraction remains time-consuming, labor-intensive, and often incomplete.

3 Preliminary: Inter-API Relationships

Effective fuzzing of PDF readers requires understanding interactions between JavaScript API functions. Existing fuzzers largely generate isolated API calls; however, many vulnerabilities are only triggered through orchestrated sequences of related calls. We use the term *inter-API relationship* to refer to a semantic dependency between two API calls that affects whether they should appear together, in which order they should execute, or what constraints their parameter values or shared state must satisfy. Inter-API relationships come in two strength tiers. Candidate relationships are the weaker tier: a coarse co-occurrence hypothesis encoding only that two API functions may meaningfully appear together, without specific parameter or ordering constraints. Strong symbolic relationships are the stronger tier: confirmed relationships that additionally carry symbolic constraints over parameter values or shared state, and optionally an ordering requirement (Section 4 details how each tier is inferred). From Adobe's JavaScript API documentation [3, 4] and prior vulnerability-triggering test cases [16, 24, 63], we identify three categories of strong symbolic relationships. Producer-consumer relationships capture the type-based data flow commonly modeled by prior API fuzzing work [9, 16]; the other two categories, value-constraint and implicit relationships, are the main types that let PDFuzzer go beyond producer-consumer-only approaches.

3.1 Producer-Consumer Relationship

A producer-consumer relationship connects API_1's return value to an argument of API_2, so the generated sequence must execute API_1 before API_2 and pass forward API_1's return value.

Listing 1: Producer-Consumer Relationship Example

```
var myIcon = Doc.getIcon({cName: "iconName"});
Field.buttonSetIcon({oIcon: myIcon, nFace: 0});
```

Doc.getIcon returns an Icon object representing a named icon in the document, which *Field.buttonSetIcon* then consumes as its *oIcon* parameter to set a button's icon. The dependency is characterized by type compatibility between the producer's return value and the consumer's input parameter.

3.2 Value-Constraint Relationship

A value-constraint relationship is a constraint over parameters from two API calls – such as equality, ranges, or set-membership – that must hold for the calls to interact meaningfully.

Listing 2: Value-Constraint Relationship Example

```
Doc.addField({cName: "myField", cFieldType: "text",
    ...});
Doc.getField({cName: "myField"});
```

Both *Doc.addField* and *Doc.getField* take a *cName* parameter specifying the field name. For meaningful (correct) behavior, the *cName* values must be the same (that is, the function arguments refer to the same field). Also, *addField* must precede *getField*, since a field must exist before it can be retrieved. Unlike producer-consumer, the dependency here is on the actual *values* of two input parameters rather type compatibility between an input and a return value. Moreover, the constraints can be broader that equality and capture more complex relationships.

3.3 Implicit Relationship

An implicit relationship arises when two API functions interact via shared system state or objects rather than explicit parameter connections, with no direct argument-to-argument or return-to-argument data flow.

Listing 3: Implicit Relationship Example

```
Field.setAction({cTrigger: "OnFocus", cScript: "
    some_action"});
Field.setFocus();
```

Field.setAction configures the action to trigger when an event (named by its *cTrigger* parameter) occurs on a field, while *Field.setFocus* transfers keyboard focus to the field and invokes the corresponding event handler. When *setAction* is called with *cTrigger* = `"OnFocus"` before *setFocus*, the focus event dispatches the previously configured action. The dependency runs through the field's shared internal state – *setAction* writes the action configuration and *setFocus* reads it – rather than through any direct argument-to-argument or return-to-argument data flow, and is activated only when *cTrigger* matches the event the second call fires.

4 Methodology

We propose PDFuzzer, an LLM-based fuzzing tool for JavaScript engines in PDF readers that generates higher-quality, complex, and semantically meaningful test cases than prior work, achieving greater code coverage and vulnerability discovery. PDFuzzer systematically addresses the three key limitations of existing PDF reader fuzzers through a principled design that leverages LLMs and constraint solvers.

4.1 Overview

PDFuzzer comprises four main components that collaboratively generate high-quality API call sequences (see Figure 1).

API Specification Extraction. The first component extracts API specifications from two sources: documented APIs via our API Manual Parser over the official manuals [3, 4], and undocumented APIs via differential analysis of execution traces (which yields only minimal signatures *<object>.<method>(<param>: <type>, ...)*). The Specification Inference module then uses an LLM to upgrade these signatures into comprehensive specifications with descriptions and parameter constraints, addressing Limitation 1.

Grammar Generation. This component compiles each specification into context-free grammars (CFGs) for individual API functions and their parameters via a two-phase, parameter-level approach: each parameter's grammar is generated independently, preserving fine-grained constraints (enumerated value sets, format requirements) that a function-level grammar would overgeneralize. This addresses part of Limitation 3.

Relationship Inference. Operating in parallel with Grammar Generation, this component identifies inter-API dependencies (Limitation 2) via a two-stage LLM process with Retrieval Augmented Generation (RAG) [44]: Stage 1 selects candidate API pairs that may co-occur meaningfully; Stage 2 promotes a subset to strong symbolic relationships, encoding parameter constraints as SMT-LIB2 [10] expressions and recording execution ordering. This separates semantic reasoning (LLM) from constraint solving (SMT).

Test Case Generator. The final component synthesizes concrete PDF inputs by instantiating each API from its CFG, sequencing related calls under candidate relationships, and using a Z3 [41] SMT solver to satisfy strong symbolic constraints. We additionally integrate Cooper's [63] cooperative mutation to populate the necessary PDF native structures and apply targeted mutations to a subset of test cases for robustness probing, completing Limitation 3.

4.2 Running Example

We use a single end-to-end example, traced through the rest of this section's component subsections and revisited in our vulnerability case study (Section 6.2), to illustrate how the four components above turn API documentation and execution traces into vulnerability-triggering inputs. The example, shown in Listing 4, is a test case PDFuzzer generated and used to trigger a use-after-free vulnerability in Foxit PDF Reader (vulnerability ID 5 in Table 1, later fixed and assigned a CVE entry).

Listing 4: Running example: PDFuzzer-generated test case triggering a use-after-free vulnerability in Foxit PDF Reader.

```
app.popUpMenuEx({cName: "popup", oSubMenu: {label: "kM
    ", action: "mFv"}});
search.query({cQuery: "popup"});
app.popUpMenu({cItem: "M", Array: ["21"]});
```

This test case simultaneously exhibits two structural features that prior fuzzers cannot capture together. First, Lines 1-2 form a *strong symbolic relationship of type value-constraint*: the *cName* of *app.popUpMenuEx* and the *cQuery* of *search.query* must hold the same string for the memory allocation in *app.popUpMenuEx* and the deallocation in *search.query* to operate on the same memory region. Second, Line 3 (*app.popUpMenu*) is associated with Line 1

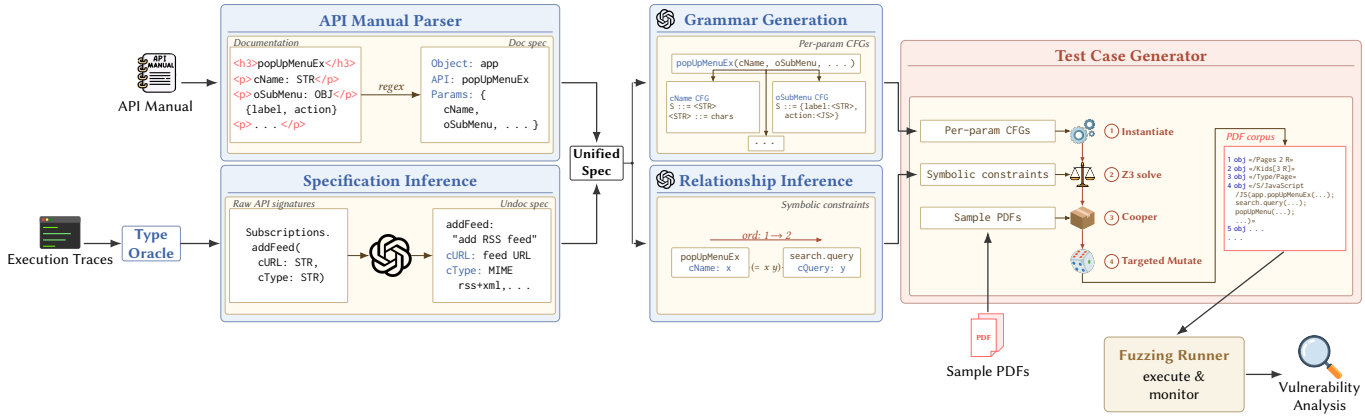


Figure 1: Overview of PDFuzzer. Documented APIs are extracted by the API Manual Parser; undocumented APIs are recovered by Specification Inference from execution traces. Both *Doc spec* and *Undoc spec* feed Grammar Generation and Relationship Inference in parallel. Their outputs (per-parameter CFGs and symbolic constraints) drive the Test Case Generator, which additionally consumes Sample PDFs for native-object templates. The resulting PDF corpus is exercised by the Fuzzing Runner and triaged by Vulnerability Analysis.

only via *candidate co-occurrence*: it contributes a related API call to the sequence but has no parameter-level coupling, so its arguments are generated independently. PDFuzzer models these two tiers explicitly: candidate relationships specify which API functions can co-occur in a sequence, and strong symbolic relationships further enforce parameter-level constraints (encoded as SMT-LIB2 [10] expressions, the standard input format consumed by SMT solvers such as Z3 [41]) and execution ordering. The remaining subsections follow this example through specification extraction, parameter-level grammar generation, two-stage relationship inference, and final test-case generation, with a closing "Running Example" paragraph in each subsection showing how its output shapes the listing (test case) above.

TypeOracle's [24] name-similarity matching would link *app.popUpMenuEx* only to *app.popUpMenu* (which share the `popUpMenu` prefix) and never to *search.query* (whose name has no overlap), so it cannot enforce the required *cName* = *cQuery*. Favocado [16] and RESTler-style [9] producer-consumer matching cannot connect *app.popUpMenuEx* and *search.query* either, since neither call returns a value consumed by the other. The bug therefore remains out of reach for prior fuzzers.

4.3 API Manual Parser and Specification Inference

The JavaScript API manual is an HTML document with a structured enumeration of the available function calls (and their parameters) for the JavaScript engine in PDF readers. The manual lists Adobe's built-in objects and the API functions that objects can call, including their properties. We evaluate three PDF readers: Adobe Acrobat Reader, Foxit PDF Reader, and PDF-XChange Editor. Because Foxit and PDF-XChange do not provide comprehensive API manuals, we use Adobe's API manual as the reference specification: Adobe's JavaScript API has become the de facto standard for PDF JavaScript engines, and Foxit and PDF-XChange both implement it, as noted in prior work [24]. For each API function, the documentation first provides a paragraph explaining its functionality. The manual then lists the parameter names for the API function. Notably, for most parameters, the documentation does not explicitly state what type is required. As a result, simply parsing the documentation is not enough, and we must infer these parameter types for building API invocation rules. If the API function has a return value, the documentation also includes a description of the return value.

We extract specifications for each API function by using regular expressions to match HTML objects and capture hierarchical information. Specific titles identify each API function's content area, which we parse into JSON files. Extracted data includes the object that owns the function, function name, type (method or property), description, parameters, parameter descriptions, and return value.

Specification Inference. PDF readers contain numerous undocumented API functions that are not described in official manuals. These may be internal-only functions not intended for external use, legacy compatibility functions, or vendor-specific extensions. Fuzzing such calls is known to be an effective method for vulnerability discovery, as they often receive less security review and may lack proper input validation [35].

TypeOracle [24] can infer parameter types via differential analysis of execution traces, but cannot capture semantic functionality or behavioral constraints. We close this gap by combining TypeOracle's type inference with LLM-based semantic specification generation in three stages.

Stage 1: Type Extraction. We apply TypeOracle's differential analysis to obtain raw API signatures of the form *<object>.<method>(<param1>: <type1>, ...)*—e.g., *Doc.getField(cName: String)*—by extracting object/method names from internal call tables and inferring parameter types from how instruction operands vary across runs (e.g., a length-related operand changing from 2 to 4 when passing `"YY"` vs. `"zzzz"` indicates a String parameter). These signatures reveal only the structural skeleton; they omit each parameter's role, valid ranges, the function's purpose, return-value semantics, and side effects.

Stage 2: Context Preparation. For each undocumented API, we collect contextual information from the structured JSON specs produced by our *API Manual Parser*. This context includes (1) the parent object's description and functionality, (2) other APIs (documented and undocumented) on the same object, and (3) the parameter names and types inferred in Stage 1.

Stage 3: LLM-based Specification Generation. We prompt the LLM to generate comprehensive specifications using contextual information from Stage 2. Importantly, this applies only to undocumented APIs; documented APIs are handled by our API Manual Parser with regular expressions, since their specifications already exist in the official manuals in natural language. The LLM uses its understanding of API design patterns and naming conventions to infer: (1) functional descriptions for the API function and each parameter, (2) parameter value constraints and valid ranges, (3) return value specifications, and (4) behavioral semantics. For example, given the raw signature *Subscriptions.addFeed(cURL: String, cType: String)*, the LLM reasons that the parent *Subscriptions* object handles feed subscriptions, that *cURL* carries a feed URL, and that *cType* must be a MIME/content-type string with concrete values such as `"application/rss+xml"` or `"application/json"`. This inference is drawn from the API name, parameter names, and parent-object context alone, since the *Subscriptions* object has no entry in Adobe's manual and no documented counterpart can serve as a template. TypeOracle alone yields only `{cURL: String, cType: String}`, insufficient for invocations a feed engine can parse.

These outputs are expressed in natural language, consistent with Adobe's API documentation, which primarily consists of prose descriptions rather than formal constraints.

The LLM generates specifications in a structured JSON format consistent with documented APIs, ensuring pipeline uniformity for both types. The JSON output includes fields requiring the LLM to explain its reasoning, enabling us to interpret and validate its inferences during accuracy evaluation (Section 6.6).

Error Handling. To handle potential LLM errors, we implement JSON format verification. Our error recovery strategy automatically retries failed attempts up to a specific number of times, incorporating specific error feedback into subsequent prompts (e.g., "Previous JSON parsing failed: [error]. Please generate a valid JSON format").

Running Example. All three APIs in Listing 4 (*app.popUpMenuEx*, *search.query*, and *app.popUpMenu*) are documented in Adobe's manual and therefore traverse the Manual Parser path, producing JSON specifications listing each API's parameters (e.g., *popUpMenuEx* owns *cName* and the nested *oSubMenu*). The Specification Inference path is exercised by undocumented APIs.

4.4 Grammar Generation

After extracting API specifications for documented and undocumented API functions, we have structured JSON specifications containing natural-language descriptions of all the API functions that the JavaScript engine makes available. Each JSON specification follows a hierarchical format where API function parameters are organized with their individual descriptions and types. However, systematic and automated test case generation requires these specifications in a machine-readable format suitable for programmatic processing. To achieve this, we leverage context-free grammars (CFGs), drawing inspiration from grammar-based fuzzing approaches [7, 30] that use CFGs to automatically generate syntactically valid test inputs. CFGs provide an ideal representation for API call statements and parameter generation rules due to their structured, parseable syntax and ability to capture complex constraints. Our initial approach involved directly feeding complete API function specifications to an LLM to generate comprehensive CFGs. While this method works effectively for simple API functions with few parameters (such as property getters and setters), it struggles with complex API functions with numerous parameters (as discussed below).

Challenge: Information Overload in Complex APIs. The challenge became apparent with API functions like *Doc.submitForm*, which requires 23 distinct parameters. Consider the *cSubmitAs* parameter: although recognized as *String* type, it accepts only six specific values (`FDF`, `XML`, `XDF`, etc.) corresponding to supported file formats. Processing such complex specifications in a single prompt causes information overload for LLMs. Consequently, this leads to overgeneralized grammars that generate random strings, failing to respect the constrained value sets. This loss of granularity occurs because LLMs struggle to prioritize critical parameter-specific constraints when processing lengthy specifications within fixed context windows.

Solution: Two-Phase Parameter-Level Grammar Generation. To address this challenge and capture fine-grained parameter constraints, we design a two-phase approach:

Phase 1: Specification Decomposition. We parse the structured JSON format to systematically isolate each parameter's natural language description from the complete API specification. For an API with n parameters, this decomposition creates n independent specification units, each preserving parameter-specific constraints, value restrictions, and type requirements. This decomposition ensures that each parameter's unique constraints receive focused attention during grammar generation.

Phase 2: Targeted Grammar Synthesis. Each parameter specification unit is processed by the LLM using specialized prompts that include metadata, while maintaining focus on the specific parameter. The metadata includes API type (distinguishing between methods and properties with different invocation syntaxes), parent object, parameter name, and description. The prompts incorporate detailed sample JSON grammar structures demonstrating proper BNF-style rule decomposition and enforce strict syntactic rules, including character escaping, complete symbol definition, and adherence to value generation constraints (e.g., enumerated string sets).

This parameter-focused approach captures syntactic patterns and semantic constraints that would otherwise be missed in comprehensive function-level processing. The resulting grammars encode parameter-specific requirements, such as constrained value sets, numeric ranges, and format specifications, leading to higher-quality test case generation that respects API semantics while maintaining syntactic validity.

Error Handling and Normalization. Despite careful prompt engineering, CFG-generated API parameters sometimes violate JavaScript engine syntax requirements. We observed four recurring

formatting issues that cause parsing failures in target readers. These include missing quotes for string literals, leading zeros in numeric values (e.g., 007), unescaped quotes inside strings, and embedded newline characters.

To address these inconsistencies, we apply lightweight type-specific post-processing during test-case generation: strings are cleaned and properly quoted, numbers are stripped of leading zeros, and arrays and objects are recursively normalized and reconstructed as valid JavaScript syntax. Invalid object keys are prefixed with a random letter. We also avoid extended Unicode escapes (`\u{XXXX}`), which are unsupported by Adobe Acrobat Reader. This normalization prevents syntactic errors from causing test-case rejection while preserving the intended parameter semantics.

Running Example. For *app.popUpMenuEx* in Listing 4, parameter-level decomposition produces independent CFGs for *cName* (a character-level string grammar) and *oSubMenu* (a nested-object grammar with separate rules for the *label* and *action* fields). A function-level grammar would tend to flatten *oSubMenu* into an arbitrary value and lose the nested-field structure required by the target PDF reader's JavaScript engine.

4.5 Relationship Inference

The previous step of PDFuzzer results in grammars generated for API functions and their parameters. However, many vulnerabilities can only be triggered when multiple related API functions are called in a specific sequence [16, 19, 51]. Therefore, we need a method to infer relationships between API functions so that we can generate sequences of related API calls during test case generation.

We group inter-API relationships into two strength tiers: candidate relationships and strong symbolic relationships. A candidate relationship is a coarse co-occurrence hypothesis: two API functions may be meaningful in the same sequence, but no order, argument mapping, or parameter constraint has been established. A strong symbolic relationship is a confirmed relationship that records related arguments or return values, symbolic variables, an SMT-LIB2 constraint, an optional ordering requirement, and one of the semantic relationship types defined in Section 3.

Analyzing all $\binom{n}{2} = 221{,}445$ pairwise relationships among $n = 666$ APIs at ∼30 s of LLM analysis each would take over 1,845 hours and is intractable. We instead use a two-stage approach that turns this $O(n^2)$ problem into $O(n) + O(k^2)$ with $k \ll n$: Stage 1 uses fast RAG queries to surface candidate pairs (linear in n); Stage 2 runs expensive constraint analysis only on those candidates.

Stage 1: Candidate Relationship Extraction. We iterate over each API and prompt the LLM to identify related APIs from our specification database, stored in a Vector Store and queried via RAG [44] to avoid context-window overflow (full prompt is available in our public artifact). The output is a set of candidate API pairs only; producer-consumer, value-constraint, and implicit category assignment happens in Stage 2.

Stage 2: Strong Relationship Analysis and Symbolization. As introduced in Section 3, each strong symbolic relationship is assigned one semantic type: Producer-Consumer, Value-Constraint, or Implicit. For each candidate API pair, we use a single comprehensive prompt to analyze their specifications and determine if constraint-based relationships exist. The prompt performs zero-shot analysis, instructing the LLM to simultaneously accomplish three key tasks in one inference pass: (1) *Parameter Symbolization*: assign symbolic variables (e.g., x, y) to related parameters or return values, (2) *Constraint Generation*: express the relationship as SMT-LIB2 constraints (e.g., (= x y) for equality), and (3) *Sequence Analysis*: determine if the API functions must be executed in a specific order.

The LLM returns a structured JSON object containing the API function names, parameter mappings, symbolic variables, execution sequence requirements, parameter types, and the SMT-LIB2 constraint expression. For instance, when analyzing *Doc.addField* and *Doc.getField*, the LLM identifies that their *cName* parameters must be equal and that *addField* must precede *getField*. The constraint `(= x y)` captures the parameter equality requirement, while the sequence flag indicates the execution dependency.

Integration with Test Generation. The inferred relationships are stored as constraint templates that guide the Test Case Generator. During test sequence construction, candidate relationships determine which API functions can be combined, while strong symbolic relationships provide the constraints that an SMT solver must satisfy when generating concrete parameter values. This two-tier approach enables efficient generation of complex, semantically meaningful API call sequences.

Running Example. For Listing 4, Stage 1 returns two candidate pairs: *(app.popUpMenuEx, search.query)* and *(app.popUpMenuEx, app.popUpMenu)*. Stage 2 promotes the first to a strong symbolic relationship of type value-constraint with parameter symbols *app.popUpMenuEx.cName* $\mapsto$ x, *search.query.cQuery* $\mapsto$ y, constraint `(= x y)`, and ordering *app.popUpMenuEx* $\rightarrow$ *search.query*; the second pair remains a candidate, contributing a related API call without parameter-level coupling.

4.6 Test Case Generator

The Test Case Generator synthesizes concrete PDF test inputs by integrating outputs from all previous components. The generation process follows a four-stage pipeline: individual API call generation, relationship-aware sequencing, constraint satisfaction, and targeted mutation.

Stage 1: Individual API Call Generation. For each target API function, we use the corresponding CFG from the Grammar Generation component to generate syntactically valid API calls. The CFG rules guide the systematic construction of parameter values, ensuring type correctness and adherence to documented constraints. This stage produces a pool of well-formed individual API calls that serve as building blocks for sequence construction.

Stage 2: Relationship-Aware Sequencing. To construct API call sequences (denoted as $c_1, c_2, ..., c_n$), we begin with a seed API call c_1 and consult the relationships identified by the Relationship Inference component. The sequencing strategy differs based on relationship tier:

(a) For candidate relationships, we select related API functions and generate them independently using their respective CFGs. These calls are concatenated without parameter-level coupling, as candidate relationships only require co-occurrence without specific parameter constraints.

(b) For strong symbolic relationships, we enforce parameter-level dependencies during generation. Starting with the parameter values generated for c_1, we extract the symbolic constraints from the Relationship Inference component and instantiate them with concrete values. We then invoke an SMT solver (Z3) to determine parameter values for c_2 that satisfy the relationship constraints. This process continues iteratively to build longer sequences that respect all symbolic dependencies. The maximum sequence length is configurable, and based on our empirical evaluation, we set it to 2,000 API calls. Longer sequences could result in excessive PDF execution time, making it difficult to distinguish between normal processing and potential hangs during testing.

Stage 3: Object-API Relationship Integration. After generating JavaScript API call sequences, PDFuzzer also needs PDF document structures that make those calls meaningful. We therefore incorporate Cooper [63], a cooperative mutation technique that links JavaScript APIs with PDF native objects, i.e., structural objects inside the PDF file such as pages, form fields, actions, and annotations. The resulting object-API relationships specify which native object classes should appear in a generated document, and are separate from the inter-API relationships among JavaScript calls. Cooper builds these relationships by clustering native objects from a PDF corpus and identifying object classes correlated with successful execution of API groups. During test-case construction, after generating API call sequences, Cooper consults this object-API relationship map to identify relevant PDF object classes that should be present in the document. Cooper enriches our test cases by extracting PDF objects from sample documents and applying targeted mutations and combinations to these objects. This produces richer PDF documents with diverse native structures, extending coverage of PDF object processing code paths while ensuring the presence of native structures needed to support JavaScript operations.

Stage 4: Targeted Mutation. The previous three stages are designed to generate syntactically and semantically valid JavaScript code that adheres to API specifications and constraints extracted from official manuals, producing test cases that should execute without crashes under normal circumstances. To test the robustness of the target JavaScript engines when receiving malformed inputs, we apply selective mutations to a subset (approximately 15%) of the generated test cases. For strings, we target decoder and escape handling by injecting non-BMP Unicode, escape sequences (e.g., `\uXXXX`, `\xXX`), and malformed encodings. For numbers, we target conversion edge cases using boundary values, `Infinity`, `NaN`, scientific-notation variants, and hexadecimal literals.

The complete generation pipeline produces test cases ranging from specification-compliant inputs (for functional testing) to deliberately malformed inputs (for security testing). Our implementation extracts 435 API functions with 507 parameters from documented specifications and 231 API functions with 379 parameters from undocumented ones, yielding CFG-based rules for 886 parameters across 666 API functions. Function selection follows a weighted random distribution based on relationship frequency.

Running Example. Stage 1 instantiates each API call invocation from its grammar, e.g., *app.popUpMenuEx(cName:"popup", oSubMenu:{label:"kM", action:"mFv"})* and *search.query(cQuery:?)*. Stage 2 substitutes *x="popup"* into `(= x y)` and Z3 solves *y="popup"*, fixing *cQuery="popup"*. Stage 3 (Cooper) supplies the necessary native PDF objects (e.g., the form-field/AcroForm and page objects that the API calls operate on). Stage 4 may inject Unicode escapes via mutation. The result is exactly Listing 4, the test case that triggered a use-after-free vulnerability later fixed by Foxit and assigned a CVE entry.

5 Evaluation Methodology

In this section, we detail our evaluation methodology for PDFuzzer. We compare PDFuzzer against current state-of-the-art techniques in two separate experiments: (1) we assess how much code can be reached (coverage), and (2) we test how many vulnerabilities can be found.

5.1 Experiment Setup

We compare PDFuzzer against the current three state-of-the-art fuzzers for PDF readers: TypeOracle [24], Favocado [16], and Cooper [63]. These fuzzers start by generating a large input corpus for the target PDF readers and iteratively run each generated target through the PDF reader, observing its behavior. This is similar to PDFuzzer's approach, making it easy to compare against these tools.

Since PDFuzzer leverages LLMs, we also compare our tool against two general-purpose LLM-based fuzzing techniques. The first is Fuzz4All, which is an LLM-based universal fuzzer capable of generating test cases for various types of inputs, independent of the fuzzing target [60]. It does so by using two LLMs: a distillation LLM and a generation LLM. The distillation LLM takes arbitrary input related to the to-be-generated test cases (such as documentation or source code) and generates a prompt that will be used by the generation LLM to create fuzzing inputs. As a last comparison, we use a naive LLM-based approach inspired by TitanFuzz [14], which assumes that documentation related to the to-be-generated fuzzing input is already part of an LLM's training corpus. Under these assumptions, there is no need to provide additional input related to the API manual since it is already part of the model's data. To generate an input corpus, we therefore simply prompt the LLM to generate test cases for fuzzing the JavaScript engine of PDF fuzzers.

For targets, we selected three widely used PDF readers with built-in JavaScript engines: Adobe Acrobat Reader v24.005.20421, Foxit PDF Reader v2024.4.0.27683, and PDF-XChange Editor v10.5.2.395 (all of which were the latest versions at the time of this study).

5.2 Experiment Design and Metrics

We configure Cooper [63], Favocado [16] using their default settings. For TypeOracle [24], in addition to using its default settings, we also compare its integration with Cooper [63] and Favocado [16]. For Fuzz4All [60], we use the JavaScript API manual [3, 4] as input. By default, Fuzz4All uses GPT-4 for the distillation LLM and StarCoder for the generation LLM. However, both models are outdated, and newer versions are available. As such, we run Fuzz4All using GPT-4o for the distillation LLM and StarCoder2 for the generation LLM. For the naive vanilla-LLM method, we test three of the most popular contemporary models: GPT-4o [45] (OpenAI's foundational general-purpose model), GPT-o3-mini [46] (OpenAI's reasoning model), and

Claude-3.7-Sonnet [6] (Anthropic's reasoning model), which were the latest versions available at the time of the experiment.

Our evaluation does not compare different online fuzzing loops. Instead, we standardize runtime exploration (fuzzing) across tools and compare only the different test-case generators. Concretely, prior systems such as Cooper [63] and Favocado [16] primarily contribute test-case-generation components. TypeOracle [24] additionally provides an open-source wrapper for lightweight execution. In our experiments, we use TypeOracle's public fuzzing wrapper uniformly for all methods. The runtime process is therefore identical across tools: generate test cases, execute and monitor them, and record the outcomes. It does not involve an AFL-style seed queue, power schedule, mutation strategy, or any coverage-guided feedback loop. Accordingly, any substantive difference among the compared systems stems from the quality of the generated test cases, rather than any different scheduling or feedback mechanisms.

Code Coverage. Because our targets are closed-sourced, we cannot directly instrument the PDF readers to collect code coverage. Instead, we rely on DynamoRIO [17] to dynamically instrument each target program and collect basic block coverage. For each tool, we perform five independent 24-hour runs on each target PDF reader.

Vulnerability Discovery. Due to the significant execution overhead and instability introduced by dynamic instrumentation tools like DynamoRIO (which can cause false positive crashes), we separate our vulnerability detection experiments from coverage collection. We let each tool generate an input corpus of test cases for each PDF reader. For each input, we run it through all three targets and detect when a target crashes using the Windows-provided `werfault.exe` [40]. We then manually analyze each crash to determine whether it was caused by a vulnerability or not. We ran each tool for two weeks on all of the target PDF readers.

Each experiment runs in a VMware Workstation virtual machine (VM) hosted on a system with an 8-core Intel Core i7-7700 processor (3.60GHz) and 64GB RAM. Each VM is configured with 4 CPU cores, 8GB of RAM, and Windows 8.1 - chosen for its lower resource requirements compared to newer Windows versions. All of our target PDF readers are fully compatible with Windows 8.1, ensuring that their use does not adversely affect the results. Whenever a crash was found by a tool, we verified its reproducibility in a newer version of Windows (Windows 10 22H2 and Windows 11 24H2) to ensure compatibility with newer operating systems.

6 Evaluation Results

6.1 Basic Block Coverage

Figure 2 shows the mean basic-block coverage achieved by each tool across all three targets. We observe that PDFuzzer (blue curve) consistently achieves the highest coverage, improving by up to 37% over TypeOracle (up to 19% over TypeOracle+Cooper and up to 17% over TypeOracle+Favocado), up to 15% over Favocado, and up to 48% over Cooper. These gains demonstrate that our higher-quality inputs indeed explore substantially more code than existing methods.

For the naive LLM-based approach, Claude-3.7-Sonnet generally performs best among LLMs, yet still underperforms PDFuzzer. Specifically, PDFuzzer achieves between 16% and 49% higher coverage compared to Claude-3.7-Sonnet. For Fuzz4All, we notice that the majority of generated test cases were invalid JavaScript code, mostly containing C code or Java code, or even just plain text. As a result, Fuzz4All has the lowest coverage on both Foxit and PDF-XChange. For Adobe, Fuzz4All is not the worst performer, which we assume is due to Adobe's more extensive error handling within the PDF reader.

Listing 5 shows an example input generated by PDFuzzer that demonstrates a value-constraint relationship between *customDictionaryCreate* and *addWord*, which only PDFuzzer captured in its input corpus. The *customDictionaryCreate* call creates a custom dictionary with a unique identifier specified by the *cName* parameter. This operation sets up the necessary precondition for any subsequent operations that depend on an existing dictionary. In the next step, *addWord* adds a new word into this dictionary referencing the same *cName* value from the previous call. This order of execution ensures that *addWord* is not executed without a prior successful call to *customDictionaryCreate*, maintaining control of the necessary dependency. Furthermore, the generated test case includes a valid *cLanguage* value (`en_US`). Although the type of the *cLanguage* parameter is `String`, an additional value constraint enforces that the string is a valid language code. PDFuzzer was able to capture this constraint and enforce it in this test case, demonstrating that merely knowing the parameter type (as done by previous work) is typically insufficient to generate valid parameter values.

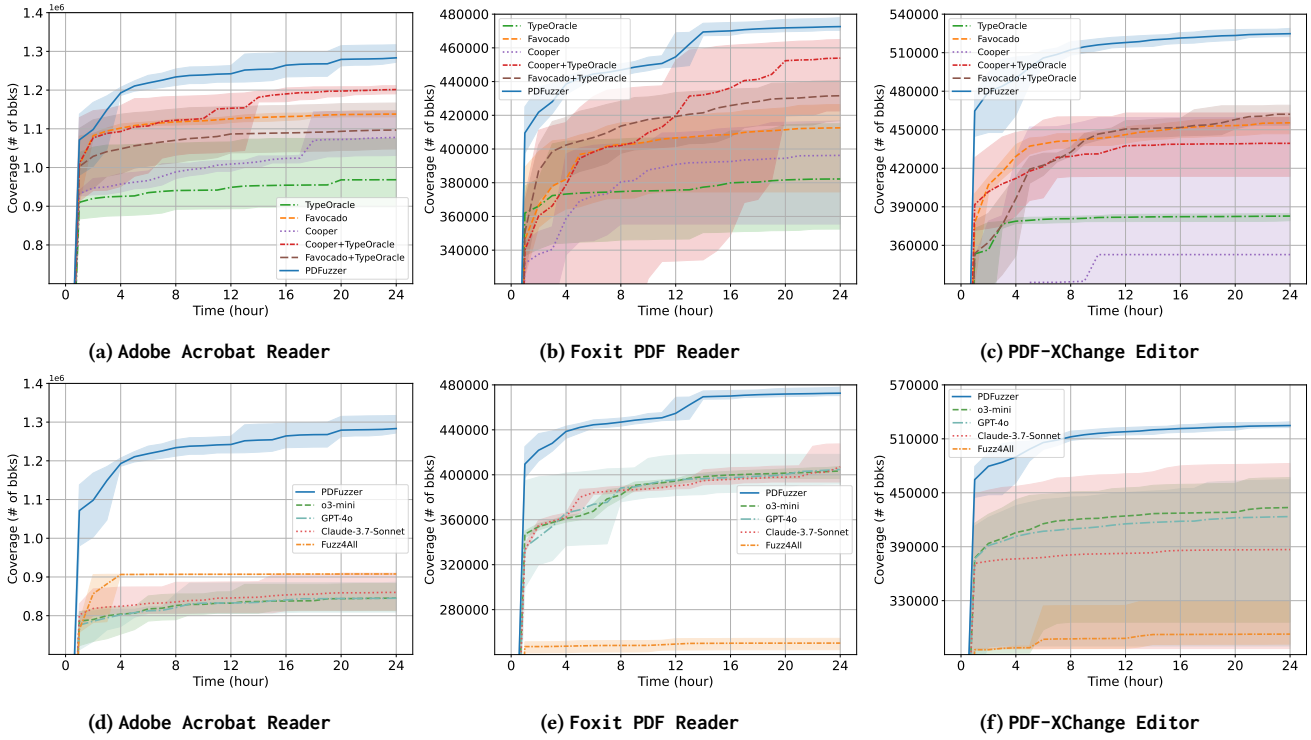
Listing 5: Example that PDFuzzer can capture more relationships and generate higher quality arguments

```
spell.customDictionaryCreate({
    cName: "\\xfe\\xff\\u1a18\\ud428\\uf9cc",
    bShow: false,
    cLanguage: "en_US"})
spell.addWord({
    cName: "\\xfe\\xff\\u1a18\\ud428\\uf9cc"}),
    cWord: "\\uea83\\u5e92\\uda70\\udcd4\\uf619\\u1775
        \\ud827\\udce1"})
```

6.2 Vulnerability Discovery

During the two-week vulnerability discovery campaign, PDFuzzer generated 57 test cases that consistently triggered crashes across the three PDF readers (23 in Adobe Acrobat Reader, 24 in Foxit PDF Reader, and 10 in PDF-XChange Editor). We manually triaged these inputs in two steps. First, for each test case, we performed statement-level reduction by iteratively deleting JavaScript statements and keeping the smallest subset that still reproduces the crash. Second, we deduplicated the minimized crashes by comparing normalized call-stack signatures, following TypeOracle [24] and Cooper [63]. After triage, the 57 crashing inputs collapse to 31 unique crash signatures, which we report as 31 distinct zero-day vulnerabilities discovered by PDFuzzer. None of the other tools was able to find a vulnerability that PDFuzzer was not able to find. We reported all vulnerabilities to the respective parties; two of the vulnerabilities have already received a bounty for a total of $2,450.

Table 1 lists all vulnerabilities found in the three PDF reader targets. Of the 31 unique zero-day vulnerabilities we discovered, 11 are high-severity issues that can potentially lead to arbitrary code execution (ACE). As of this writing, vendors have fixed 26 of the 31 vulnerabilities, and 10 have been assigned CVE entries. Below, we discuss one case study of the vulnerabilities found by PDFuzzer.



(a) Adobe Acrobat Reader (b) Foxit PDF Reader (c) PDF-XChange Editor

(d) Adobe Acrobat Reader (e) Foxit PDF Reader (f) PDF-XChange Editor

Figure 2: (Top row) PDFuzzer vs. traditional tools. (Bottom row) PDFuzzer vs. LLM-based tools.

Case Study: Vulnerability Involving Strong Relationship. The running example introduced in Section 4.2 (Listing 4) is precisely the test case generated by PDFuzzer that triggered this use-after-free vulnerability in Foxit PDF Reader, corresponding to ID 5 in Table 1 (later fixed and assigned a CVE entry). Specifically, the first API function *app.popUpMenuEx*, which creates a pop-up menu at the current mouse position, allocates a memory region through dynamic memory management. The second API function *search.query* searches for characters corresponding to *cQuery* within the document scope, but the call to *search.query* releases the memory region allocated by *app.popUpMenuEx*. When we call the popup menu-related API functions again, this memory address is accessed again, causing a use-after-free vulnerability. As can be seen, this vulnerability requires a value-constraint relationship between the *cName* parameter of *app.popUpMenuEx* and the *cQuery* parameter of *search.query*. It is extremely unlikely to generate a test case where these two values are equal through TypeOracle's [24] random generation. This relationship also goes beyond Favocado's [16] type-based matching between return value and parameter. Therefore, it can only be discovered by PDFuzzer. In our two-week evaluation, this bug was only triggered when PDFuzzer enabled strong symbolic relationships (P^s/P^f in Table 1), while baseline tools (TypeOracle, Favocado, Cooper, and their combinations) did not trigger it.

6.3 Inferred Relationship Distribution

Among 1,019 inferred strong symbolic relationships, 887 (87.0%) are value-constraint, 63 (6.2%) are implicit, and only 69 (6.8%) are producer-consumer; a producer-consumer-only model would miss most of the strong symbolic relationships PDFuzzer instantiates. Table 2 additionally breaks down value-constraint subtypes (showing patterns beyond direct equality).

6.4 Strong Symbolic Relationship Instantiation

A strong symbolic relationship is useful only if its constraints can be solved during generation to produce a concrete API call sequence. To assess how reliably this happens, we generated 1,000 PDF test cases and tracked, for every selected strong symbolic relationship, whether the Test Case Generator (Section 4, Stage 2) successfully produced a satisfying assignment. Table 3 reports the resulting success rates, broken down by semantic type.

Out of 71,586 selected strong symbolic relationships, 65,508 (91.5%) are successfully instantiated, indicating that the relationships PDFuzzer infers are practical to enforce during generation. Value-constraint relationships achieve the highest success rate (95.1%) because their constraints are typically simple equalities or set-membership conditions that Z3 solves directly; producer-consumer relationships reach 81.5%, occasionally failing when an upstream parameter has already been bound to an incompatible concrete value. Implicit relationships are the hardest to instantiate (64.0%), since they often require both a parameter constraint (e.g., a specific event name) and a state precondition (e.g., the target field being focusable). Most of the remaining failures stem from two sources: (1) the targeted mutation pass (Stage 4) intentionally violates the constraint to probe robustness, and (2) implementation edge cases such as parameter values being assembled before the

Table 1: Discovered Zero-Day Vulnerabilities. Severity: ACE (High Severity), Information Disclosure (Moderate Severity), DoS (Low Severity); T: TypeOracle, F^1: Favocado, L: Pure LLM, F^2: Fuzz4All, C: Cooper, F+T: Favocado+TypeOracle, C+T: Cooper+TypeOracle; P^a: PDFuzzer with Function-level Grammar, P^p: PDFuzzer with Param-level Grammar, P^c: PDFuzzer with Candidate Relation; P^s: PDFuzzer with Strong Relation, P^f: PDFuzzer-Full

ID	Target	Type	Potential Impact	Status	Baseline Tools							PDFuzzer Variants				
					T	F^1	L	F^2	C	F+T	C+T	P^a	P^p	P^c	P^s	P^f
1	Adobe	Out-of-bounds Write	Arbitrary Code Execution	CVE-2025-43575											✓	✓
2	Adobe	Memory Corruption	Arbitrary Code Execution	Confirmed	✓	✓			✓	✓	✓		✓	✓	✓	✓
3	Adobe	Buffer Overflow	Arbitrary Code Execution	Fixed	✓	✓				✓	✓			✓	✓	✓
4	Adobe	Use-after-free	Arbitrary Code Execution	Fixed												✓
5	Foxit	Use-after-free	Arbitrary Code Execution	CVE-2026-3777											✓	✓
6	Foxit	Use-after-free	Information Disclosure	CVE-2025-55308		✓				✓	✓			✓	✓	✓
7	Foxit	Use-after-free	Arbitrary Code Execution	CVE-2025-55314											✓	✓
8	Foxit	Buffer Overflow	Arbitrary Code Execution	CVE-2025-55312											✓	✓
9	Foxit	Out-of-bounds Read	Information Disclosure	Fixed	✓					✓	✓	✓	✓	✓	✓	✓
...	Foxit	Out-of-bounds Read	Information Disclosure	Fixed											✓	✓
15	Foxit	Out-of-bounds Read	Information Disclosure	CVE-2025-55307										✓	✓	✓
16	Foxit	Out-of-bounds Read	Information Disclosure	Fixed									✓	✓	✓	✓
17	Foxit	Out-of-bounds Read	Information Disclosure	Fixed										✓	✓	✓
18	Foxit	Uncontrolled Recursion	Denial-of-service	CVE-2026-3778										✓	✓	✓
19	Foxit	Null-pointer-dereference	Denial-of-service	CVE-2025-55313											✓	✓
20	Foxit	Null-pointer-dereference	Denial-of-service	CVE-2026-3776								✓	✓	✓	✓	✓
21	Xchange	Memory Corruption	Arbitrary Code Execution	CVE-2025-6661											✓	✓
22	Xchange	Use-after-free	Arbitrary Code Execution	Fixed											✓	✓
23	Xchange	Use-after-free	Arbitrary Code Execution	Fixed											✓	✓
24	Xchange	Use-after-free	Arbitrary Code Execution	Fixed												✓
25	Xchange	Out-of-bounds Read	Information Disclosure	Fixed									✓	✓	✓	✓
...	Xchange	Null-pointer-dereference	Denial-of-service	Submitted											✓	✓
29	Xchange	Null-pointer-dereference	Denial-of-service	Submitted	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓
30	Xchange	Stack Exhaustion	Denial-of-service	Fixed	✓					✓	✓	✓	✓	✓	✓	✓
31	Xchange	Stack Exhaustion	Denial-of-service	Submitted					✓							✓
Aggregated Results					5	4	1	0	2	6	6	4	7	12	28	31

Table 2: Distribution of inferred strong symbolic relationships, including subtypes of value-constraint relationships.

Category	Count	Proportion
Strong symbolic relationships		
Producer-Consumer	69	6.8%
Value-Constraint	887	87.0%
Implicit	63	6.2%
Total	**1,019**	**100.0%**
Value-Constraint subtypes		
Direct Equality	667	75.2%
Relational Comparisons	69	7.8%
Set Memberships	59	6.7%
Range Constraints	26	2.9%
String Constraints	8	0.9%
Complex Relationships	58	6.5%
Total	**887**	**100.0%**

SMT solver can re-bind them; genuinely unsatisfiable constraint sets are rare.

Table 3: Generation success and failure breakdown by semantic type for strong symbolic relationships over 1,000 generated PDF test cases.

Type	Invocations	Success	Failure	Rate
Producer-Consumer	4,704	3,832	872	81.5%
Value-Constraint	60,675	57,706	2,969	95.1%
Implicit	6,207	3,970	2,237	64.0%
Total	**71,586**	**65,508**	**6,078**	**91.5%**

6.5 Cost Analysis

To run PDFuzzer end-to-end, the total cost inferred by the LLM components is $60.77, $73.38, and $92.36 for GPT-4o, o3-mini, and Claude, respectively. For our naive LLM approach, this cost is significantly higher, rising to $76.14, $156.76, and $174.83 for GPT-4o, o3-mini, and Claude, respectively. PDFuzzer is, therefore, not only better at exploring code and finding vulnerabilities compared to the naive LLM method, but it also does so at a lower cost. This makes our tool more accessible to be used in practice by users with limited resources. Furthermore, the average generation time per test case for the naive LLM method exceeds 60 seconds for the OpenAI models and up to 90 seconds for Claude, highlighting

computational inefficiency and scalability limitations of pure LLM-based approaches. In contrast, PDFuzzer generates approximately three outputs per second for strong symbolic relationships and can achieve speeds as high as 20 outputs per second for regular test cases. This difference in throughput underscores the efficiency and scalability of PDFuzzer compared to LLM-based methods.

6.6 The Accuracy of LLM in Each Step

We randomly sampled 60 LLM outputs from each of PDFuzzer 's LLM-based steps and compared them to our manual annotations. The sample size of 60 was chosen considering the manual annotation burden, where each output requires expert verification to determine correctness.

Specification Inference. We reverse-engineered 60 undocumented API functions (covering 87 parameters) to obtain their ground-truth type information. PDFuzzer achieved 94% accuracy (82/87 correct) in identifying the parameter types, with mistakes spread across 4 API functions. Analysis of these errors reveals two distinct patterns in LLM reasoning. In successful cases, the LLM corrected TypeOracle's misclassifications by leveraging semantic understanding from parameter names. For example, when TypeOracle incorrectly classified the *cFileFilter* parameter of *browseForMultipleDocs* as a Boolean, the LLM correctly identified it as a String type based on the "Filter" naming convention and browsing context. However, the LLM exhibited systematic errors when parameter names strongly suggested Boolean semantics but actual implementations required string types. In the *dcSignup* API function, both *cEmailPerm* and *cConnectPerm* parameters were incorrectly classified as Boolean due to the "Perm" suffix implying permissions, when they actually accept string values representing permission levels.

Grammar Generation. On 60 API functions (covering 30 documented functions with 45 parameters and 30 undocumented functions with 58 parameters), PDFuzzer achieves 97% accuracy (58/60 correct) for APIs and 98% for parameters (101/103 correct).

Relationships Inference. On 60 relation pairs, PDFuzzer achieved 93% accuracy (56/60 correct), with three errors in API sequence identification and one in constraint inference.

The high accuracy of LLMs across the pipeline underscores our effective design, while PDFuzzer's overall performance suggests that occasional LLM errors have minimal impact. It is worth noting that, as a fuzzing method, our seed generation accuracy is already remarkably high compared to traditional fuzzers that rely on random mutations, where valid inputs are far rarer. Moreover, accuracy is not the typical primary goal in fuzzing; instead, the emphasis lies in broad coverage and exposing edge cases. Imperfect inputs from LLM errors may even be beneficial by contributing to the exploration of unexpected program behaviors, aligning with fuzzing's objective of uncovering vulnerabilities through diverse inputs.

7 Ablation Experiments

We validate the efficacy of PDFuzzer's components along four dimensions: (1) undocumented API specification inference; (2) context-free grammar generator; (3) relationship symbolizing, and (4) PDF objects generation integration. Additionally, we evaluate the robustness of PDFuzzer 's performance across different LLMs. Results are shown in Table 4.

Table 4: Component ablation study along four dimensions.

Dims.	Target	Setting	24H Cov.	Change
Dim. 1: Undoc. APIs	Adobe	**PDFuzzer**	1008279	–
	Adobe	TypeOracle	725776	↓ 28%
	Foxit	**PDFuzzer**	349635	–
	Foxit	TypeOracle	279361	↓ 20%
	XChange	**PDFuzzer**	336724	–
	XChange	TypeOracle	333408	↓ 1%
Dim. 2: Grammar Granularity	Adobe	**Param-Level**	1121910	–
	Adobe	Function-Level	921160	↓ 18%
	Foxit	**Param-Level**	406448	–
	Foxit	Function-Level	371901	↓ 9%
	XChange	**Param-Level**	440480	–
	XChange	Function-Level	416297	↓ 5%
Dim. 3: Relation Tier	Adobe	**Strong**	1183295	–
	Adobe	Candidate	1141773	↓ 4%
	Adobe	PC-Only	1133093	↓ 4%
	Adobe	Favocado	1137989	↓ 4%
	Adobe	None	1121910	↓ 5%
	Foxit	**Strong**	459634	–
	Foxit	Candidate	425201	↓ 8%
	Foxit	PC-Only	397916	↓ 13%
	Foxit	Favocado	412543	↓ 10%
	Foxit	None	406448	↓ 12%
	XChange	**Strong**	500579	–
	XChange	Candidate	466248	↓ 7%
	XChange	PC-Only	488575	↓ 2%
	XChange	Favocado	457543	↓ 9%
	XChange	None	440480	↓ 12%
Dim. 4: PDF Objects	Adobe	**PDFuzzer-full**	1283296	–
	Adobe	PDFuzzer-js	1183295	↓ 8%
	Foxit	**PDFuzzer-full**	472620	–
	Foxit	PDFuzzer-js	459634	↓ 3%
	XChange	**PDFuzzer-full**	524782	–
	XChange	PDFuzzer-js	500579	↓ 5%

Dimension 1: Undocumented specification inference. We compare the coverage of TypeOracle (which relies on execution traces to infer parameter types) and PDFuzzer when considering specification inference only for undocumented APIs. To isolate the effect of specification inference, we disabled relationship inference in both approaches. PDFuzzer achieves 28% and 20% higher coverage than TypeOracle on Adobe Acrobat Reader and Foxit PDF Reader, respectively. For XChange Editor, where the gain is limited (1%), our reverse-engineering showed that it does not implement any undocumented APIs, including those found in Adobe.

Dimension 2: Grammar Granularity. We run PDFuzzer using function-level grammars versus parameter-level grammars and compare the coverage. Parameter-level grammars achieve 5–18% higher coverage, as the LLM captures more detail when focusing on parameter descriptions rather than processing them at once.

Dimension 3: Relationship Tiers and Semantic Types.

We compare coverage across five relationship-modeling settings: *None* (no inter-API relationships), *Candidate* (candidate co-occurrence only), *PC-Only* (PDFuzzer configured to use only producer-consumer strong symbolic relationships), *Favocado* [16] (a fuzzer included as an external producer-consumer baseline), and *Strong* (all

three semantic types of strong symbolic relationships, i.e., producer-consumer, value-constraint, and implicit). As shown in the Relation Tier rows of Table 4, stronger relationship modeling consistently improves coverage. PC-Only performs close to Favocado, confirming that our PC-Only configuration replicates Favocado-style producer-consumer matching. In contrast, Strong outperforms both PC-Only and Favocado on all targets, with a maximum gain of 15.5% on Foxit PDF Reader. This confirms that value-constraint and implicit relationships contribute beyond producer-consumer relationships.

Dimension 4: PDF objects generation integration. We evaluate the impact of integrating PDF objects generation with JavaScript API calls by comparing PDFuzzer-js (JavaScript only) against PDFuzzer-full (with PDF objects integration via Cooper [63]). PDFuzzer-full achieves 3–8% higher coverage, validating the effectiveness of Stage 3 in Test Case Generation, where Cooper's cooperative mutation approach leverages relationships between PDF native objects and JavaScript APIs. Since many vulnerabilities depend on specific PDF document structures alongside JavaScript code, PDFuzzer-full ensures that generated documents include the native structures necessary to support JavaScript operations.

LLM Influence. Our earlier experiments used GPT-4o (OpenAI, May 2024). Here, we additionally evaluate Claude-3.7-Sonnet (Anthropic, February 2025) and o3-mini (OpenAI, February 2025, optimized for reasoning tasks), focusing our LLM ablation on Adobe Acrobat Reader. We found that the choice of LLM has minimal impact on PDFuzzer. Comparing test coverage with o3-mini, Claude-3.7-Sonnet, and GPT-4o, the differences remain within 10,000 basic blocks, less than 1% of the total in Adobe Acrobat Reader. Thus, using newer LLMs beyond GPT-4o does not substantially affect the overall performance or stability of PDFuzzer.

8 Generalization Beyond PDF Readers

While our evaluation primarily focuses on JavaScript engines in PDF readers, the fundamental methodology underlying PDFuzzer is not limited to this specific domain. PDFuzzer takes API documentation as input, extracts machine-readable specifications, and then derives grammars and cross-API constraints for generating high-quality test inputs. Since rich API documentation is widely available across software ecosystems, the same pipeline can be applied beyond JavaScript engines in PDF readers.

To validate this generalization in a different setting, we applied our specification extraction and grammar construction pipeline to Microsoft Word's VBA (Visual Basic for Applications) APIs. VBA is the macro language for automating Microsoft Office applications (e.g., Word, Excel, PowerPoint), which are widely deployed, closed-source Windows programs with large and distinct API surfaces. From the official documentation (5,920 pages), we extracted specifications for 2,925 APIs covering 3,768 parameters, and generated context-free grammars for all of them. We observed that the three relationship types defined in Section 3 (producer–consumer, value-constraint, and implicit/state-based relationships) also occur frequently in VBA APIs. This suggests that our relationship modeling generalizes beyond JavaScript in PDF readers.

Given these grammars and inferred relationships, the next step is straightforward: generate VBA macro test cases and embed them into Word documents for fuzzing—analogous to how PDFuzzer embeds JavaScript into PDF files. This feasibility study suggests that PDFuzzer can be readily adapted to other documentation-rich domains with large API surfaces.

9 Related Work

API Security. Existing work on API security involves vulnerability detection through static analysis or dynamic testing, as well as methods to infer relationships and dependencies among APIs. These techniques have been applied to various targets, such as RESTful APIs [9, 20, 36], the Linux kernel [11–13, 19, 26, 28, 29, 47, 50, 51, 62], JavaScript engines [16, 24, 63], open-source projects [23, 33], smart contracts [43], microservices [25], software libraries [21, 32, 34, 42, 67], and deep learning libraries [61]. Static methods like SyzDescribe [28] and GraphFuzz [21] extract signatures from source code, while dynamic approaches use execution traces [24] or symbolic execution [11, 12]. However, closed-source PDF readers preclude direct application of these techniques, and their natural-language API manuals resist automated extraction.

For API relationship inference, static approaches like RESTler [9] identify producer-consumer dependencies through type matching, while dynamic methods like HEALER [51] and APIGraph [34] observe execution feedback. Recent work also applied neural networks for sequence prediction [23, 33, 61, 62]. Our approach differs by using LLMs to generate API rules and infer relationships, reducing manual effort and improving scalability. Unlike prior work that mostly relies on name-based matching, we model richer interactions, including value constraints and implicit dependencies.

LLM-based Fuzzing. Prior work has explored using LLMs to facilitate other fuzzing targets, such as protocols [38, 39, 58], the Linux kernel [64], interpreters [60], deep learning libraries [14, 15], command-line applications [55], and open-source libraries [37]. They use LLMs to generate or mutate testing cases. For example, LLMs have been used to produce initial seeds [8, 14, 39] or concrete test cases [15, 49], generate scripts for test case creation [66], generate fuzz drivers [37], summarize documentation [58, 60], and analyze source code to derive specifications [37, 64]. In mutation, they can operate at either the byte level [65], the token level [14, 18], or the whole input level [60]. Our experiments show that using LLMs to directly generate test cases for PDF reader fuzzing is slow and costly. Moreover, the closed-source nature of PDF readers makes LLM-driven specifications extraction extremely challenging. Instead, we employ LLMs to analyze API manuals, generate API invocation rules, and infer intricate API relationships.

Grammar-based Fuzzing. Grammar-based fuzzing uses context-free grammars (CFGs) to generate syntactically valid inputs for structured data, targeting syntactic validity [30, 56], semantic validity [27, 31], and behavioral exploration [22]. Existing methods use different strategies: code fragment reuse (LangFuzz [31], CodeAlchemist [27]), custom IRs (DUMPLING [54]), or direct CFGs [5, 7, 30]. Mutation approaches include grammar-aware [7, 31, 56], token-based [48, 56], and IR-guided mutations [22, 57]. However, traditional grammar-based approaches embed only limited semantic constraints within grammar rules, failing to capture complex API relationships, including those defined in Section 3. Our approach extends beyond syntactic generation by explicitly modeling these semantic constraints between API parameters and return values,

then dynamically solving them to generate realistic, security-critical API usage sequences that traditional methods cannot achieve.

10 Conclusion

Fuzzing JavaScript engines of PDF readers faces critical challenges, including incomplete API specifications and limited API relationships inferred by previous works. In this paper, we propose PDFuzzer, an LLM-assisted fuzzing tool to generate API function invocation rules based on documentation to improve fuzzing performance on the PDF reader's JavaScript engine. On three widely used PDF readers (Adobe Acrobat Reader, Foxit PDF Reader, and PDF-XChange Editor), we show that PDFuzzer improves fuzzing effectiveness over prior tools. It achieves higher code coverage and uncovers more zero-day vulnerabilities. Specifically, while current tools identify only 0–6 such vulnerabilities, PDFuzzer successfully uncovered 31 zero-day vulnerabilities.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive comments. This material is based in part upon work supported by DARPA under agreement N66001-22-2-4037. It is also supported by the National Science Foundation under grant no. 2229876 and is supported in part by funds provided by the Department of Homeland Security and by IBM. Yigitcan Kaya was supported by an appointment to the Intelligence Community Postdoctoral Research Fellowship Program at the National Institute of Standards and Technology administered by Oak Ridge Institute for Science and Education (ORISE) through an interagency agreement between the U.S. Department of Energy and the Office of the Director of National Intelligence (ODNI).

References

[1] 2008. CVE-2008-2992. https://www.coresecurity.com/core-labs/advisories/adobe-reader-buffer-overflow/.
[2] 2017. CVE-2017-10951. https://www.zerodayinitiative.com/blog/2017/8/17/busting-myths-in-foxit-reader/.
[3] Adobe. 2025. Doc and Doc.Media APIs — Acrobat-PDFL SDK: JavaScript Reference.
[4] Adobe. 2025. JavaScript APIs — Acrobat-PDFL SDK: JavaScript Reference.
[5] José Antonio Zamudio Amaya. 2024. Shaping Test Inputs in Grammar-Based Fuzzing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, Maria Christakis and Michael Pradel (Eds.). ACM, 1901–1905.
[6] Anthropic. 2025. Anthropic Claude 3.7 Sonnet Model.
[7] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS*.
[8] Asmita, Yaroslav Oliinyk, Michael Scott, Ryan Tsang, Chongzhou Fang, and Houman Homayoun. 2024. Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing. In *33rd USENIX Security Symposium, USENIX Security*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association.
[9] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: stateful REST API fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 748–758.
[10] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2025. *The SMT-LIB Standard: Version 2.7.* Technical Report. Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.
[11] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Lee Schales, Jiyong Jang, and Zhiyun Qian. 2024. SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In *IEEE Symposium on Security and Privacy, SP*. IEEE, 4661–4677.
[12] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. 2021. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 749–763.
[13] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2123–2138.
[14] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, René Just and Gordon Fraser (Eds.). ACM, 423–435.
[15] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE*. ACM, 70:1–70:13.
[16] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *28th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.
[17] DynamoRIO. 2025. DynamoRIO. https://dynamorio.org/.
[18] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, Maria Christakis and Michael Pradel (Eds.). ACM, 1656–1668.
[19] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. 2023. ACTOR: Action-Guided Kernel Fuzzing. In *32nd USENIX Security Symposium, USENIX Security*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 5003–5020.
[20] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 725–736.
[21] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE*. ACM, 1070–1081.
[22] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *30th Annual Network and Distributed System Security Symposium, NDSS*.
[23] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 631–642.
[24] Suyue Guo, Xinyu Wan, Wei You, Bin Liang, Wenchang Shi, Yiwei Zhang, Jianjun Huang, and Jian Zhang. 2023. Operand-Variation-Oriented Differential Analysis for Fuzzing Binding Calls in PDF Readers. In *45th IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 95–107.
[25] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1387–1397.
[26] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-based Fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2345–2358.
[27] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *26th Annual Network and Distributed System Security Symposium, NDSS*.
[28] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. 2023. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers. In *44th IEEE Symposium on Security and Privacy, SP*. IEEE, 3262–3278.
[29] Yu Hao, Hang Zhang, Guoren Li, Xingyun Du, Zhiyun Qian, and Ardalan Amiri Sani. 2022. Demystifying the Dependency Challenge in Kernel Fuzzing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE*. ACM, 659–671.
[30] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE*, Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir (Eds.).
[31] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium*, Tadayoshi Kohno (Ed.).

[32] Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium, USENIX Security*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2271–2287.
[33] Tien-Duy B. Le and David Lo. 2018. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, Frank Tip and Eric Bodden (Eds.). ACM, 106–117.
[34] Jiayi Lin, Qingyu Zhang, Junzhe Li, Chenxin Sun, Hao Zhou, Changhua Luo, and Chenxiong Qian. 2025. Automatic Library Fuzzing through API Relation Evolvement. In *32nd Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.
[35] Yuwei Liu, Yanhao Wang, Xiangkun Jia, Zheng Zhang, and Purui Su. 2024. AFGen: Whole-Function Fuzzing for Applications and Libraries. In *IEEE Symposium on Security and Privacy, SP*. IEEE, 1901–1919.
[36] Chenyang Lyu, Jiacheng Xu, Shouling Ji, Xuhong Zhang, Qinying Wang, Binbin Zhao, Gaoning Pan, Wei Cao, Peng Cheng, and Raheem Beyah. 2023. MINER: A Hybrid Data-Driven Approach for REST API Fuzzing. In *32nd USENIX Security Symposium, USENIX Security*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 4517–4534.
[37] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Association for Computing Machinery, 3793–3807.
[38] Xiaoyue Ma, Lannan Luo, and Qiang Zeng. 2024. From One Thousand Pages of Specification to Unveiling Hidden Bugs: Large Language Model Assisted Fuzzing of Matter IoT Devices. In *33rd USENIX Security Symposium, USENIX Security*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association.
[39] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model guided Protocol Fuzzing. In *31st Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.
[40] Microsoft. 2025. Windows Error Reporting. https://learn.microsoft.com/en-us/windows/win32/wer/windows-error-reporting.
[41] Microsoft Research. 2025. Z3 Solver.
[42] Hoan Anh Nguyen, Hung Dang Phan, Syeda Khairunnesa Samantha, Son Nguyen, Aashish Yadavally, Shaohua Wang, Hridesh Rajan, and Tien N. Nguyen. 2022. A Hybrid Approach for Inference between Behavioral Exception API Documentation and Implementations, and Its Applications. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE*. ACM, 2:1–2:13.
[43] Mitchell Olsthoorn, Dimitri Michel Stallenberg, Arie van Deursen, and Annibale Panichella. 2022. SynTest-Solidity: Automated Test Case Generation and Fuzzing for Smart Contracts. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion*. ACM/IEEE, 202–206.
[44] OpenAI. 2025. File search. https://platform.openai.com/docs/guides/tools-file-search.
[45] OpenAI. 2025. OpenAI GPT-4o Model.
[46] OpenAI. 2025. OpenAI o3-mini Model.
[47] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium, USENIX Security*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 729–743.
[48] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. 2021. Token-Level Fuzzing. In *30th USENIX Security Symposium, USENIX Security*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2795–2809.
[49] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Trans. Software Eng.* 50, 1 (2024), 85–105.
[50] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 351–366.
[51] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 344–358.
[52] The MITRE Corporation. 2026. CVE-2024-28888. https://www.cve.org/CVERecord?id=CVE-2024-28888/.
[53] The MITRE Corporation. 2026. CVE-2024-34099. https://www.cve.org/CVERecord?id=CVE-2024-34099/.
[54] Liam Wachter, Julian Gremminger, Christian Wressnegger, Mathias Payer, and Flavio Toffalini. 2025. DUMPLING: Fine-grained Differential JavaScript Engine Fuzzing. In *32nd Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.
[55] Dawei Wang, Geng Zhou, Li Chen, Dan Li, and Yukai Miao. 2024. ProphetFuzz: Fully Automated Prediction and Fuzzing of High-Risk Option Combinations with Only Documentation via Large Language Model. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM, 735–749.
[56] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 724–735.
[57] Jiming Wang, Yan Kang, Chenggang Wu, Yuhao Hu, Yue Sun, Jikai Ren, Yuanming Lai, Mengyao Xie, Charles Zhang, Tao Li, and Zhe Wang. 2024. OptFuzz: Optimization Path Guided Fuzzing for JavaScript JIT Compilers. In *33rd USENIX Security Symposium, USENIX Security*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association.
[58] Jincheng Wang, Le Yu, and Xiapu Luo. 2024. LLMIF: Augmented Large Language Model for Fuzzing IoT Devices. In *IEEE Symposium on Security and Privacy, SP*. IEEE, 881–896.
[59] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages.
[60] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE*. ACM, 126:1–126:13.
[61] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. 2022. DocTer: documentation-guided fuzzing for testing deep learning API functions. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 176–188.
[62] Jiacheng Xu, Xuhong Zhang, Shouling Ji, Yuan Tian, Binbin Zhao, Qinying Wang, Peng Cheng, and Jiming Chen. 2024. MOCK: Optimizing Kernel Fuzzing Mutation with Context-aware Dependency. In *31st Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.
[63] Peng Xu, Yanhao Wang, Hong Hu, and Purui Su. 2022. COOPER: Testing the Binding Code of Scripting Languages with Cooperative Mutation. In *29th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.
[64] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25)*. Association for Computing Machinery.
[65] Hongxiang Zhang, Yuyang Rong, Yifeng He, and Hao Chen. 2024. LLAMAFUZZ: Large Language Model Enhanced Greybox Fuzzing. *CoRR* abs/2406.07714 (2024). arXiv:2406.07714
[66] Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. 2025. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. *CoRR* abs/2501.19282 (2025). arXiv:2501.19282
[67] Hao Zhong, Na Meng, Zexuan Li, and Li Jia. 2020. An empirical study on API parameter rules. In *ICSE '20: 42nd International Conference on Software Engineering*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 899–911.

A Ethical Considerations

We carefully considered the ethical implications of conducting and publishing this research. While vulnerability discovery may create risks through premature disclosure or potential misuse, it also provides substantial benefits by enabling vendors to remediate serious security flaws and protecting PDF reader users from future exploitation. To mitigate these risks, we promptly reported all 31 discovered vulnerabilities to the affected vendors through coordinated disclosure, allowed sufficient time for validation and patch development, and confirmed their disclosure status before publication. We also avoid providing unnecessary exploitation details. Based on these safeguards, we believe that the security benefits and scientific value of this work outweigh its potential harms.

B Open Science

Our artifact can be found at https://github.com/ucsb-seclab/PDFuzzer. Due to the page limit, the full version of this paper is available on https://arxiv.org/abs/2608.06641.