

SoK: What Are We Really Fuzzing? A Systematization of Scripting Language Fuzzing

Chibin Zhang, Qinying Wang†, and Mathias Payer
EPFL
chibin.zhang@epfl.ch qinying.wang@epfl.ch mathias.payer@nebelwelt.net

***Abstract*—Scripting language engines for JavaScript, Python, Ruby, or PHP are essential for web browsers, cloud services, and security-sensitive applications, yet their correctness is difficult to assure: dynamic typing, multi-tier JIT compilation based on speculative type assumptions, garbage collection, and standard libraries spanning hundreds of thousands of lines of code combine to produce a behavior space no test suite can exhaustively cover. Fuzzing has emerged as the most effective technique for finding bugs in these engines, and over the past 15 years researchers have published 61 papers on the topic. Despite this volume of work, the field remains fragmented: each paper introduces its unique terminology, evaluates on different targets with different metrics, and leaves critical questions unanswered about which components are vulnerable, which techniques actually work, and how progress should be measured. Our goal is to tame this unnecessary complexity by introducing the first systematization of knowledge for scripting language fuzzing. We define a four-component engine model, formalize two distinct threat models, and conduct a large-scale empirical study of 156,595 bug reports across V8, CPython, PHP, and Ruby. Our analysis shows that bug profiles differ dramatically by engine architecture: V8 is dominated by JIT optimization and memory management bugs, while interpreted languages concentrate bugs in standard libraries. We further identify three component-specific syntactic idioms (Standard library, Optimization, Callback) shared by bug-triggering inputs across all four engines, which any component-targeted fuzzer must encode to reach the corresponding bugs. We unify the fragmented mutation terminology, showing that despite distinct names across the 51 surveyed fuzzers, nearly all are captured by two fundamental operations (recursive descent generation and context-matching splicing) plus a value-mutation layer. We also reveal that preprocessing dominates fuzzer implementation footprint (around 65% of LoC on average) and is the dominant driver of coverage: augmenting Nautilus with builtin specifications and JIT-triggering patterns more than doubles its V8 coverage (60K to 121K edges), while ablating Fuzzilli's knowledge drops V8 coverage by 32% (155K to 106K edges). Yet this critical work is routinely dismissed in a single paragraph. Finally, we demonstrate empirically that aggregate edge coverage, taken alone, is an incomplete metric for language fuzzing. Two fuzzers achieving similar aggregate coverage can explore entirely disjoint engine components. We propose per-component coverage as a complementary metric. We release a standardized benchmark and open dataset to enable fair comparison in future work.**

†Corresponding author

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.230757
www.ndss-symposium.org

I. Introduction

Scripting languages permeate all layers of the software stack. These languages enable rapid prototyping, seamless system integration, and automation at scale, making them indispensable to both everyday users and critical infrastructure. Their flexibility and approachability have made them ubiquitous: they empower web applications, cloud services, data science frameworks, and increasingly, security-sensitive applications like browser engines, game engines, document processors, and LLM agent frameworks.

This ubiquity comes with serious security implications due to vulnerabilities in scripting language engines. Memory corruption vulnerabilities in language runtimes enable attackers to escape sandboxes, bypass security boundaries, and compromise entire systems. A single bug in V8 (Chrome's JavaScript engine) affects billions of users worldwide. In 2024 alone, Google paid $3.4 million in bug bounties for 337 Chrome vulnerabilities [1], and Chrome represents just one of many JavaScript engine implementations [2], [3], [4], [5], alongside interpreters for Python [6], PHP [7], and Ruby [8]. Yet despite how important it is to secure these engines, our understanding of how to systematically test them remains fragmented.

Fuzzing has emerged as the most effective technique for finding bugs in scripting language engines. Google's Fuzzilli, Mozilla's jsfunfuzz, and academic fuzzers like CodeAlchemist [9], [10], [11], [12] and DIE regularly uncover critical vulnerabilities. But fuzzing scripting languages presents unique challenges. While the input format for file parsers and network protocols is straightforward, language fuzzers must generate syntactically valid programs, and preferably semantically correct ones. Unlike compiler fuzzing, which focuses on miscompilations in optimization passes, scripting language fuzzing must contend with dynamic typing, complex runtime state, garbage collection, JIT compilation based on speculative type assumptions, and massive standard libraries with intricate APIs.

The Problem: A Fragmented Research Landscape. Over the past 15 years, at least 61 papers have been published on fuzzing interpreters and scripting languages across top security, programming languages, and systems venues. Yet despite this extensive body of work, the field remains opaque. Each paper introduces its own terminology, claims generality (sometimes without considering impact across fields), and evaluates effectiveness on non-overlapping case studies:

Unclear threat models. Existing work does not clearly distinguish between two fundamentally different attack scenarios: (1) adversaries who control the script being executed (e.g., malicious JavaScript in a browser), and (2) adversaries who control only the input data to a trusted script (e.g., as with deserialization vulnerabilities). The literature applies the more serious threat model uniformly across all engines, reporting crashes reachable only from adversarial scripts as vulnerabilities even where the language's deployment threat model excludes attacker-controlled scripts. This overstates impact for server-side engines like CPython, PHP, and Ruby and obscures which fuzzing results translate to real-world security.

Unclear bug characterization. While papers present case studies of discovered bugs, we lack a systematic understanding of why bugs occur in each component and what bug-triggering inputs look like. Do JIT bugs require loops? Do type system bugs need specific object patterns? What syntactic features correlate with bugs in different components? Without this knowledge, we cannot design targeted fuzzing strategies.

Confusing Terminology. A few mutation techniques are frequently "rediscovered" under different names. What one paper calls "aspect-preserving mutation," another calls "semantics-aware code assembly," and yet another calls "ingredient injection." At an abstract level, these techniques boil down to recursive code structure expansion and context-matching code splicing, but this commonality is obscured by terminology coined by respective papers to claim novelty.

Hidden implementation complexity. Research papers emphasize novel mutation algorithms, but our ablation experiments show that preprocessing quality, not mutation novelty, drives coverage: augmenting Nautilus with builtin type signatures and JIT-triggering patterns more than doubles its V8 edge coverage (60K to 121K), while removing Fuzzilli's 4,000-line JavaScript builtin specifications and its JIT-triggering templates drops V8 coverage by 32% (155K to 106K). Yet this critical preprocessing work (specification & seed corpus collection, code normalization, and profiling), which accounts for around 65% of implementation footprint on average, is at best dismissed in a single paragraph, making reproduction difficult and obscuring what actually makes language fuzzers effective.

Non-standardized evaluation. To evaluate fuzzers for general-purpose software, the fuzzing community has developed benchmarks like FuzzBench [13], LAVA-M [14], MAGMA [15], but scripting language fuzzing has none. Every paper evaluates on different targets, with different build and runtime configurations, different seed corpora, and different specifications, making comparison challenging. Most papers claim to target "engines", "runtimes", or "interpreters" generally, but they effectively explore only a few specific components, often just the JIT compiler or parser. We lack a clear understanding of how techniques map to distinct components: frontend (parser), virtual machine, type system, JIT optimizer, garbage collector, and standard library. Even when papers measure code coverage, aggregate coverage alone is an incomplete signal: JavaScript fuzzers typically achieve $\sim$15% edge coverage, with 9% in standard library code, leaving just 6% for the core engine. Aggregate coverage obscures which components are actually being tested, and fuzzers with similar overall coverage may excel at completely different parts of the engine.

As a result, despite a decade of accumulated work, the field has three blind spots. It cannot reliably attribute reported gains to novel mutation algorithms rather than to the expert knowledge (specifications, seed corpora, bug-triggering idioms) that our ablations show drives coverage gains. It cannot tell whether two fuzzers with similar aggregate coverage exercise the same engine components or near-disjoint ones. And it continues to re-coin terminology for the same two underlying operations. We substantiate each of these claims through ablation experiments and per-component coverage measurements on V8, CPython, PHP, and Ruby. Throughout, PHP and Ruby refer to their reference implementations (the Zend Engine-based PHP runtime and CRuby/MRI, respectively).

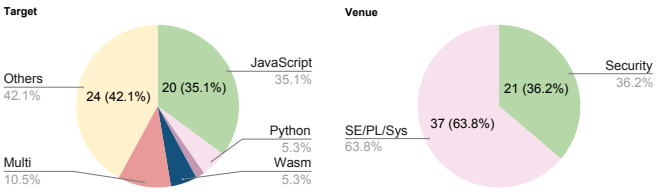
This paper provides the first comprehensive systematization of knowledge for scripting language fuzzing, organized around three questions: (Q1) *where do bugs live, and what inputs trigger them?* (section III, section IV); (Q2) *how do fuzzers reach those bugs, and where does the implementation footprint actually go?* (section V); and (Q3) *how should fuzzer effectiveness be measured?* (subsection V-D). We answer them by analyzing 61 papers published from 2011 to 2025, collecting and characterizing over 156K real-world bugs across V8, CPython, PHP, and Ruby, and validating our claims through preprocessing-ablation and per-component coverage experiments on all four engines, packaged as a reproducible benchmark. The campaign also surfaced three previously unknown bugs in PHP.

Our contributions are:

- **Component-based architecture and bug characterization.** We define the four core components of scripting language engines (Frontend, Core – Code, Core – Data, Standard library), clarify their interactions, and formalize two distinct threat models that map to different security boundaries. Building on this architecture, we systematically analyze 156,595 bug reports and 4,803 CVEs to map bugs to components, identify three component-specific syntactic idioms (Standard library, Optimization, Callback) shared across all four engines, and characterize temporal trends including new bug-class introductions tied to architectural changes such as JIT and generational GC.
- **Technique systematization.** We unify fragmented mutation terminology across the 51 surveyed fuzzers, and propose an abstract model of program mutations they all share. We also reveal that preprocessing accounts for around 65% of implementation footprint on average despite being underreported.
- **Evaluation metric review and recommendations.** We show that aggregate edge coverage alone is an incomplete measure of fuzzer effectiveness: across V8, CPython, PHP, and Ruby, the best surveyed fuzzers leave a 40-percentage-point gap below each engine's test suite, and aggregate scores can hide near-disjoint subsystem coverage driven

TABLE I: Top-tier venues included in the literature survey.

Area	Venues
Software Engineering	ICSE, ISSTA, FSE, ASE, PLDI
Cyber Security	USENIX Sec, S&P, CCS, NDSS
Computer Systems	OSDI, SOSP, USENIX ATC, EuroSys



(a) Targets fuzzed (b) Publication venues

Fig. 1: Distribution of surveyed papers by target and venue.

by preprocessing quality. We propose per-component coverage and a three-category taxonomy of coverage barriers (configuration, API knowledge, environment).

- **Benchmark and future directions.** We provide the first standardized benchmark for scripting language fuzzers, comprising (i) the labeled bug-report dataset, (ii) the extraction pipeline (prompts, schemas, crawler, and the validation data of section IV), and (iii) the build, run, and coverage-measurement scripts with the target configurations and evaluation parameters of subsection V-D. We identify key open problems including preprocessing automation and component-targeted strategies, and will release the benchmark as an open-source artifact upon acceptance [16].

II. Methodology

We first explain literature-corpus construction, then how it and real-world bug data structure the analysis.

Survey methodology. We surveyed papers published between 2011 and 2025 at top-tier venues in software engineering, security, and systems (Table I), selecting venues ranked A* or A by CORE [17]. We searched DBLP using a query combining fuzzing/testing terms with scripting language targets, then manually applied inclusion criteria (novel technique or analysis for a scripting language engine or subsystem) and snowballed citations forward and backward until no new papers qualified. This yielded a final corpus of 61 papers. The search query, inclusion/exclusion criteria, and snowballing procedure appear in subsection H. Figure 1 summarizes the distribution by target language and venue. Our corpus also includes fuzzers for the JVM, C/C++ compilers, WebAssembly, and Vulkan shader compilers. We include them because they introduce techniques that transfer directly to scripting-language engines: V8 and JVM fuzzers both generate hot loops to trigger JIT optimization, and Artemis, for example, was originally developed for JVM implementations before being extended to JavaScript [18]. We call this scope "scripting language engine/interpreter fuzzing"; Table XVI annotates each entry's target domain.

Paper organization. section III defines the four engine components and two threat models used throughout the paper. section IV characterizes 156,595 real-world bugs, showing

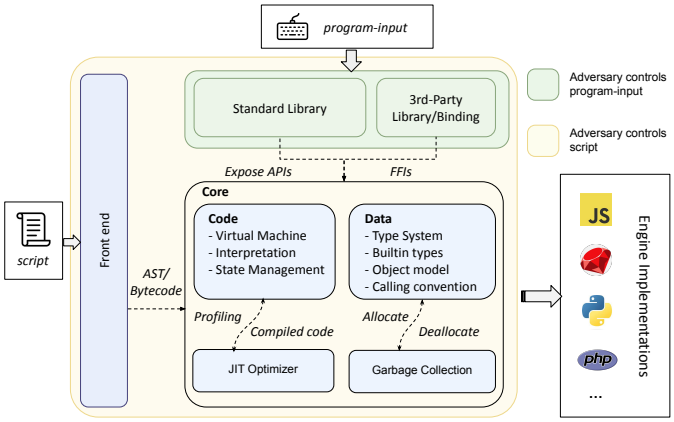


Fig. 2: Typical component model and threat models for scripting-language engines. Shaded regions indicate the two threat models considered in this work: adversary-controlled scripts and adversary-controlled program inputs.

that bug profiles differ sharply across engines and components. section V systematizes 61 fuzzing papers: the surveyed generation and mutation techniques are built around two operations, preprocessing dominates the implementation footprint, semantic correctness is the central generation challenge, and aggregate edge coverage alone is an incomplete metric.

III. A Unified Model of Scripting Language Engine

We answer Q1 (where bugs live) by modeling every scripting-language engine through a four-component architecture and two threat models. We first characterize per-component complexity, then use the threat models to determine reachability, fuzzing implications, and security impact.

A. Component Model and Characterization

As shown in Figure 2, we distill a unified four-component model that captures the common architecture of scripting-language engines: **Frontend**, **Core – Code**, **Core – Data**, and **Standard library**. This decomposition follows compiler textbooks [19] and every major language specification (ECMAScript [20], CPython Language Reference [21], PHP internals [22]), which all separate lexical/syntactic processing (frontend) from the semantic core operating over code and a data/object model, and from the surrounding standard library and host interface.

Frontend (Parser). The frontend translates source text into an internal program representation, such as an AST, bytecode, or intermediate representation. It includes lexical analysis, parsing, syntax validation, and early semantic checks. Parser bugs typically cause malformed-syntax crashes but can also enable code execution through memory corruption during AST construction.

Core – Code. The VM executes bytecode instructions and manages the call stack, local variables, and execution context. For JavaScript, this includes managing execution contexts, scope chains, and the prototype inheritance system. Engines like V8's Ignition, SpiderMonkey's baseline interpreter, and JavaScriptCore's LLInt interpret bytecode before JIT compilation. VM bugs often involve incorrect instruction implementation, stack corruption, or improper handling of exceptions.

JIT Compilation. Just-In-Time compilers optimize hot code paths by compiling bytecode to native machine code. Modern engines employ multi-tier compilation: V8 uses Sparkplug (baseline), Maglev (mid-tier), and TurboFan (optimizing); JavaScriptCore has LLInt, Baseline, DFG, and FTL; SpiderMonkey features Baseline and IonMonkey. JIT compilers perform speculative optimizations based on observed types, with deoptimization bailouts when assumptions fail. JIT bugs are among the most common and exploitable, arising from incorrect optimizations, improper bounds check elimination, flawed type inference, or interaction with garbage collection during compilation.

Core – Data. *Value representations and type checking.* Scripting languages use dynamic typing where types are determined at runtime. Engines implement object models with hidden classes (V8's Maps, SpiderMonkey's Shapes) to optimize property access. Type system bugs arise from incorrect type transitions, improper handling of prototype chains, or race conditions in concurrent type updates. These bugs can lead to type confusion primitives useful for exploitation.

Object Lifecycle / Memory Management. Memory management in scripting languages is automatic through garbage collection, or reference counting. Typically, engines first build an abstraction layer over the raw libc allocator interface: (malloc/realloc/free). Engines that use reference counting cannot count on it entirely as there are cyclical references, and fall back to garbage collection when necessary. Engines use generational collectors with multiple heaps (young generation, old generation) and concurrent/incremental marking to reduce pause times. GC bugs manifest as use-after-free vulnerabilities when objects are prematurely collected, or memory corruption when GC interacts incorrectly with JIT-compiled code or concurrent execution.

Standard library. The standard library provides built-in objects, functions, and APIs to scripts. For browser engines, this includes DOM bindings, WebAssembly support, and extensive Web APIs. Standard library bugs arise from incorrect API implementations, improper argument validation, or complex interactions between API calls. Browser-specific bugs often involve the binding layer between JavaScript and native C++ code, where type mismatches and lifetime issues can occur.

Component characterization. We further characterize the complexity across components for CPython, PHP, Ruby, and V8. We manually labeled each file and directory in the source tree by component, then used tokei to measure lines of code (LoC), aggregating the results by component. Table II presents the breakdown across 2–3 million lines of code per engine. We have the following findings. *First, frontend parsers represent minimal implementation footprint*, around 1% of total codebase size across engines; Ruby is an outlier at 4.0% because it maintains three parsers (Ripper, Prism, and the original Bison/Flex parser). *Second, optimization investment varies dramatically by language.* V8 dedicates 372,715 LoC to JIT optimization (59.5% of Core – Code, 13.1% of its codebase), PHP and Ruby invest moderately (5.5% and 4.1% respectively), and CPython, with only an experimental JIT as of 3.13, dedicates just 1,557 LoC (0.06%). *Third, standard libraries dominate the implementation footprint*, accounting for 26.7–48.6% of codebase size for CPython, Ruby, and PHP, far more than the core VM. V8's 11.3% is artificially low because it delegates filesystem, networking, and threading APIs to its embedding environments (browser WebAPIs, Node.js).

B. Threat Models Over The Component Model

Figure 2 distinguishes two realistic threat models: adversarial script text or adversarial data consumed by a trusted script.

Threat Model 1 (TM1): Adversary controls the script. The attacker supplies the program text executed by the engine. This model captures deployments that intentionally execute untrusted code, such as web pages running JavaScript, documents with embedded scripts, and plugin or extension systems that accept user-provided scripts. The engine itself is the security boundary: attacker-controlled scripts can exercise the full language surface and potentially reach all four components. Typical high-impact bugs include type confusion and arbitrary read/write primitives arising from optimization, type-system, and memory-management logic. CVE-2014-3176 [23] (Figure 6a in subsection A) illustrates this threat model: an attacker-written JavaScript program triggers a V8 `Array.prototype.concat` heap corruption and then uses subsequent script execution to shape the corrupted state.

Threat Model 2 (TM2): Adversary controls input to script but not code itself. Here, the script executed by the engine is trusted, but the data consumed by that script is attacker-controlled. This model captures deployments where trusted applications process external inputs, such as server-side scripts parsing HTTP requests, deserializing JSON or MessagePack data, rendering templates with user-controlled fields, or embedded Lua scripts processing network packets. The security boundary is the input-processing logic exposed by the trusted script: the attacker may supply crafted data, but cannot choose arbitrary program behavior. Typical high-impact bugs include code execution, denial of service, and information disclosure through parsers, deserializers, template engines, regular-expression engines, or standard-library APIs. CVE-2010-0397 [24] (Figure 6b in subsection A) illustrates this threat model: a trusted PHP script calls `xmlrpc_decode_request` on an attacker-supplied XML-RPC request, and a malformed request crashes the worker. The script is trusted. Only the request body is adversarial.

Fuzzing implications. The two threat models lead to different fuzzing inputs and different interpretations of crashes. First, the appropriate fuzzing input differs. Under TM1, fuzzers generate programs because the adversary controls the script itself. This can exercise the full language surface and potentially reach all engine components. Under TM2, fuzzers should generate external data consumed by trusted scripts, such as JSON, XML, MessagePack, template inputs, regular-expression inputs, or network packets. Only bugs reachable through these exposed input-processing paths are deployment-relevant. Thus, TM1 strictly subsumes TM2. Second, the security interpretation of crashes differs. Program-generating

TABLE II: Codebase composition (LoC %), bug distribution (Bugs %), and bug density (B/KL = bugs per 1,000 LoC) by top-level component. *LoC* columns label each file by component and aggregate with tokei. *B/KL* divides the raw set-union bug count for each component by that component's LoC and multiplies by 1,000. The *Test & Tooling* row applies only to LoC, and unclassified reports are excluded, so its B/KL is undefined. The *Total* row's B/KL uses engine-code LoC (excluding Test & Tooling) as the denominator. Bold marks the largest LoC % and Bugs % per engine.

	V8			**CPython**			**PHP**			**Ruby**		
Component	LoC	Bugs	B/KL	LoC	Bugs	B/KL	LoC	Bugs	B/KL	LoC	Bugs	B/KL
Frontend	1.1%	4.6%	15.4	1.8%	2.9%	34.2	0.9%	2.9%	62.8	4.0%	7.1%	11.2
Core – Code	22.0%	**41.5%**	6.9	5.4%	12.2%	47.6	14.8%	11.4%	15.0	10.4%	19.0%	11.6
Core – Data	8.3%	29.9%	13.2	6.2%	8.1%	27.5	2.2%	12.7%	112.8	6.8%	12.6%	11.7
Stdlib / Ext	11.3%	24.1%	7.8	**48.6%**	**76.8%**	33.3	33.5%	**73.0%**	42.6	26.7%	**61.4%**	14.5
Test & Tooling	**57.2%**	—	—	38.0%	—	—	**48.8%**	—	—	**52.1%**	—	—
Total	2,842K	7,503	6.18	2,679K	50,915	30.66	1,953K	33,302	33.17	1,629K	8,936	11.45

fuzzing aligns with JavaScript deployments, where browsers and client-side runtimes commonly execute untrusted scripts, so crashes reachable from adversarial JavaScript can have direct security impact. In typical CPython, PHP, and Ruby deployments, however, attackers usually control only external inputs. Crashes requiring adversarial scripts may be real engine bugs, but not necessarily deployment-relevant vulnerabilities. This distinction is consistent with language security policies and CVE practice, which exclude or downgrade obviously malicious code invocations and evaluate severity based on expected use [25], [26]. Independent evaluations of fuzzing methodology reach a similar conclusion, cautioning that CVE-based metrics are an unreliable proxy for the security impact of found bugs [27]. Exceptions arise in sandboxed code-execution settings, such as PHP shared hosting with `open_basedir`/`disable_functions` and Python sandboxes used by agentic systems [28], [29], [30], [31]. In these settings, non-JavaScript engines may also face TM1.

Despite these consequences, only one surveyed fuzzer, Dataflow Fusion [32], states its deployment threat model explicitly, distinguishing malicious inputs to trusted PHP applications from attacker-authored PHP scripts. The rest never specify whether they target TM1 or TM2, so the security relevance of their crashes must be inferred.

These boundaries are deliberately soft: a bug spanning several components is assigned to its dominant one, and hybrid deployments (a Node.js service executing third-party plugins while parsing untrusted JSON) face both threat models, the relevant one set by the input that reaches the crash.

Finding 1: The surveyed fuzzers are oriented almost entirely toward TM1: 45 of 51 generate full programs, including fuzzers for CPython and PHP, whose dominant deployments expose TM2. Because papers rarely state which threat model a crash is reachable under, the security impact of reported crashes is frequently under-evaluated.

IV. Bug Characterization and Analysis

Continuing Q1 (what inputs trigger them), bug profiles diverge across engines as sharply as engine architectures themselves, and within each component bug-triggering inputs cluster into a small set of distinctive syntactic signatures (*idioms*) that any fuzzer must encode to reach the corresponding bugs. We substantiate this through four analyses of 156,595 real-world bugs from V8, CPython, PHP, and Ruby: mapping bugs to components, analyzing the root causes of crashes, tracking their temporal evolution, and extracting the idioms shared by bug-triggering PoCs.

We collected 156,595 bug reports from the issue trackers of V8 (8,174), CPython (74,058), PHP (62,738), and Ruby (11,625), spanning from 1998 to 2025. Additionally, we crawled 4,803 CVEs from the National Vulnerability Database (NVD) and matched them to issue tracker entries via reference URLs, yielding 3,071 CVE-to-issue mappings. A detailed breakdown of security/crash/PoC counts per engine and the CVE analysis appears in Table VIII (subsection B).

Methodology. Manual analysis at this scale is infeasible: CPython alone has over 141,000 issues, and PHP has over 80,000. We devised an LLM-assisted pipeline to efficiently characterize all bug reports. The pipeline scrapes each issue tracker, applies a deterministic inclusion filter to drop pull requests and tracker-typed non-bugs, and passes each surviving report through a fixed DeepSeek-V3 prompt that extracts the primary component(s), security relevance, crash and root cause, and proof-of-concept code or URLs, assigning each field only when it is explicitly evidenced. The full pipeline (crawling, inclusion filter, prompt schema, and automatic PoC checks) is described in subsection C. As a robustness check, we re-labeled the bug report corpus with two independent models (Kimi-K2.7 and GLM-5.2). Crash and security labels are largely reproducible across models (87–99% agreement; Table XI). Component assignment is harder, but the model pairs still reach substantial agreement (Cohen's κ; Table XI), supporting the reliability of the component labels behind Table II. As a further check, three independent experts re-annotated a 1,000-report validation set (300 unique reports per expert plus a 100-report shared subset). Against the expert consensus, DeepSeek-V3 reaches an F1 of 0.819 on component assignment, 0.917 on crash, and 0.796 on security, with 0.861 PoC-extraction precision (Table X). On the 100-report shared subset, the experts disagreed on the component label for 11 reports: 9 were resolved by majority vote and 2 defaulted to the most experienced expert.

The LLM assigned a component label to 97,652 of the 156,595 reports (62.4%), and only where the report provided developer evidence (component tags, root-cause analysis, or

affected files). Otherwise it left the list empty. The remaining 37.6% therefore comprises non-bugs (feature requests, documentation, build or configuration issues, user errors, invalid reports, and third-party problems), reports with insufficient evidence for reliable assignment, and test/tooling infrastructure outside the four components. Analyses that require component assignment (Table II) use the classified subset; root-cause and temporal analyses use the full corpus. Table IX (subsection C) shows that the unclassified reports are not a random drop-out: they carry a 3–27× lower security rate and a 2–8× lower crash rate than the classified subset and are dominated by *Resolved No Fix* outcomes, confirming that this exclusion removes non-bugs rather than discarding bug coverage.

To explore the corpus by root cause, we additionally embedded the reports and built an interactive clustering visualization of the corpus, which groups bugs of similar root cause; details and a screenshot are in subsection D.

A. Component-Specific Bug Distribution

Different components require different mutation strategies to reach (JIT bugs need warmup loops, standard library bugs need structured API calls, parser bugs need exotic syntax), so the per-engine distribution of bugs across components informs which generation techniques are worth applying to which target. Table II (*Bugs* columns) reveals strikingly different bug profiles across engines, mirroring the codebase composition shown in the same table's *LoC* columns.

Bug profiles diverge by engine architecture. *First, V8 is dominated by core-engine bugs.* Code-execution bugs (interpreter and JIT) account for 41.5% of V8 bugs (JIT alone 25.9%), and data-model bugs (type system and memory management) add another 29.9%, together over 70% of V8 bugs and far exceeding any other engine. V8 also has the highest security (17.4%) and crash (53.8%) bug rates, consistent with its role as a browser attack surface. *Second, interpreted languages are dominated by standard library bugs.* CPython (76.8%), PHP (73.0%), and Ruby (61.4%) all have their plurality of bugs in the standard library, mirroring its 26.7–48.6% share of codebase composition (Table II). *Third, optimization bugs are near-exclusive to JIT engines.* CPython, PHP, and Ruby each show under 2% optimization bugs (1.5%, 1.4%, 1.9%), so JIT-targeting techniques are inapplicable to most interpreted languages. *Finally, frontend bugs are rare but non-negligible.* Parsers account for only 2.9–7.1% of bugs despite processing untrusted input first; Ruby's higher rate (7.1%) reflects its complex grammar.

Finding 2: Bug distribution broadly aligns with codebase composition. V8's aggressive optimization creates a qualitatively different bug profile from interpreted languages. A fuzzer designed for one profile will miss the dominant bug class of the other.

B. Crash Root Cause Analysis

Crash root causes split into two layers across our 20,933 crash bugs (Table III): engine-specific peaks driven by implementation choices (JIT, refcount, recursive parsers), and a language-agnostic baseline where null-deref and UAF are top-two in every engine, together accounting for roughly 35% of all crashes. The two layers demand different oracles and generation strategies, so their distribution across engines informs which fuzzer techniques are worth investing in.

TABLE III: Top crash root causes across languages (20,933 total crash bugs). Bold marks the top two causes per engine. Shading is proportional within columns.

Root Cause	V8	CPython	PHP	Ruby
Null ptr deref	**527**	**833**	**2,042**	**430**
Use-after-free	455	**770**	**2,049**	**481**
Type confusion	**749**	254	240	126
Buffer overflow	207	304	622	94
Stack overflow	108	343	519	184
Integer overflow	123	158	234	14
Uninit. memory	155	119	191	39
Double free	4	72	250	21
Assertion failure	91	92	129	18
Total	4,431	4,686	10,532	2,247

Type confusion is disproportionately common in V8 (749 cases, 16.9% of V8 crashes), driven by speculative type optimization: a JIT-compiled function caches a type assumption, a side effect (reassignment, shape mutation, GC relocation) invalidates it, and the next call operates on a mistyped value because its type check was eliminated. Type confusion is the most exploitable crash class, since stale assumptions yield read/write primitives at controlled offsets.

Null-deref and UAF dominance has structural roots common to all engines. Three patterns recur across all four engines' null-deref and UAF clusters: (1) user-supplied callbacks running mid-operation that free or mutate the object the caller still holds; (2) lifecycle calls that violate documented init/teardown ordering; (3) unchecked `NULL` returns from C-extension APIs at the binding boundary. These shared patterns explain null-deref/UAF dominance across engines despite disjoint GC models; PoC signatures appear in subsection IV-D.

Finding 3: Crash root causes split into an engine-specific layer (V8 type confusion, PHP double-free, PHP/Ruby stack overflows) and a dominant language-agnostic layer (null-deref and UAF, driven by three shared structural patterns), so fuzzing strategies targeting the latter are likely to generalize across engines.

C. Temporal Evolution

Whether fuzzing remains worthwhile in mature engines, and when new bug classes warrant retooled mutators, are empirical questions answered by the temporal distribution. Figure 3 plots bug density per year by component, normalized by the volume of newly added code.

Three temporal patterns recur across all four engines. *First, bug discovery rates do not decline over time.* Despite decades of testing, PHP (since 1998) and CPython (since 2000) continue to receive hundreds of bug reports annually, and V8 maintains over 500 reports per year since 2015. This contradicts

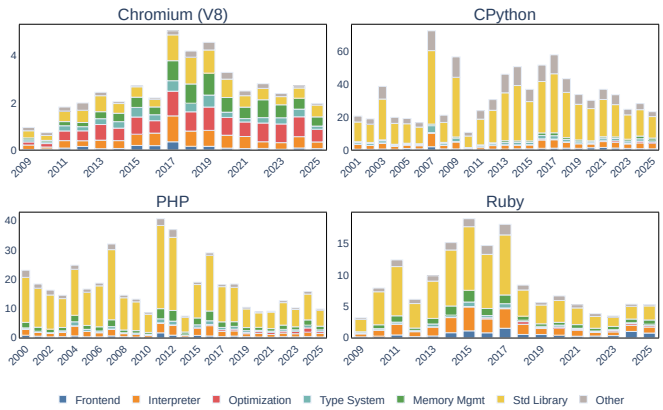


Fig. 3: Bug density by component over time for each language, normalized by the volume of newly added code (bugs per kLoC of added source lines, with generated and vendored files excluded). Stacked bars show the component composition each year.

the assumption that mature software converges toward zero bugs and underscores the need for continuous automated testing. *Second, standard library bugs dominate throughout each engine's history,* maintaining a consistent proportion even as total bug volume fluctuates, which suggests that standard library testing is a persistent, unsolved challenge rather than a transitional phase. *Third, new components introduce new bug classes.* V8 optimization bugs accelerated after 2014 with the introduction of TurboFan and increasing reliance on speculative optimization, peaking around 2017–2018 alongside major architectural changes (Ignition+TurboFan replacing Full-codegen+Crankshaft). Ruby's memory management bugs increased after the introduction of the generational GC in Ruby 2.1 (2013). PHP saw a spike in optimization bugs following the Zend Opcache JIT in PHP 8.0 (2020). Each architectural addition reopens the attack surface. These trends are robust to normalization: recomputing by bugs per kLoC of newly added code, component share of annual issues, or bugs per commit leaves the standard-library dominance unchanged, though the non-decline result is qualified, since per-unit-of-code and per-commit density is flat or rising in V8 and CPython but declines in PHP and Ruby. Normalized density ranges from roughly 2 to 30 reported bugs per kLoC of new code, consistent with prior defect-density research (15–50 defects per kLoC before testing and 1–3 in shipped commercial software [33]).

Finding 4: Bug discovery rates remain stable over decades, and new architectural features (JIT, generational GC) introduce new vulnerability classes. Mature scripting-language engines do not converge toward zero bugs.

D. Component-Specific Syntactic Signatures

Bug-triggering inputs are not syntactically uniform across components. We parsed all 61,779 extracted PoCs with tree-sitter and computed, per component, the *presence rate* of each grammar production rule, defined as the fraction of files containing at least one occurrence. Rules deviating >15 percentage points from the cross-component mean are flagged

```
var x = new Intl.NumberFormat();
x.formatToParts();
```

(a) **Standard library idiom** (Chromium issue 40089013). A single `Intl.NumberFormat` call with no arguments triggers a DCHECK failure: `formatToParts` indexes past the end of an empty parts collection in the builtin iteration loop.

```
var v1 = [], v2 = [];
v1.__proto__ = v2;
function f() {
  var a = [];
  for (var i = 0; i < 2; i++) {
    a.push([]);
    a = v2;
  }
}
f();
%OptimizeFunctionOnNextCall(f);
f();
```

(b) **Optimization idiom** (Chromium issue 40432418). A JIT-warmed function with a `for` loop that reassigns its inner array through a mutated `__proto__` produces a type-confusion crash via Hydrogen's stale inline-cache map. `%OptimizeFunctionOnNextCall` is a V8 testing intrinsic used to force optimization deterministically in PoCs. In production, JIT compilation is triggered by repeatedly calling a function, typically inside a hot loop.

```
RangeError.prototype.__defineGetter__(
  "name", () => 2147483647);
var v = new RangeError();
new ArrayBuffer(v).slice();
```

(c) **Callback idiom** (Chromium issue 40494817). A user-overridden `name` getter on `RangeError.prototype` returns a huge integer that flows through `ArrayBuffer(rangeErr).slice()` as a buffer length, crashing the slice path. The getter is the re-entry point.

Fig. 4: Representative V8 PoCs for the three productive idioms (subsection IV-D), from the Chromium tracker; corresponding CPython, PHP, and Ruby PoCs are in subsection E.

as *distinctive*. We call the resulting per-component bundles of distinctive rules *idioms*: the structural prerequisites a fuzzer must reproduce to reach the corresponding component's bugs. Three productive idioms emerge across all four languages, plus a fourth pattern (frontend) defined by the absence of any idiom. Representative PoCs appear in Figure 4 (V8) and subsection E (CPython, PHP, Ruby).

Standard library idiom: one-shot API call chains. Call and member-access nodes dominate (*e.g.,* JS `member_expression` +19 pp, Python `attribute` +36 pp over the language mean), while function definitions, loops, and arithmetic are all *below* mean. PoCs are flat call chains with no surrounding scaffolding (Figure 4a), mirroring how attackers exercise standard library bugs in practice: a malformed payload (XML, JSON, regex, date string) supplied to a single API call. This idiom dominates in CPython, PHP, and Ruby, where standard library bugs account for 61–77% of all bugs (Table II).

Optimization idiom: loops and function definitions. Function declarations, formal parameters, and loop iteration are consistently elevated (+17–35 pp across languages). These constructs are necessary for JIT compilation and for type-feedback slots to mature into optimization targets; at the crash-class level this idiom maps to V8 type confusion (section IV), and fuzzers like FuzzJIT and Fuzzilli encode it directly (Figure 4b).

Callback idiom: class definitions and magic-method overrides. A third pattern cuts across type-system and memory-management bugs: re-entering user-defined code

mid-operation (Figure 4c). PoCs share a class- and method-definition signature (Python `class_definition` +31 pp, JS `assignment_expression` +16 pp, PHP `method_declaration` +15 pp). The vehicle differs by language, but the mechanism is the same: user-overridable hooks (`__del__`, `__eq__`, `Proxy` traps, PHP `__get`/`__call`) let user code re-enter and violate the caller's invariants. This is the syntactic counterpart of the null-deref/UAF dominance in section IV; about half of these PoCs also invoke explicit GC pressure to surface latent lifetime errors.

Frontend bugs are defined by the absence of any idiom. No production rule exceeds the +15 pp threshold across any language. Instead, frontend PoCs score below mean on nearly every runtime construct. Parser bugs are triggered by exotic *syntactic* edge cases (destructuring, heredocs, string interpolation) that each appear in <5% of files and therefore do not concentrate into a single rule. Fuzzers targeting frontend bugs need structural diversity, not idiom matching, and so grammar-based fuzzers dominate parser coverage (subsection V-D).

Finding 5: Bug-triggering inputs carry component-specific syntactic signatures (*idioms*) consistent across JavaScript, Python, Ruby, and PHP. Three productive idioms emerge: *Standard library* (flat API call chains), *Optimization*, and *Callback*. This suggests current fuzzers may waste mutation budget on inputs structurally incompatible with the targeted component.

V. Fuzzing Techniques Systematization

Having established the four-component engine model and per-component bug profiles, we turn to Q2 (how fuzzers reach those bugs and where the implementation footprint goes). Four claims structure the rest of the section. We first describe the four-stage pipeline that organizes language fuzzers (preprocessing, generation/mutation, execution, feedback) whose implementation footprint is heavily skewed. Second, preprocessing quality, not mutation novelty, drives coverage. Third, despite a proliferation of names, the surveyed generation and mutation techniques are captured by two operations plus a value-mutation layer. Finally, aggregate edge coverage masks near-disjoint per-component coverage.

Table XVI classifies all 51 fuzzers along five dimensions: *Input*, whether the fuzzer generates full programs (P), input data (D), or both (B); *Framing*, whether the paper positions itself as security (S) or correctness (C) work; *Category*, the core generation or mutation operation; *Component*, which engine subsystem(s) the fuzzer targets; and *Idiom*, which bug-triggering code pattern, if any, the fuzzer's generation encodes. The remaining 10 of 61 surveyed papers are related work, bug studies, or prior surveys excluded from this section. To verify our claims we package a controlled-comparison benchmark in which Nautilus and Fuzzilli, representing the two dominant generation paradigms (grammar-based and IR-based), run with three preprocessing ablations under matched 24-hour campaigns on V8, CPython, PHP, and Ruby, with results reported per-component (subsection V-B, subsection V-D).

TABLE IV: Line count analysis of language fuzzers by functional category (% of total LoC). Preprocessing includes compilation, lifting (syntax handling), environment modeling, and target profiles. Gen/Mut covers intermediate representations, AST definitions (e.g., FuzzIL), mutators, and code generators. Execution includes script running, coverage evaluation, and minimization. Fuzzilli's high "Other" percentage reflects auto-generated protobuf code (33K lines) and distributed fuzzing infrastructure. Nautilus and Polyglot, both grammar-driven, encode language specifications as data, pushing their preprocessing share above 85%.

Fuzzer	Pre	Gen/Mut	Exec	Other	LoC
Fuzzilli	19.6%	32.7%	4.8%	42.9%	89K
DIE	71.5%	22.9%	4.2%	1.5%	19K
CodeAlchemist	59.6%	28.3%	2.8%	9.2%	4K
Nautilus	86.8%	8.2%	2.7%	2.4%	32K
Polyglot	87.4%	6.4%	1.8%	4.3%	10K
Average	**65.0%**	**19.7%**	**3.3%**	**12.1%**	**31K**

A. Fuzzer Architecture

Language fuzzers organize work into a four-stage pipeline (Figure 5): *preprocessing* of grammars, type information, and seed corpora; *generation/mutation* of test programs that are syntactically and ideally semantically valid; *execution* on the target with coverage instrumentation; and *feedback* on what to mutate next. Across the fuzzers we measured (Table IV), preprocessing alone consumes around 65% of total LoC, generation/mutation around 20%, and execution under 5%.

Preprocessing (around 65% of LoC, on average). Seed corpus collection from test suites and real-world code, corpus normalization and validity filtering, grammar or type-spec extraction (manually written for jsfunfuzz, Dharma; statically inferred for CodeAlchemist, SoFi), API and dependency analysis, hot-path profiling, and auxiliary structures such as fragment databases or type-inference caches.

Generation/Mutation (around 20%). Grammar-based generation [34], [35], AST-level mutation (CodeAlchemist [11], DIE [12], Superion [36]), IR-level mutation (Fuzzilli's FuzzIL [9]), and fragment-based assembly. Despite the proliferating names, these are built around recursive descent generation and context-matching splicing (subsection V-C).

Execution (under 5%). Coverage-instrumented runs in the target engine, with fast-reset modes (Fuzzilli's REPRL) and differential harnesses across engines or optimization levels.

Oracles. Crash detection via sanitizers (ASan, UBSan) and debug assertions, plus JIT-specific oracles comparing optimized and unoptimized execution (FuzzJIT [37], JIT-Picker [38]).

Feedback. Edge coverage prioritizes interesting inputs, but is an incomplete signal for language fuzzing (subsection V-D).

B. Preprocessing, tedious, uninteresting yet very important

Research papers typically devote one paragraph to preprocessing while dedicating pages to mutation algorithms. Yet our analysis of open-source implementations reveals preprocessing accounts for a substantial portion of actual code and implementation footprint. Table IV shows our line count analysis of five prominent fuzzers: DIE dedicates 71.5% and CodeAlchemist 59.6% of their code to parsing, type inference, and constraint analysis, while the two grammar-driven fuzzers Nautilus and Polyglot exceed 85% because their target profiles (API names,

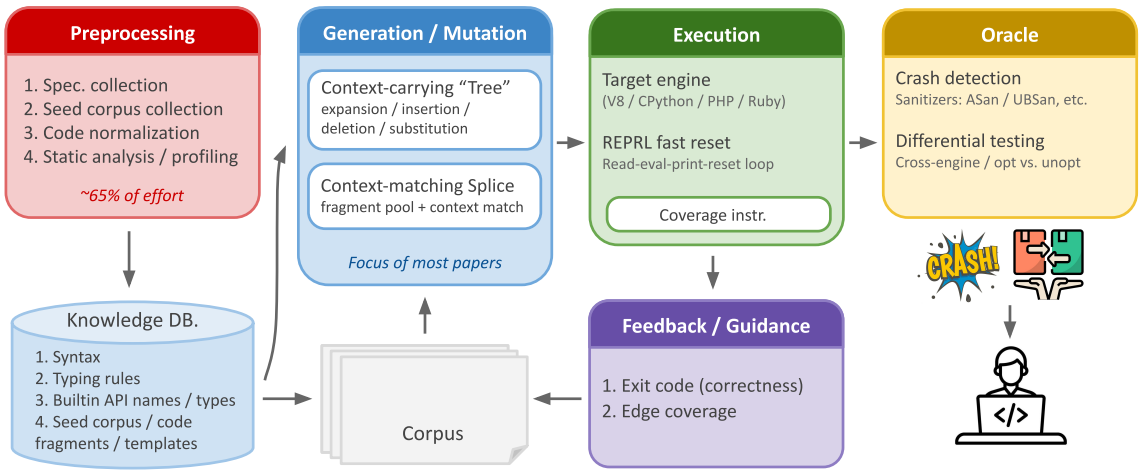


Fig. 5: Overview of a language fuzzer pipeline. Unlike general-purpose fuzzers, language fuzzers dedicate a large share of their implementation footprint to preprocessing (grammar/type extraction, seed curation) and semantic correctness to reach deep engine components.

TABLE V: Ablation study: edge coverage (absolute) after 24h campaigns. *Naut-bare*: bare ANTLR grammar, no builtins. *Naut-aug*: grammar augmented with builtin API names and types, plus fixed-iteration loops and simple function calls. *Fuzz-abl*: Fuzzilli with `JavaScriptEnvironment.swift` commented out (no builtin type signatures) and its hard-coded JIT-triggering templates and generators disabled. Run-to-run variance and Mann-Whitney U significance are reported in Table XV.

Target	Naut-bare	Nautilus	Naut-aug	Fuzz-abl	Fuzzilli
V8	52K	60K	121K	106K	155K
CPython	23K	—	28K	—	—
PHP	9.5K	21K	24K	—	—
Ruby	9.7K	18K	30K	—	—
V8 per-component (K edges)					
Frontend	7.1K	6.7K	6.5K	4.6K	5.6K
Interpreter	10.9K	11.0K	12.5K	12.2K	13.8K
JIT / Opt	4.1K	6.2K	29.0K	43.6K	57.8K
Type System	10.3K	12.2K	30.7K	14.4K	30.2K
Memory Mgmt	5.2K	6.5K	10.0K	10.4K	12.2K
Stdlib / Ext	14.5K	17.4K	32.7K	21.3K	35.7K

types, syntax rules) are checked in as code. The average across these fuzzers is 65%, and this likely underestimates the true implementation footprint since preprocessing code is often the most complex to develop and debug. The details of how we labeled source files in each fuzzer can be found in our artifact [16].

To isolate the effect of preprocessing quality, we conduct ablation experiments (Table V). Nautilus-bare uses only a bare ANTLR grammar with no builtin API names. Nautilus-aug extends the grammar with API signatures and JIT-triggering patterns (fixed-iteration loops and simple function calls). Fuzzilli-abl removes the 4,000-line `JavaScriptEnvironment.swift` file containing type signatures for hundreds of builtin objects and disables its JIT-triggering templates and generators. On V8, this drops Fuzzilli's coverage from 155K to 106K edges (−32%), while augmenting Nautilus's grammar more than doubles coverage from 60K to 121K. The pattern holds across all targets: on Ruby, augmentation yields a 67% improvement (18K to 30K edges). These results indicate that preprocessing quality is a dominant driver of coverage.

As a real-world demonstration of the same argument, we implemented a simple grammar fuzzer driven by a more comprehensive specification of PHP's DOM API than Nautilus's stock grammar contains. Within a 24-hour campaign it surfaced three previously unknown bugs in PHP's DOM parsing and manipulation library (Table XIII, subsection B): a null dereference in `ext/dom/node.c`, an assertion failure in the pre-insert validity check, and a use-after-free on a libxml2-allocated node freed during request shutdown. Porting the same specification into Nautilus's grammar format led Nautilus to rediscover one of these DOM bugs within minutes, confirming that the bug-discovery delta tracks the specification rather than the choice of mutator.

Four preprocessing tasks dominate the implementation footprint. *Specification collection* takes widely varying forms: Nautilus consumes a hand-written CFG, Polyglot extends BNF with JSON semantic annotations, Skyfire learns a probabilistic context-sensitive grammar from a corpus, CodeAlchemist builds a type-annotated fragment database, and SoFi mines TypeScript declarations. *Seed corpus curation*: the DIE corpus (∼14,700 JavaScript files from V8/JSC/SM regression tests and `js-vuln-db` PoCs) has since served as a canonical baseline for subsequent JS-engine fuzzers. *Built-in object instantiation*: DOM and binding fuzzers construct stateful objects through specific API call sequences. Favocado hand-writes initialization statements for binding objects whose dependencies cannot be derived from IDL, Cooper relies on manually edited API templates, and FreeDom encodes DOM-specific generation logic across 30K LoC of Python. *Code normalization* lifts external seeds into the fuzzer's IR. Fuzzilli's `JavaScriptCompiler`, for example, parses JavaScript with Babel and lowers it into FuzzIL SSA form (variables renamed to `v0`, `v1`, ...). The predominance of these tasks implies that reproducing published fuzzers is difficult without preprocessing detail, that fair comparison requires preprocessing-quality accounting, and that mutation-novelty claims should be ablated against preprocessing improvements before being credited.

Preprocessing automation is therefore a key open problem (section VI).

Finding 6: Preprocessing (specification and seed-corpus preparation) averages around 65% of surveyed fuzzers' implementation footprint, yet their papers routinely dismiss it in a single paragraph.

C. Input Generation / Mutation Techniques

Generation and mutation proceed in three steps. We first define semantic correctness and the three error categories (reference, type, value) that any generation or mutation technique must address. We then show that the surveyed techniques, despite a proliferation of names, are captured by two operations (recursive descent generation and context-matching splicing) plus a value-mutation layer, distinguished by which error categories each suppresses. Finally, we examine the structural prerequisites (*idioms*) that fuzzers encode to bias generation toward bug-triggering engine subsystems.

Semantic Correctness Challenges. A program is *semantically correct* if it executes to completion without uncaught exceptions (excluding crashes and assertion failures that represent target bugs); the *correctness rate* is the fraction of generated programs that exit cleanly. Three error categories cause semantic incorrectness. *Reference errors* are accesses to undefined variables or properties, caused by scope mismatches during splicing; solutions include static identifier tracking (CodeAlchemist [11]) and variable lifetime tracking (Fuzzilli [9]). *Type errors* (calls to non-functions, primitive property access, wrong-typed arguments) are harder to prevent because dynamic typing makes validity depend on runtime types; solutions include static type inference [11], runtime type profiling [37], and TypeScript declarations (SoFi [39], TypeNFuzz [40]). *Value errors* are well-typed operations that fail at runtime (division by zero, out-of-bounds access, invalid API arguments); full prevention requires SMT solving or symbolic execution, so most fuzzers use heuristics or accept some error rate.

Two Fundamental Operations. With the error categories defined, we now turn to the operations fuzzers use to navigate them. Across the 51 technique-contributing fuzzers in our corpus, generation and mutation techniques carry a proliferation of names (aspect-preserving mutation in DIE, semantics-aware code assembly in CodeAlchemist, ingredient injection in JavaTailor, structure-aware mutation in GRIMOIRE, FuzzIL instruction splicing in Fuzzilli, among others), yet nearly all are captured by two fundamental operations plus a secondary value-mutation layer. We use this two-operation reduction as an organizing framework that captures the common structure of the surveyed fuzzers while abstracting over differences in graph-based IRs, optimization-guided search, differential oracles, semantic alignment, and environment-aware generation. The three operations differ in which of the error categories above each suppresses, and the surveyed fuzzers' novelty is mostly in how aggressively each is constrained:

- **Recursive descent generation (RD).** A program is built top-down by expanding nonterminals until only terminals remain. Grammar-based fuzzers (Nautilus, GRIMOIRE, Gramatron) are canonical, but the same structure appears in spec-driven generators (COMFORT), template-hole fuzzers (JAttack, Csmith), and LLM-based fuzzers (Montage, CovRL-Fuzz, Fuzz4ALL) whose autoregressive sampling approximates grammar expansion. RD alone does not suppress reference, type, or value errors unless the grammar is augmented with side conditions, which most production grammars lack.
- **Context-matching splicing (CMS).** A code fragment from a pool is inserted at a program point whose *context* (available variables, their types, data-flow dependencies) matches the fragment's requirements. This is the core operation behind DIE ("aspect" = scope + type), CodeAlchemist's assembly step, FuzzIL splicing, Favocado (binding types), JavaTailor's ingredient injection, and Tacoma (scope matching). All share five steps: (1) parse seeds and fragment pool; (2) identify insertion points; (3) determine required context; (4) retrieve matching fragments; (5) splice with variable renaming. Error-guided and optimization-directed fuzzers (GLeeFuzz, OptFuzz) add a selection bias, but the underlying splice is still CMS. CMS is the operation built specifically to suppress reference and type errors: DIE matches scope plus type, CodeAlchemist additionally constrains argument feasibility, Fuzzilli's FuzzIL adds variable lifetime. Value errors remain unhandled by all surveyed CMS fuzzers.
- **Value/token mutation (Val).** A secondary pass substitutes literal values, operators, or tokens without altering program structure. Token-Level Fuzzing [41] relies purely on this; most other fuzzers include it as a subsidiary step. Val preserves AST topology and so cannot fix any of the three error categories, and an operator swap can even introduce a type error in dynamic languages.

Of the 51 fuzzers in Table XVI, 12 are classified as recursive descent (Gen(RD)), 17 as context-matching splicing (Mut(CMS)), 14 combine both (Mut(RD+CMS)), 2 rely on value-level mutation alone (Mut(Val)), and 1 (PyRTFuzz) combines RD with value-level mutation. The remaining 5 fuzzers contribute preprocessing (Pre), oracle (Ora), or postprocessing (Post) exclusively; they are classified in the Category column but are not counted in this breakdown.

Idioms as Search-Space Constraints. The space of syntactically valid programs is exponential in the number of grammar productions, and the vast majority of those programs are semantically rejected or exercise only trivial engine paths. To make mutation tractable, every surveyed fuzzer biases its search toward a small set of *idioms*: code patterns that are never formally specified in language standards but that reliably trigger interesting engine behavior. Because these idioms are informal, fuzzer authors reverse-engineer them from engine source code, seed corpora, or manual experimentation, and the encoding choice determines which engine subsystem the fuzzer can reach. The three productive idioms identified in section IV (Standard library, Optimization, Callback) partition the surveyed fuzzers

along the same axis they partition bugs. We label each fuzzer in Table XVI with the idiom it targets and the encoding mechanism (template, specialized generator/mutator, grammar, typed IR, or signature/spec). DOM and binding fuzzers typically encode both standard library and callback. We identify three recurring and productive idioms from historical bug-triggering inputs, capturing common ways in which fuzzers bias the search space.

- **Optimization idiom: hot loops and warmed functions.** Reaching this idiom requires warming up a function until it is JIT-compiled, then perturbing one of its captured types (Figure 4b). Fuzzilli encodes it via its typed FuzzIL (typed instructions, mostly hot loops); FuzzJIT wraps programs in a JIT-warmup template; OptFuzz steers splicing toward specific TurboFan/Maglev passes. JVM fuzzers (JAttack, Artemis, Jetris) and C-compiler fuzzers (Csmith, YARPGen) reach the same idiom via templates and specialized generators. The idiom is structurally absent from CPython and PHP fuzzers because they lack a production JIT (optimization bugs are 1.5% and 1.4% of their bugs, Table II).
- **Standard library idiom: typed API-call chains.** Reaching this idiom requires syntactically simple programs that exercise specific standard-library APIs with type-correct arguments (Figure 4a). The challenge is preprocessing breadth, not generation novelty: a fuzzer can only call APIs whose signatures it knows, via hand-curated builtin signatures (Fuzzilli's `JavaScriptEnvironment.swift`), TypeScript-declaration extraction (SoFi, TypeNFuzz), or browser-API context modeling (Favocado, GLeeFuzz, DarthShader). PyRTFuzz instead generates Python applications around each API and mutates the data they consume, matching Python's TM2-dominant deployments (subsection III-B).
- **Callback idiom: class and magic-method overrides.** Reaching this idiom requires registering user code (a getter, a magic method, a `Proxy` trap, a DOM event handler) that is re-entered mid-operation by a caller holding a stale reference (Figure 4c). DOM and binding fuzzers encode it by construction (Domato's `onload`/`onclick` bodies, FreeDom's event-handler list, Favocado's property setters, Cooper's script+binding mutation). Outside DOM, JS-engine fuzzers reach it only incidentally: mutational fuzzers inherit `Proxy` handlers and getter/setters from seeds, while grammar fuzzers (Nautilus, jsfunfuzz) miss it unless hook names (`__get__`, `__del__`, `Proxy`) appear in their grammar, which production grammars omit.

Frontend bugs do not cluster into an idiom (section IV). The inputs that trigger them are exotic syntactic edge cases that share no production rule, so we do not place them in this taxonomy. Grammar-based fuzzers (Nautilus, GRIMOIRE, Gramatron, jsfunfuzz, Superion) cover this space as a side effect of broad CFG sampling. Conversely, IR-based fuzzers like Fuzzilli normalize most syntactic edge cases away during IR-to-source lifting, leaving a structural blind spot that no idiom-targeted technique compensates for.

Finding 7: Two techniques are key to fuzzer effectiveness and are repeatedly rediscovered under different names across papers: recursive-descent generation and context-matching splicing (plus a secondary value-mutation layer). The distinguishing choices among surveyed fuzzers are the context representation used to suppress reference and type errors and the idiom encoded to bias the search space, not the mutation operators themselves.

D. Feedback and Evaluation

Edge coverage is the standard feedback signal and evaluation metric for mutational fuzzers (Q3), yet prior work in language fuzzing has rarely asked what those numbers actually cover. We ran 24-hour Fuzzilli and Nautilus campaigns on V8, CPython, PHP, and Ruby, and benchmarked the resulting coverage against each engine's official test suite. Two findings emerge. First, aggregate edge coverage is inadequate as a metric for language fuzzing: Fuzzilli and stock Nautilus reach very different totals (13% vs. 3.5%) but near-disjoint component profiles, and even augmented Nautilus, which closes much of the aggregate gap (10%), still reaches only about half of Fuzzilli's JIT/optimization coverage; the best fuzzer for each engine also trails the test-suite reachability reference by 40.6 percentage points on average. Second, per-component coverage exposes the blind spots aggregate scores hide and motivates a three-category taxonomy of coverage barriers (configuration, API knowledge, environment) that names the engine subsystems no surveyed fuzzer reaches. All coverage results come from eight 24-hour runs per fuzzer-target configuration, on builds instrumented with AddressSanitizer and SanitizerCoverage for all targets, with per-input timeouts and grammar or self-generated seeds (no file corpus); approximate target versions and build flags are in Table XIV. We assess the significance of reported differences with a two-sided Mann-Whitney U test over the eight runs, with p-values in Table V.

1) Inadequacy of Edge Coverage: Aggregate edge coverage alone is an incomplete metric for language engines, where a single binary contains on the order of 100K edges (when instrumented using sancov) and the standard library accounts for the majority. We treat Fuzzilli (canonical IR-based, type-preserving CMS) and Nautilus (canonical grammar-based RD) as representatives of the two paradigms identified in subsection V-C, not as a benchmark. Both are widely deployed, among the most popular language fuzzers on GitHub, and proved highly reproducible to build and run. Yet on V8 the near-disjoint subsystem profiles of stock Nautilus and Fuzzilli (24.2% vs. 19.6% parser coverage, but 2.6% vs. 18.3% JIT coverage) make a single aggregate number nearly uninformative. Augmenting Nautilus (10% aggregate) narrows the total gap but still leaves JIT/optimization coverage at about half of Fuzzilli's.

Edge coverage treats all code equally: a parser branch handling rare syntax has the same weight as a JIT optimization path that processes billions of operations, but for security not all code is equally important. Fuzzilli reaches 18.3% of JIT compiler code on aggregate (13–28% across src/compiler

TABLE VI: Total edge coverage: fuzzer vs. official test suite (24h campaign). The gap shows how much of the reachable code base fuzzers still miss.

Target	Fuzzer	Cov.	Suite	Gap	Edges
V8	Fuzzilli	13.2%	40.8%	−27.6	1170K
CPython	Nautilus	14.8%	62.7%	−47.9	117K
PHP	Nautilus	13.7%	70.1%	−56.4	156K
Ruby	Nautilus	16.2%	46.7%	−30.5	108K
Average		14.5%	55.1%	−40.6	388K

TABLE VII: V8 per-component edge coverage: test suite vs. fuzzers (24h). *Naut.* is stock Nautilus without optimization-triggering patterns; the augmented grammar is reported in Table V. Gaps are percentage-point shortfalls from the test suite.

Component	Suite	Fuzz.	Naut.	Gap F	Gap N
Frontend	52.8%	19.6%	24.2%	−33.2	−28.6
Interpreter	35.6%	21.8%	18.2%	−13.8	−17.4
JIT / Opt	29.1%	18.3%	2.6%	−10.8	−26.5
Type System	32.2%	14.6%	8.7%	−17.6	−23.5
Memory Mgmt	36.9%	19.5%	11.4%	−17.4	−25.5
Stdlib / Ext	34.0%	5.5%	3.3%	−28.5	−30.7
Average	36.8%	16.6%	11.4%	−20.2	−25.4

and src/maglev individually) but only 19.6% of parser code. Stock Nautilus inverts the pattern: 24.2% parser coverage but 2.6% of the optimization pipelines (Table VII). Augmented Nautilus raises JIT/optimization coverage to 7.3% (Table V), narrowing but not closing the gap. These are not within-noise differences. They are the structural consequence of which error categories each fuzzer suppresses (subsection V-C): Fuzzilli's typed FuzzIL produces type-stable hot loops that cross JIT compilation thresholds, while Nautilus's bare grammar generates strings that the V8 interpreter parses but never warms enough to optimize. The ablation in Table V confirms this is preprocessing-driven rather than operation-driven.

To quantify room for improvement, we compare fuzzer coverage against each engine's official test suite (Table VI). The best fuzzer for each target achieves only 13–16% edge coverage, while test suites reach 41–70%, a 40.6 percentage-point gap. Across the three interpreted engines, test suites reach 62.7–70.1% of edges vs. Nautilus's 13.7–16.2%. This is not a dead-code artifact: test suites exercise these edges.

Finding 8: Aggregate edge coverage hides near-disjoint per-component exploration: Fuzzilli and stock Nautilus cover different subsystems, and even augmented Nautilus leaves a roughly twofold JIT/optimization gap; the best fuzzer for each engine leaves a 40-point coverage gap concentrated in components no surveyed fuzzer encodes the idioms for (section IV).

2) Component-Level Coverage as a Complementary Metric: As shown above, aggregate edge coverage is dominated by the standard library (Table II, Table IV). A fuzzer can inflate the headline number simply by improving its standard-library API specification, as our ablation with augmented Nautilus and ablated Fuzzilli demonstrates (Table V). A more honest metric is per-component coverage, which makes blind spots visible.

Component-level coverage reveals blind spots. On V8, JIT optimization components (src/compiler + src/maglev, 398K edges) exhibit stark differences: Fuzzilli reaches 18.3% on aggregate (13–28% across the two subdirectories) while Nautilus achieves 2.6%; conversely, Nautilus dominates parser coverage (24.2% vs Fuzzilli's 19.6%), as grammar fuzzers explore more diverse syntax. For V8 specifically, builtin signatures are not the only encoded expert knowledge: reaching the JIT also requires optimization-triggering patterns. We therefore add fixed-iteration loops and simple function calls to *Naut-aug* and additionally disable *Fuzz-abl*'s hard-coded JIT-triggering templates and generators; this also explains the JIT/Optimization component gap between *Fuzzilli* and *Fuzz-abl* in Table V. Large components remain unreachable: WebAssembly (83K edges, 0.5% all fuzzers) requires binary Wasm modules not JS source. DevTools inspector (4.7K edges, 0%) requires debugger protocol attachment.

Nautilus achieves 0% coverage for all of Maglev, TurboFan, and TurboShaft: grammar-generated code lacks the type-stable hot loops to cross compilation thresholds, and TurboShaft also requires runtime flags absent from default harnesses.

Standard library coverage tells a similar story: two files, `builtins-temporal.cc` (17,623 edges, 0.0%) and `builtins-intl.cc` (5,612 edges, 0.8%), account for over half of the 44,500 edges in `src/builtins/`. V8's test suite covers them via dedicated drivers that exercise Temporal and Intl APIs in loops, but no current fuzzer generates such structured call sequences.

On PHP, the largest component ext/dom (28K edges, 1.5%) is API-driven and requires explicit `new DOMDocument()` calls that grammars never generate. The Zend VM dispatch loop (18K edges, 1.7%) has 200 opcode handlers, each reachable with specific PHP constructs, representing prime grammar improvement targets. Opcache optimizer (17K edges, 0.07%) is dead code: a configuration, not grammar, barrier.

Coverage barriers. Our analysis identifies three systematic gap categories. (1) **Configuration**: compile-time flags (`-enable-debug`), runtime flags (V8 `-jitless`, PHP `opcache.jit`), and environment variables (`PYTHONMALLOC=debug`, `RUBY_GC_STRESS=1`) gate optimization tiers, allocator instrumentation, and debug assertions that few harnesses enable. (2) **API knowledge**: import/require statements are absent from most grammars, and many APIs need specific call sequences (`createElement` before `appendChild`) or argument syntax (JSON, XML, regex). (3) **Environment**: web APIs, binding APIs, filesystem archives (`ext/phar`), and network protocols need setup beyond code generation.

VI. Future Recommendations and Direction

Our systematization reveals three patterns: bugs and idioms cluster by component; generation and mutation techniques reduce to a few operations distinguished by context representation, with preprocessing rather than mutation novelty driving coverage; and aggregate edge coverage hides near-disjoint subsystems and a 40-point gap below test suites.

Specify the threat model. Fuzzing papers should state whether their target deployment exposes TM1 or TM2 and

report the security impact of crashes reachable only under TM1, which imply different inputs and fuzzing strategies (subsection III-B). Today only one surveyed fuzzer states its threat model explicitly.

Specify target component, primary contributing technique, and metric. Authors should state which primary contributing technique (preprocessing, generation, oracle) and engine component (frontend, interpreter, JIT, standard library, GC) their technique targets, and report per-component coverage against a test-suite reference rather than aggregate edge coverage, which standard-library-dominated numbers inflate (subsection V-D).

Describe contributions by input, framing, context representation, and idiom. Future fuzzers should describe their contribution as a choice of (i) whether they generate full programs, data, or both; (ii) security or correctness framing; (iii) the context representation used by CMS; and (iv) the idiom encoded to bias the search space, since novelty lives in these choices rather than in new mutation operators.

Ablate expert knowledge. Preprocessing quality is a dominant driver of coverage (Table V), so evaluations should ablate it with dummy corpora, bare grammars, and removed specifications to isolate a technique's contribution.

Invest in preprocessing automation. Around 65% of fuzzer implementation footprint goes into preprocessing, yet this work is rarely published. We advocate automating specification collection via reflection APIs, Language Server Protocol integration, and LLMs.

Component-targeted strategies. The three coverage-barrier categories (configuration, API knowledge, environment) name under-fuzzed subsystems (V8 builtins, PHP Opcache, Ruby YJIT, embedded Wasm). Component-targeted fuzzing needs component-targeted feedback: track JIT-pass activations rather than edges, and count invoked API endpoints when fuzzing the standard library.

VII. Conclusion

Bug profiles in scripting-language engines diverge as sharply as their architectures: V8 is JIT- and memory-dominated, while CPython, PHP, and Ruby concentrate bugs in their standard libraries. Across 61 papers and 156,595 bugs, we systematize this field around a four-component engine model and two threat models. Preprocessing accounts for around 65% of the implementation footprint and is the dominant coverage driver, yet is routinely dismissed; aggregate edge coverage masks near-disjoint subsystem coverage, leaving a 40-point gap below test suites. We release a labelled bug dataset and per-component-coverage benchmark.

VIII. Ethical Considerations

We disclosed the three previously unknown PHP bugs surfaced by our fuzzing campaigns responsibly to the upstream maintainers. All other bug reports and CVEs analyzed in this paper come from public issue trackers and the National Vulnerability Database, so no non-public security information is reproduced. Our fuzzing campaigns produced minimal crash reproducers only, with no exploit development.

Acknowledgments

We would like to thank Milan Duric for his initial exploratory studies of fuzzers, targets, and bug analysis, as well as for developing a Nix-based build system for the benchmark to ensure reproducible builds of fuzzer and target binaries as part of his master's thesis. We also thank Qiang Liu for contributing to the initial brainstorming discussions and the HexHive crew for their valuable feedback throughout the project. The research was partly supported by SNSF grant number 200021-236559 and SCCH COMET module DEPS (FFG 888338).

References

[1] D. Göhmann, "Vulnerability reward program: 2024 in review," Google Security Blog, Mar. 2025, https://security.googleblog.com/2025/03/vulnerability-reward-program-2024-in.html.

[2] Google, "V8 JavaScript engine," https://v8.dev/, accessed: 2026-02-02.

[3] Mozilla Foundation, "SpiderMonkey JavaScript/WebAssembly engine," https://spidermonkey.dev/, accessed: 2026-02-02.

[4] Apple Inc., "JavaScriptCore," https://developer.apple.com/documentation/javascriptcore, accessed: 2026-02-02.

[5] Microsoft, "ChakraCore," https://github.com/chakra-core/ChakraCore, 2016, open source JavaScript engine.

[6] Python Software Foundation, "CPython: The Python programming language implementation," https://github.com/python/cpython, accessed: 2026-02-02.

[7] The PHP Group, "PHP: Hypertext preprocessor," https://www.php.net/, accessed: 2026-02-02.

[8] Ruby Core Team, "Ruby programming language," https://www.ruby-lang.org/, accessed: 2026-02-02.

[9] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, "FUZZILLI: Fuzzing for JavaScript JIT compiler vulnerabilities," in *Proceedings of the 2023 Network and Distributed System Security Symposium*. Internet Society, Feb. 2023. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/fuzzilli-fuzzing-for-javascript-jit-compiler-vulnerabilities/

[10] J. Ruderman, "Introducing jsfunfuzz," https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, 2007, mozilla JavaScript fuzzer; source at https://github.com/MozillaSecurity/funfuzz.

[11] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines," in *Proceedings of the 2019 Network and Distributed System Security Symposium*. Internet Society, Feb. 2019. [Online]. Available: https://doi.org/10.14722/ndss.2019.23263

[12] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing JavaScript engines with aspect-preserving mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2020, pp. 1629–1642. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00067

[13] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "FuzzBench: An open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 1393–1403. [Online]. Available: https://doi.org/10.1145/3468264.3473932

[14] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2016, pp. 110–121. [Online]. Available: https://doi.org/10.1109/SP.2016.15

[15] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 49:1–49:29, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3428334

[16] C. Zhang, "Artifact for "sok: What are we really fuzzing? a systematization of scripting language fuzzing"," https://doi.org/10.5281/zenodo.23704185, 2026, zenodo record 23704185.

[17] Computing Research and Education Association of Australasia, "CORE conference rankings portal," https://portal.core.edu.au/conf-ranks/, accessed: 2026-02-12.

[18] C. Li, Y. Jiang, C. Xu, and Z. Su, "Validating JIT Compilers via Compilation Space Exploration," in *Proceedings of the 29th Symposium on Operating Systems Principles*. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 66–79. [Online]. Available: https://doi.org/10.1145/3600006.3613140

[19] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2006.

[20] Ecma International, "ECMA-262: ECMAScript language specification," https://tc39.es/ecma262/, accessed: 2026-04-29.

[21] Python Software Foundation, "The Python language reference," https://docs.python.org/3/reference/, accessed: 2026-04-29.

[22] PHP Group, "PHP Internals Book," https://www.phpinternalsbook.com/, accessed: 2026-04-29.

[23] Google Project Zero, "CVE-2014-3176: Out-of-bounds write in V8 Array.prototype.concat," https://nvd.nist.gov/vuln/detail/CVE-2014-3176; PoC and analysis at https://github.com/tunz/js-vuln-db/blob/master/v8/CVE-2014-3176.md, 2014, accessed: 2026-04-30.

[24] MITRE, "CVE-2010-0397: NULL pointer dereference in PHP xmlrpc_decode_request on missing methodname," https://nvd.nist.gov/vuln/detail/CVE-2010-0397; reproducer at https://bugs.php.net/bug.php?id=51288, 2010, accessed: 2026-04-30.

[25] PHP Group, "PHP security classification," https://wiki.php.net/security, lists "invocation of specific code, which may be valid but is obviously malicious" under "Not a security issue." Accessed: 2026-05-05.

[26] Python Software Foundation, "Python security response team (PSRT) workflow," https://devguide.python.org/developer-workflow/psrt/, "Severity is calculated from expected or known use, not from worst-case hypothetical scenarios." Accessed: 2026-05-05.

[27] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent Evaluation Practices for Fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2024, pp. 1974–1993. [Online]. Available: https://doi.org/10.1109/SP54263.2024.00137

[28] PHP Group, "PHP manual: Description of core php.ini directives (open_basedir, disable_functions)," https://www.php.net/manual/en/ini.core.php, accessed: 2026-05-05.

[29] C. Fol, "Generic remote exploit techniques for the PHP allocator, and 0days," Talk at BlackAlps 2022, https://www.blackalps.ch/ba-22/talks.php, 2022, accessed: 2026-05-05.

[30] MITRE, "CVE-2026-27952: Sandbox escape in Agenta-API custom code evaluator via RestrictedPython allowlist," https://nvd.nist.gov/vuln/detail/CVE-2026-27952, 2026, accessed: 2026-05-05.

[31] ——, "CVE-2025-9959: Sandbox escape in smolagents local Python executor via dunder attributes," https://nvd.nist.gov/vuln/detail/CVE-2025-9959, 2025, accessed: 2026-05-05.

[32] Y. Jiang, C. Zhang, B. Ruan, J. Liu, M. Rigger, R. H. C. Yap, and Z. Liang, "Fuzzing the PHP Interpreter via Dataflow Fusion," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 6143–6158.

[33] C. Jones, *Software Quality: Analysis and Guidelines for Success*. Boston, MA, USA: International Thomson Computer Press, 1997.

[34] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for Deep Bugs with Grammars," in *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2019. [Online]. Available: https://doi.org/10.14722/ndss.2019.23412

[35] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz, "GRIMOIRE: Synthesizing structure while fuzzing," in *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019, pp. 1985–2002. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko

[36] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, May 2019, pp. 724–735. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00081

[37] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler," in *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 2023, pp. 1865–1882. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/wang-junjie

[38] L. Bernhard, N. Schiller, and T. Holz, "JIT-Picking: Differential Fuzzing of JavaScript Engines," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 351–364. [Online]. Available: https://doi.org/10.1145/3548606.3560624

[39] X. He, X. Xie, Y. Li, J. Sun, F. Li, W. Zou, Y. Liu, L. Yu, J. Zhou, W. Shi, and W. Huo, "SoFi: Reflection-augmented fuzzing for JavaScript engines," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 2229–2242. [Online]. Available: https://doi.org/10.1145/3460120.3484823

[40] Y. Zeng, Y. Wu, and C. Zhang, "TypeNFuzz: Dynamic Type-aware Object Dependence Graph-Guided Fuzzing for JavaScript Library Bug Discovery," *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 6, pp. 1–22, Jun. 2026. [Online]. Available: https://doi.org/10.1145/3765760

[41] C. Salls, C. Jindal, J. Corina, C. Kruegel, and G. Vigna, "Token-Level fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021, pp. 2795–2809. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/salls

[42] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1802.03426

[43] A. Fioraldi, A. Mantovani, D. Maier, and D. Balzarotti, "Dissecting American Fuzzy Lop: A FuzzBench Evaluation," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–26, 2023. [Online]. Available: https://doi.org/10.1145/3580596

[44] J. Wang, Y. Kang, C. Wu, Y. Hu, Y. Sun, J. Ren, Y. Lai, M. Xie, C. Zhang, T. Li, and Z. Wang, "OptFuzz: Optimization Path Guided Fuzzing for JavaScript JIT Compilers," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 865–882. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/wang-jiming

[45] H. Xu, Z. Jiang, Y. Wang, S. Fan, S. Xu, P. Xie, S. Fu, and M. Payer, "Fuzzing JavaScript Engines with a Graph-based IR," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Dec. 2024, pp. 3734–3748. [Online]. Available: https://doi.org/10.1145/3658644.3690336

[46] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 2613–2630. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lee-suyoung

[47] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang, "Automated conformance testing for JavaScript engines via deep compiler fuzzing," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 435–450. [Online]. Available: https://doi.org/10.1145/3453483.3454054

[48] J. Park, D. Youn, K. Lee, and S. Ryu, "Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 493–515, Jun. 2023. [Online]. Available: https://doi.org/10.1145/3591240

[49] L. Wachter, J. Gremminger, C. Wressnegger, M. Payer, and F. Toffalini, "DUMPLING: Fine-grained Differential JavaScript Engine Fuzzing," in *Proceedings 2025 Network and Distributed System Security Symposium*. Internet Society, 2025.

[50] J. Eom, S. Jeong, and T. Kwon, "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*. New York, NY, USA: Association for Computing Machinery, Sep. 2024, pp. 1656–1668. [Online]. Available: https://doi.org/10.1145/3650212.3680389

[51] I. Fratric, "The Great DOM Fuzz-off of 2017," https://googleprojectzero.blogspot.com/2017/09/the-great-dom-fuzz-off-of-2017.html, 2017, google Project Zero blog. Domato source at https://github.com/googleprojectzero/domato.

[52] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupe, and Y. Shoshitaishvili, "Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases," in *Proceedings 2021 Network and Distributed System Security Symposium*. Virtual: Internet Society, 2021. [Online]. Available: https://doi.org/10.14722/ndss.2021.24224

[53] C. Zhou, Q. Zhang, M. Wang, L. Guo, J. Liang, Z. Liu, M. Payer, and Y. Jiang, "Minerva: Browser API fuzzing with dynamic mod-ref analysis," in *Proceedings of the 30th ACM Joint European*

Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 1135–1147. [Online]. Available: https://doi.org/10.1145/3540250.3549107

[54] W. Xu, S. Park, and T. Kim, "FREEDOM: Engineering a State-of-the-Art DOM Fuzzer," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 971–986. [Online]. Available: https://doi.org/10.1145/3372297.3423340

[55] P. Xu, Y. Wang, H. Hu, and P. Su, "COOPER: Testing the Binding Code of Scripting Languages with Cooperative Mutation," in *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022.

[56] J. Wang, P. Qian, X. Huang, X. Ying, Y. Chen, S. Ji, J. Chen, J. Xie, and L. Liu, "Tacoma: Enhanced Browser Fuzzing with Fine-Grained Semantic Alignment," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, Sep. 2024, pp. 1174–1185. [Online]. Available: https://doi.org/10.1145/3650212.3680351

[57] L. Bernhard, N. Schiller, M. Schloegel, N. Bars, and T. Holz, "DarthShader: Fuzzing WebGPU Shader Translators & Compilers," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Dec. 2024, pp. 690–704. [Online]. Available: https://doi.org/10.1145/3658644.3690209

[58] H. Peng, Z. Yao, A. A. Sani, D. J. Tian, and M. Payer, "GLeeFuzz: Fuzzing WebGL Through Error Message Guided Mutation," in *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 2023, pp. 1883–1899. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/peng

[59] J. Lin, C. Luo, M. Zhang, L. Lin, P. Li, and C. Qian, "Fuzzing JavaScript Engines by Fusing JavaScript and WebAssembly," in *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*, 2026.

[60] J. Kwon, B. Jang, J. Lee, and K. Heo, "Optimization-Directed Compiler Fuzzing for Continuous Translation Validation," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 627–650, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3729275

[61] W. Li, H. Yang, X. Luo, L. Cheng, and H. Cai, "PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1645–1659. [Online]. Available: https://doi.org/10.1145/3576915.3623166

[62] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for JVM testing," in *Proceedings of the 44th International Conference on Software Engineering*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 1133–1144. [Online]. Available: https://doi.org/10.1145/3510003.3510059

[63] Y. Zhao, Z. Wang, J. Chen, R. Fu, Y. Lu, T. Gao, and H. Ye, "Program Ingredients Abstraction and Instantiation for Synthesis-based JVM Testing," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Dec. 2024, pp. 3943–3957. [Online]. Available: https://doi.org/10.1145/3658644.3690366

[64] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang, "JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. Melbourne, Australia: IEEE, May 2023, pp. 56–68. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00017

[65] M. Wu, Y. Ouyang, M. Lu, J. Chen, Y. Zhao, H. Cui, G. Yang, and Y. Zhang, "SJFuzz: Seed and Mutator Scheduling for JVM Fuzzing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1062–1074. [Online]. Available: https://doi.org/10.1145/3611643.3616277

[66] Z. Zang, N. Wiatrek, M. Gligoric, and A. Shi, "Compiler Testing using Template Java Programs," in *37th IEEE/ACM International Conference on Automated Software Engineering*. Rochester MI USA: ACM, Oct. 2022, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3551349.3556958

[67] Z. Zang, F.-Y. Yu, A. Thimmaiah, A. Shi, and M. Gligoric, "Java JIT Testing with Template Extraction," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1129–1151, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3643777

[68] S. Chaliasos, T. Sotiropoulos, D. Spinellis, A. Gervais, B. Livshits, and D. Mitropoulos, "Finding typing compiler bugs," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 183–198. [Online]. Available: https://doi.org/10.1145/3519939.3523427

[69] T. Gao, J. Chen, D. Wang, Y. Guo, Y. Zhao, and Z. Wang, "Selecting Initial Seeds for Better JVM Fuzzing," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. IEEE Press, Sep. 2025, pp. 2012–2023.

[70] J. Park, Y. Kim, and I. Yun, "RGFuzz: Rule-Guided Fuzzer for WebAssembly Runtimes," in *2025 IEEE Symposium on Security and Privacy (SP)*, May 2025, pp. 920–938. [Online]. Available: https://doi.org/10.1109/SP61157.2025.00003

[71] C. Watt, M. Trela, P. Lammich, and F. Märkl, "WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 100–123, Jun. 2023. [Online]. Available: https://doi.org/10.1145/3591224

[72] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, Jun. 2011, pp. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[73] S. Li, T. Theodoridis, and Z. Su, "Boosting Compiler Testing by Injecting Real-World Code," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 223–245, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3656386

[74] V. Livinskii, D. Babokin, and J. Regehr, "Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1826–1847, Jun. 2023. [Online]. Available: https://doi.org/10.1145/3591295

[75] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: Bounded translation validation for LLVM," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 65–79. [Online]. Available: https://doi.org/10.1145/3453483.3454030

[76] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 579–594. [Online]. Available: https://doi.org/10.1109/SP.2017.23

[77] P. Srivastava and M. Payer, "Gramatron: Effective grammar-aware fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Denmark: ACM, Jul. 2021, pp. 244–256. [Online]. Available: https://doi.org/10.1145/3460319.3464814

[78] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Fuzz4ALL: Universal Fuzzing with Large Language Models," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Apr. 2024, pp. 1547–1559.

[79] G. Ye, T. Hu, Z. Tang, Z. Fan, S. H. Tan, B. Zhang, W. Qian, and Z. Wang, "A Generative and Mutational Approach for Synthesizing Bug-Exposing Test Cases to Guide Compiler Fuzzing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1127–1139. [Online]. Available: https://doi.org/10.1145/3611643.3616332

[80] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, Jun. 2012, pp. 335–346. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[81] A. F. Donaldson, P. Thomson, V. Teliman, S. Milizia, A. P. Maselco, and A. Karpiński, "Test-case reduction and deduplication almost for free with transformation-based compiler testing," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 1017–1032. [Online]. Available: https://doi.org/10.1145/3453483.3454092

APPENDIX

A. Threat Model PoCs

```
a = [1];
b = [];
a.__defineGetter__(0, function () {
    b.length = 0xffffffff;
});

c = a.concat(b);
```

(a) TM1 PoC for CVE-2014-3176 (V8): a getter on `a` resizes `b` mid-`concat`, and the fast path copies past the buffer, corrupting the V8 heap.

```
$req = '<?xml version="1.0"?><methodCall></methodCall>';
$method = '';
xmlrpc_decode_request($req, $method);
```

(b) TM2 PoC for CVE-2010-0397 (PHP): an XML-RPC request lacking `<methodName>` crashes the PHP worker via a NULL dereference.

B. Bug Dataset Statistics

TABLE VIII: Bug dataset overview across the four engines. CPython slightly exceeds PHP in report count; both have long tracker histories (PHP since 1998, Python since 2000). V8's dataset is smaller but denser in security-critical bugs. Across all languages, we extracted 61,779 proof-of-concept inputs (39.5% of all issues).

	V8	CPython	PHP	Ruby
Total issues	8.2K	74.1K	62.7K	11.6K
Security bugs	1.4K (17.4%)	708 (1.0%)	541 (0.9%)	83 (0.7%)
Crash bugs	4.4K (53.8%)	4.7K (6.3%)	9.6K (15.4%)	2.2K (19.2%)
With PoC	3.1K (37.8%)	22.0K (29.7%)	31.0K (49.3%)	5.8K (49.5%)
NVD CVEs	3.8K	158	733	112
CVEs matched	2.6K	102	363	6

C. Dataset Construction

We crawled four issue trackers end-to-end, applied a deterministic inclusion filter, and then LLM-labeled the survivors. The LLM leaves the component list empty for non-bugs and for reports with insufficient evidence for reliable assignment.

Crawling. We enumerated sequential issue IDs on GitHub (CPython, PHP) and Redmine (Ruby), fetching each report's JSON or HTML. For Chromium, whose IDs are non-sequential and whose pages require JavaScript rendering, we enumerated V8-relevant issues through the tracker's query interface and fetched pages with a headless browser. Proof-of-concept files were downloaded alongside each report.

Inclusion filter. Before labeling we exclude non-bug reports using pull-request status and the trackers' developer-assigned type labels. For CPython this removes 66,772 pull requests from 141,393 fetched items, leaving 74,545 issues. For Ruby we keep only `Bug` reports, dropping 4,565 `Feature` and 695 `Misc`. For PHP we merge the `bugs.php.net` tracker (keeping `Bug` entries) with the GitHub mirror (excluding 15,577 pull requests and feature-request-only reports). Chromium's V8 issues are pre-curated by the tracker's own component taxonomy. Final packaging removes cross-tracker PHP duplicates and a small number of duplicate or invalid records across the other engines, reducing the corpus from 160,862 to 156,595 reports. Open, unconfirmed, duplicate, and later-invalid reports are not dropped here. LLM component assignment below leaves the component list empty without evidence of a genuine bug. We validated the filter on 50 manually labeled reports per language (97.0% precision), though it may admit irrelevant reports that labeling removes. This yields the 156,595 reports summarized in Table VIII.

LLM component extraction. Each surviving report is passed to DeepSeek V3 with a fixed prompt and component taxonomy, producing schema-constrained JSON. The prompt instructs the model to assign a component only where the report provides developer evidence (component tags, root-cause analysis, or affected files) and to leave the list empty otherwise. A report is *classified* if assigned a component in the four-component taxonomy, and *unclassified* otherwise: the empty list covers reports the model judges not to be engine bugs (feature requests, documentation, build or configuration issues, user errors, invalid reports, and third-party problems) as well as reports with insufficient evidence for reliable assignment; test/tooling/documentation reports receive `other`.

Classified vs. unclassified. Table IX compares the two subsets. The unclassified reports are not a random drop-out: their security rate is 3–27× lower than the classified subset (0.2–0.7% vs. 0.9–19.0%) and their crash rate is 2–8× lower, and they are dominated by *Resolved No Fix* resolutions (up to 80.8% for PHP). Excluding them therefore removes non-bugs rather than discarding bug coverage. Uneven removal (8.8% V8; 51.0% PHP) reflects tracker intake rather than labeling bias: `bugs.php.net` has historically accepted far more user-side and configuration reports than Chromium's tracker.

TABLE IX: Classified vs. unclassified subsets across the four engines.

	V8	CPython	PHP	Ruby
Total issues	8,174	74,058	62,738	11,625
Classified (component)	7,451 (91.2%)	50,586 (68.3%)	30,732 (49.0%)	8,883 (76.4%)
Unclassified	723 (8.8%)	23,472 (31.7%)	32,006 (51.0%)	2,742 (23.6%)
Security (classified)	19.0%	1.3%	1.5%	0.9%
Security (unclassified)	0.7%	0.2%	0.3%	0.2%
Crash (classified)	58.3%	8.0%	25.3%	21.9%
Crash (unclassified)	7.4%	2.6%	5.8%	10.5%
Unclassified w/ *Resolved No Fix*	10.1%	48.2%	80.8%	63.0%

TABLE X: Agreement between DeepSeek-V3 labels and the expert reference on the 1,000-report validation set (300 unique reports per expert plus a 100-report shared subset). Component precision, recall, and F1 are micro-averaged over the six one-vs-rest component decisions; κ is the mean of the per-component DeepSeek-vs-expert Cohen's κ values; brackets are 95% bootstrap confidence intervals (10,000 resamples).

Label	Precision	Recall	F1 [95% CI]	Cohen's κ
Component	0.753	0.899	0.819 [0.799, 0.839]	0.655
Crash	0.905	0.930	0.917 [0.882, 0.949]	0.903
Security	0.685	0.949	0.796 [0.695, 0.878]	0.786
PoC	0.861 [0.827, 0.895]	—	—	—

Precision, recall, and F1 are evaluated against the three-expert consensus (majority vote, with Expert 1 breaking ties). Component precision, recall, and F1 are micro-averaged over the six one-vs-rest component decisions; the component κ is the mean of the six per-component values. Cohen's κ is chance-corrected agreement ($\kappa = (P_o - P_e)/(1 - P_e)$, where 0.41–0.60 is moderate, 0.61–0.80 substantial, >0.80 near-perfect). Bracketed intervals are 95% bootstrap percentiles over 10,000 resamples. For PoC extraction we report only precision: experts verified DeepSeek's extracted PoCs rather than independently extracting them, so missed PoCs are unobserved.

TABLE XI: Cross-model agreement on component, crash, and security labels (percentage agreement and Cohen's κ). No ground-truth label exists across models, so precision and recall are undefined and agreement rates are reported instead.

Model pair	V8	CPython	PHP	Ruby	Overall
Component agreement (%) / κ					
DeepSeek-V3 vs Kimi-K2.7	85.7	90.0	89.3	82.4	88.8 / 0.734
DeepSeek-V3 vs GLM-5.2	87.8	87.8	87.3	79.1	86.9 / 0.723
Crash agreement (%) / κ					
DeepSeek-V3 vs Kimi-K2.7	94.7	98.0	97.6	93.5	97.3 / 0.887
DeepSeek-V3 vs GLM-5.2	86.9	98.0	95.5	96.4	96.3 / 0.823
Security agreement (%) / κ					
DeepSeek-V3 vs Kimi-K2.7	88.3	97.0	90.4	90.5	93.4 / 0.303
DeepSeek-V3 vs GLM-5.2	97.2	99.5	99.2	99.5	99.3 / 0.766

D. Bug Analysis Workflow and Interactive Clustering

Figure 7 summarizes the end-to-end bug-analysis workflow: we crawl four issue trackers, apply a deterministic inclusion filter, and label the survivors with an LLM to extract per-report component, security, crash, and PoC fields (the pipeline is described in subsection C). To visualize similar root causes, we embedded the bug reports so semantically similar reports have closer vectors. We project the 512-dimensional vectors to two dimensions with UMAP [42] and cluster the points with HDBSCAN. Each cluster is automatically labeled with its most distinctive terms using c-TF-IDF, and dots are colored by cluster membership. Hovering over a dot shows its LLM characterization and PoC snippet (if available). Clicking opens the original issue in a new tab. Figure 8 shows the tool.

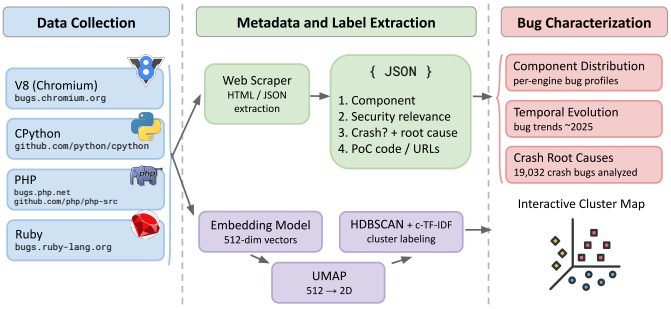


Fig. 7: Overview of the bug-analysis workflow.

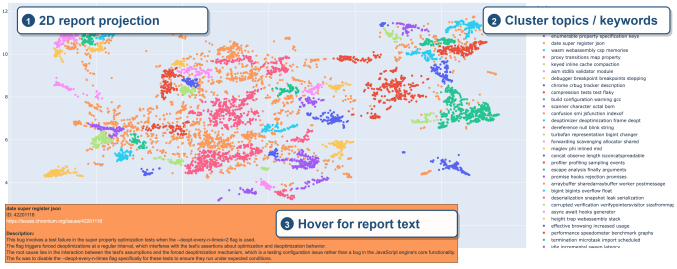


Fig. 8: Screenshot of the interactive report cluster explorer. (1) UMAP projects bug reports into 2D, placing similar reports nearby, and HDBSCAN groups them into clusters. (2) c-TF-IDF keywords summarize each cluster. (3) Hovering over a dot reveals the report text, LLM characterization, and extracted PoC. Clicking a dot opens the original issue.

E. Representative Language Idiom Examples

TABLE XII: Representative language bug reports for the three productive idioms (subsection IV-D). The V8 examples are in Figure 4. Each row links to the tracked report containing the PoC.

Idiom	Issue	Trigger / crash
Stdlib / Ext	CPython-51922	A single `audioop.reverse` call on a one-byte fragment with a sample width of 2 overflows the output buffer.
Stdlib / Ext	PHP-66060	A `DateInterval` with a 2^{127}-day period overflows the internal day counter and crashes.
Optimization	PHP-17577	A `foreach` that reassigns its own array while an inner loop indexes it makes the tracing JIT's packed-type guard dereference a corrupted iterator (segfault).
Optimization	Ruby-15971	A million-iteration hot loop miscompiles the in-place `*=` under MJIT and corrupts the stack.
Callback	CPython-126033	An `__eq__` override clears the tree mid-`remove`, re-entering the caller while it still references the element being compared (use-after-free).
Callback	Ruby-20050	A `hash` override that deletes keys from the very table being built re-enters the insertion path and frees a key still in use (use-after-free).

F. Newly Discovered PHP Bugs

TABLE XIII: Three previously unknown PHP DOM bugs found by our 24-hour fuzzing campaign, all confirmed upstream.

Issue	Crash site (C)	Trigger (PHP)
#1xxxx	`ext/dom/node.c` null deref	`DOMNode::insertBefore`
#1xxxx	`ext/dom/parentnode/tree.c` assert	`DOMNode::before`
#1xxxx	`ext/dom/php_dom.c` UAF	`DOMNode::cloneNode`

G. Fuzzing Campaign Configuration and Ablation Variance

We used default per-execution timeouts, increased to 1,000 ms for configurations with frequent pilot timeouts to prevent timeout handling from dominating execution. Prior work suggests that reasonable timeout adjustments have limited effects on fuzzing outcomes [43].

Neither fuzzer was provided with an external seed corpus. Nautilus is purely generative, while Fuzzilli does not use a seed corpus by default. Although Fuzzilli includes an experimental compiler that translates JavaScript files into its internal FuzzIL representation, we did not enable it in our evaluation. This avoids confounding effects from seed-corpus selection and quality.

TABLE XIV: Build configurations for the 24-hour fuzzing campaigns.

Target	Version	Build flags
V8	14.0.9	`is_debug=false dcheck_always_on=true v8_static_library=true v8_enable_verify_heap=true sanitizer_coverage_flags="trace-pc-guard" target_cpu="x64"`
CPython	3.15.0a6	`-disable-shared -with-static-libpython=yes -with-assertions=yes -with-address-sanitizer=yes -with-pymalloc=no`
PHP	8.5.0-dev	`-disable-all -enable-static -disable-shared -enable-embed=static -enable-debug-assertions -with-pic`
Ruby	3.4.10dev	`-disable-shared -with-static-linked-ext -enable-static -disable-dln -enable-yjit`

TABLE XV: Ablation run-to-run variance and significance. Each *mean* ± *sd* reports 8 runs from the Table V

Target	variant	base	variant (mean ± sd)	base (mean ± sd)	p
V8	Fuzz-abl	Fuzzilli	106,305 ± 4,557	155,880 ± 9,222	1.55×10^{-4}
V8	Naut-bare	Nautilus	52,305 ± 5,156	60,463 ± 5,242	6.22×10^{-4}
V8	Naut-aug	Nautilus	118,339 ± 8,125	61,563 ± 5,242	1.55×10^{-4}
CPython	Naut-bare	Naut-aug	22,653 ± 1,195	27,704 ± 1,872	1.55×10^{-4}
CPython	Naut-aug	Naut-bare	27,704 ± 1,872	22,653 ± 1,195	1.55×10^{-4}
PHP	Naut-bare	Nautilus	9,467 ± 555	21,334 ± 1,448	1.55×10^{-4}
PHP	Naut-aug	Nautilus	24,008 ± 1,279	21,334 ± 1,448	1.86×10^{-3}
Ruby	Naut-bare	Nautilus	9,639 ± 876	17,367 ± 1,466	1.55×10^{-4}
Ruby	Naut-aug	Nautilus	29,749 ± 4,057	17,367 ± 1,466	1.55×10^{-4}

H. Survey Methodology

Initial Search. We queried DBLP for 2011–2025 papers at ICSE, ISSTA, FSE, ASE, PLDI, USENIX Sec, S&P, CCS, NDSS, OSDI, SOSP, and EuroSys with `(fuzzing | testing) & (JavaScript | V8 | Spidermonkey | JSC | WebAssembly | PHP | Python | JVM | "JIT compiler")`.

Inclusion Criteria. Papers must (1) target a scripting-language engine or its immediate infrastructure (e.g., JIT compiler, standard library, garbage collector) and (2) introduce a novel input-generation, mutation, or oracle technique. We excluded papers using scripting languages only as case studies.

TABLE XVI: Surveyed fuzzers (academic except Domato). **Input**: P = full program, D = data, B = both, — = N/A. **Framing**: S = Security, C = Correctness. **Category**: Gen(RD) = recursive descent, Mut(CMS) = context-matching splicing, Mut(Val) = value mutation, Pre/Ora/Post = preprocessing/oracle/postprocessing only; "Pre,"/"+Ora" mark novel preprocessing/oracles. **Component**: FE = Frontend, VM = Interpreter/VM, Opt = Optimization/JIT, TS = Type System, GC = Garbage Collection, SL = Stdlib, Bind = Binding. **Idiom: Mechanism**: Opt = hot loops, Stdlib = typed API-call chains, CB = callback re-entry, — = none (T = template, M = mutator, Gr = grammar, IR = typed IR, Sig = signature/spec). JVM/C++/WebAssembly/Vulkan entries transfer (§ 2).

Fuzzer	Target(s)	Venue	Year	Input	Framing	Category	Component	Idiom
JavaScript / Browser Engines								
Fuzzilli [9]	V8, JSC, SM	NDSS	2023	P	S	Mut(RD+CMS)	Opt	Opt: IR
CodeAlchemist [11]	V8, JSC, SM, ChakraCore	NDSS	2019	P	S	Mut(CMS)	FE, GC, Opt, SL, TS, VM	Opt: T
DIE [12]	V8, JSC, SM, ChakraCore	S&P	2020	P	S	Pre,Mut(RD+CMS)	FE, GC, Opt, SL, VM	Opt: M
SoFi [39]	V8, JSC, SM	CCS	2021	P	S	Mut(CMS)	FE, GC, Opt, SL, VM	Stdlib: M
FuzzJIT [37]	V8, JSC, SM, ChakraCore	USENIX Sec	2023	P	S	Mut(CMS)+Ora	Opt	Opt: T
OptFuzz [44]	V8, JSC, SM	USENIX Sec	2024	P	S	Mut(RD+CMS)	Opt	Opt: T
FuzzFlow [45]	V8, JSC, SM	CCS	2024	P	S	Mut(CMS)	GC, Opt, VM	Opt: M
Montage [46]	V8, JSC, SM, ChakraCore	USENIX Sec	2020	P	S	Mut(CMS)	FE, Opt, VM	—
COMFORT [47]	10 JS engines	PLDI	2021	P	C	Gen(RD)	FE, Opt, SL, VM	Stdlib: Sig
JEST [48]	JS engines + transpilers	PLDI	2023	P	C	Pre,Mut(CMS)	FE, SL, VM	—
JIT-Picker [38]	V8, JSC, SM	CCS	2022	P	S	Ora	Opt, VM	Opt: IR
DUMPLING [49]	V8, JSC, SM	NDSS	2025	P	S	Ora	Opt, VM	Opt: IR
Superion [36]	JS, XML interpreters	ICSE	2019	B	S	Mut(CMS)	FE, VM	—
CovRL-Fuzz [50]	JS interpreters	ISSTA	2024	P	S	Mut(Val)	FE, VM	—
Domato [51]	Chrome, Firefox, Safari	PZ Blog	2017	P	S	Gen(RD)	Bind, FE, SL	Stdlib, CB: Gr
Favocado [52]	V8, JSC	NDSS	2021	P	S	Gen(RD)	Bind	Stdlib: Sig
Minerva [53]	Safari, Firefox, Chrome	FSE	2022	P	S	Gen(RD)	Bind, SL	Stdlib: Sig
FreeDom [54]	Safari, Firefox, Chrome	CCS	2020	P	S	Mut(RD+CMS)	Bind, FE, SL	Stdlib, CB: IR
Cooper [55]	Acrobat, Foxit, Office	NDSS	2022	B	S	Mut(RD+CMS)	Bind, SL	Stdlib: T, Sig
Tacoma [56]	Chromium, Safari, Firefox	ISSTA	2024	P	S	Gen(RD)	Bind	Stdlib: Sig
DarthShader [57]	Chrome, Firefox, Safari	CCS	2024	P	S	Mut(RD+CMS)	FE, Opt	—
GLeeFuzz [58]	WebGL (browsers)	USENIX Sec	2023	P	S	Pre,Mut(RD+CMS)	Bind, SL	Stdlib: Sig
Mad-Eye [59]	V8, SM, JSC + Wasm	ICSE	2026	P	S	Mut(RD+CMS)	Bind, FE, GC, Opt, TS, VM	CB: M
Optimuzz [60]	TurboFan (V8 JIT)	PLDI	2025	P	C	Mut(CMS)	Opt	Opt: M
TypeNFuzz [40]	Node.js libraries	TOSEM	2026	P	C	Gen(RD)	Bind, SL	Stdlib: Sig
PHP								
Dataflow Fusion [32]	PHP interpreter	USENIX Sec	2025	P	S	Mut(CMS)	FE, GC, Opt, SL, VM	Stdlib: Sig
Python								
PyRTFuzz [61]	CPython runtime	CCS	2023	B	S	Pre,Gen(RD)+Mut(Val)	FE, SL, VM	Stdlib: Sig
JVM (Java/Kotlin/Groovy)								
Artemis [18]	HotSpot, OpenJ9, ART	SOSP	2023	P	C	Mut(CMS)	Opt, VM	Opt: T
JavaTailor [62]	HotSpot, OpenJ9	ICSE	2022	P	C	Mut(CMS)	FE, GC, Opt, SL, VM	Opt: T
Jetris [63]	HotSpot, OpenJ9	CCS	2024	P	S	Mut(RD+CMS)	FE, Opt, VM	Opt: T
JITfuzz [64]	HotSpot, OpenJ9	ICSE	2023	P	C	Mut(CMS)	Opt, VM	Opt: M
SJFuzz [65]	HotSpot, OpenJ9	FSE	2023	P	C	Mut(CMS)	FE, Opt, VM	—
JAttack [66]	HotSpot	ASE	2022	P	C	Gen(RD)	Opt	Opt: T
LeJit [67]	HotSpot, OpenJ9, Graal	FSE	2024	P	C	Gen(RD)	Opt	Opt: T
Hephaestus [68]	Java/Kotlin/Groovy	PLDI	2022	P	C	Mut(RD+CMS)	FE, TS	—
Seed Selection [69]	HotSpot, OpenJ9	ICSE	2025	—	C	Pre	VM	—
WebAssembly								
RGFuzz [70]	wasmtime, V8, SM, JSC	S&P	2025	P	S	Gen(RD)	FE, Opt	Opt: Sig
WasmRef [71]	wasmtime	PLDI	2023	—	C	Ora	VM	—
C/C++ Compilers								
Csmith [72]	GCC, LLVM	PLDI	2011	P	C	Gen(RD)	FE, Opt	Opt: Gr
CReal [73]	GCC, LLVM	PLDI	2024	P	C	Mut(CMS)	Opt	Opt: M
YARPGen v2 [74]	GCC, LLVM, ISPC	PLDI	2023	P	C	Gen(RD)	Opt	Opt: T
Alive2 [75]	LLVM IR	PLDI	2021	P	C	Gen(RD)	Opt	Opt: T
Multi-language / General								
Skyfire [76]	JS, XML engines	S&P	2017	P	S	Pre,Gen(RD)	FE, VM	—
NAUTILUS [34]	Interpreters (general)	NDSS	2019	P	S	Mut(RD+CMS)	FE, GC, Opt, SL, TS, VM	—
GRIMOIRE [35]	Interpreters (general)	USENIX Sec	2019	P	S	Mut(CMS)	FE, VM	—
Gramatron [77]	Interpreters (general)	ISSTA	2021	P	S	Mut(RD+CMS)	VM	—
Token-Level [41]	Interpreters (general)	USENIX Sec	2021	P	S	Mut(Val)	FE, GC, Opt, SL, VM	—
Fuzz4ALL [78]	C/C++/Go/Java/Python	ICSE	2024	P	C	Mut(RD+CMS)	FE, Opt, SL, VM	—
COMFUZZ [79]	JS + Java compilers	FSE	2023	P	C	Mut(RD+CMS)	FE, Opt, SL, VM	Opt, Stdlib: M
Postprocessing / Reduction								
C-Reduce [80]	C compilers	PLDI	2012	—	C	Post	—	—
spirv-fuzz [81]	Vulkan GPU compilers	PLDI	2021	P	C	Mut(CMS)	FE, Opt, VM	Opt: T