TYPEFUZZ: Type Coverage Directed JavaScript Engine Fuzzing (Registered Report)

Tobias Wienand
Ruhr-Universität Bochum
tobias.wienand@rub.de

Lukas Bernhard
Ruhr-Universität Bochum
lukas.bernhard@rub.de

Flavio Toffalini
Ruhr-Universität Bochum
flavio.toffalini@rub.de

***Abstract*—JavaScript (JS) engines apply heavy code optimizations to the executed JS code through Just-in-Time (JIT) compilation. Incorrectly handling JS types during JIT compilation can lead to exploitable bugs in the engine. Current fuzzing techniques for JS engines rely solely on code coverage as the dominant feedback mechanism. However, code coverage primarily captures control-flow diversity rather than data-flow diversity. This limitation is crucial for JS engines, where runtime type information drives JIT compiler optimization decisions.**

In this work, we investigate whether type coverage can improve bug-finding effectiveness over traditional code coverage in JS engines. Our prototype, TYPEFUZZ, tracks heap object types at optimization-sensitive locations during JIT compilation and directs fuzzing exploration toward under-tested type locations. We have implemented TYPEFUZZ on top of Fuzzilli and instrumented V8's Maglev and Turbofan compilers to track 463 type-sensitive locations. Our preliminary evaluation demonstrates that type coverage successfully increases data-flow diversity during JIT compilation by 37.5% compared to code coverage alone, effectively exploring substantially more type-sensitive compiler states. In our preliminary campaign, we discovered four bugs in non-experimental features of V8. All bugs were discoverable with both metrics in this preliminary evaluation, yet the substantial increase in type-diverse states explored suggests potential for discovering type-specific bugs with extended campaigns, enhanced bug oracles (differential testing), and cross-engine evaluation on JavaScriptCore.

I. INTRODUCTION

JavaScript engines is a ubiquitous part of everyday life for billions of people. In web browsers, they execute JavaScript code from websites, while frameworks like Electron use them to power desktop applications such as Discord, VSCode, and Slack. JS engines' complexity and processing of untrusted web code create a large attack surface, making them frequent targets for exploitation [1]. Fuzzing has proven effective for finding JavaScript engine vulnerabilities. Fuzzilli [2] has become a widely-used fuzzer for JavaScript engines, with many works building upon or comparing against it [3], [4], [5], [6], [7], [8]. Like most fuzzers, Fuzzilli uses code coverage as its feedback mechanism to guide test generation.

Fuzzing Workshop (FUZZING) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-04-6
https://dx.doi.org/10.14722/fuzzing.2026.23005
www.ndss-symposium.org

Code coverage feedback is the dominant feedback metric for guided fuzzing, with 77% of fuzzing papers using it for evaluation [9]. However, code coverage primarily captures control-flow diversity rather than data-flow diversity. For JavaScript engines, this means that JIT compilers' speculative type assumptions may not receive adequate fuzzing attention. If these assumptions turn out to be wrong, type confusion vulnerabilities may arise and can lead to memory corruption. Such vulnerabilities have historically been common in JavaScript engines and often serve as initial primitives in exploit chains [10], [11].

This work investigates whether type coverage can improve bug-finding effectiveness over traditional code coverage in JavaScript engines. To achieve this, we propose a type coverage which achieves three objectives: collision-free feedback, side-effect-free instrumentation, and maintainability. First, our type coverage definition is enumerable at compilation time, thus fulfilling the collision-free objective and providing a clear signal to the fuzzer (**O1**). Second, we select hook locations that do not interfere with the JIT optimization steps, ensuring aggressive optimizations are preserved rather than falling back to conservative behavior that might hide bugs (**O2**). Third, our building pipeline automatically detects type-relevant locations, achieving maintainability (**O3**). Specifically, we track heap object types observed at optimization-sensitive locations during JIT compilation, creating a feedback mechanism that rewards exploring various type-dependent compiler behaviors. We have implemented type coverage in TYPEFUZZ by instrumenting V8's Maglev and Turbofan compilers to track 463 type-sensitive locations. For our prototype, we select V8 as it has the most fine-grained type system among major JavaScript engines (313 types vs. 91 for JavaScriptCore and 44 for SpiderMonkey, see Section VII), maximizing the explorable state space for type coverage. Our preliminary results show 37.5% more type-varied states explored compared to when using code coverage feedback instead, with substantial overlap between metrics motivating our proposed extended evaluation.

To investigate whether type coverage can reveal new bug classes, we plan an extended evaluation with enhanced bug oracles. Traditionally, fuzzing JavaScript engines does not utilize memory safety bug oracles such as ASan [12] because traditional memory corruption such as UAF or OOB accesses are rare [1]. Instead, they rely on manually deployed assertions to detect invalid JavaScript code executions. Recent works in

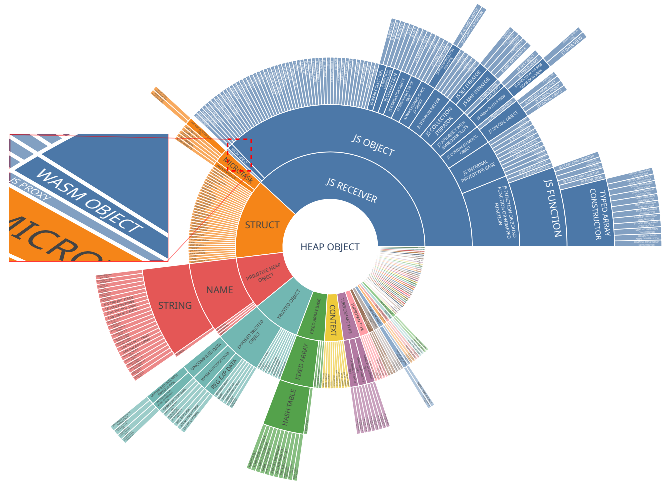

Fig. 1. Complete hierarchy of V8's 313 heap object types. Each object has a map encoding one of these leaf types, which the optimizing compilers use for specialization decisions. The inset shows that WASM objects are the rare exception showing not all JS receivers are JS objects. This edge case caused the type confusion detailed in Section II-D, motivating this work.

JavaScript engine testing propose advanced differential bug oracles that observe invalid code optimizations by comparing execution traces between JIT-optimized and interpreted execution. Dumpling [5] instruments V8 to compare frame states at deoptimization points, detecting miscompilations that produce incorrect results without triggering crashes. These oracles demonstrate that JavaScript engines contain subtle errors rooted in incorrect JIT compilation, which may remain unnoticed for years [13]. Our extended evaluation will combine these differential testing oracles with type coverage to determine whether our feedback mechanism improves bug-finding capabilities.

The main contributions of this paper are:

- We present the design and implementation of TypeFuzz, a type coverage-guided fuzzing system for V8's JIT compilers using automated instrumentation of 463 type-sensitive locations.
- We present preliminary evaluation results demonstrating 37.5% increased type coverage, with four bugs discovered showing substantial overlap between metrics.
- We propose an extended evaluation combining differential testing and extended campaigns to investigate whether deeper type-state exploration reveals unique bug classes.

Our prototype, along with all supporting materials, is available at https://github.com/TobiasWienand/TypeFuzz.

II. Background

We provide background on V8's multi-tiered JIT compilation architecture, its type system, and the Fuzzilli fuzzer that forms the basis of TypeFuzz.

A. V8 JIT Compilation

V8 executes JavaScript using a multi-tiered compilation strategy to balance startup latency and peak performance. Initially, V8's Ignition interpreter executes bytecode generated from the source code. For frequently executed code, V8 employs progressively more aggressive optimizing compilers: Sparkplug (baseline compiler), Maglev (mid-tier), and Turbofan (optimizing compiler).

During execution, Maglev and Turbofan perform *speculative optimization* based on runtime feedback collected during interpretation. V8 records object types and property access patterns of observed JavaScript values in feedback vectors. The optimizing compilers analyze this feedback and generate specialized machine code under the assumption that future values will match past observations. For example, if property accesses consistently target objects with the same memory layout, the compiler can replace dynamic lookups with direct memory accesses at fixed offsets.

V8's speculative approach requires deoptimization guards: runtime checks that verify assumptions remain valid. When assumptions are violated, V8 discards the optimized code and returns to interpreted execution. This mechanism ensures that if the underlying assumptions become invalid, the engine can abandon potentially erroneous code and continue with correct, reliable execution. In the past, several vulnerabilities were caused by neglecting to validate the assumptions made after they have changed [14], [15], [16]. Finding such JIT compiler bugs is thus critical to improve the JS engine security.

B. V8 Type System

V8 represents all JavaScript values (e.g., objects, arrays, strings, and even primitives like `null` and `undefined`) as heap objects. The only exception to this rule are small integers (range [-2^{31}, 2^{31}-1] on 64-bit systems), which are not allocated on V8's heap.

All heap objects have a pointer to a *map* (also called a hidden class) that describes the object's type and how its properties are laid out in memory. The map encodes a fine-grained type as a numeric identifier used by the optimizing compilers for specialization. While the ECMAScript specification defines relatively coarse-grained types, V8 internally maintains 313 distinct leaf types for optimization purposes. For instance, V8 distinguishes between 24 different string representations, whereas ECMAScript defines only a single string type [17], [18]. For example, V8 distinguishes one-byte strings (Latin-1) from two-byte strings (UTF-16). This fine-grained type system enables aggressive specializations during JIT compilation but also creates a large space of possible program states that must be explored during testing. Figure 1 shows the complete hierarchy of V8's 313 heap object types. The sunburst diagram illustrates the hierarchical relationships between types, starting from the root `HeapObject` type and expanding into specialized subtypes. Major branches include primitive types (strings, numbers, symbols), structured types (contexts, arrays, maps), JavaScript objects, and WASM types.

C. Fuzzilli

Fuzzilli is a fuzzer for JavaScript engines with a specific emphasis on finding bugs related to JIT compilation [2]. Since its release in 2019, it has found dozens of bugs and vulnerabilities in JavaScript engines [2]. It addresses the challenge that JavaScript is highly structured, making traditional bit-level mutations such as those from AFL++ ineffective [19]. Fuzzilli operates on FuzzIL, an intermediate representation for JavaScript with a lightweight type system that enables syntactically correct and semantically meaningful mutations, e.g., changing operation inputs, combining programs, or inserting generated code.

Fuzzilli can function as both a mutational and generative fuzzer. Unless a set of seed inputs is used as an initial corpus, Fuzzilli relies on its code generators to build one. These include specialized generators for JIT compiler testing, such as generators that create typed arrays (frequently involved in JIT bugs [7]) and generators that force JIT compilation using V8-specific intrinsics. When new coverage has not been observed for a fixed number of iterations, Fuzzilli transitions to mutation-based fuzzing.

Like most modern fuzzers, Fuzzilli uses code coverage as its feedback mechanism, retaining inputs that trigger new control-flow edges. However, as discussed in Section I, code coverage primarily captures control-flow diversity rather than data-flow diversity, potentially missing bugs that depend on specific runtime type combinations during JIT compilation.

D. Motivating Example: Issue 367818758

A concrete example illustrates the limitations of code coverage-directed fuzzing for finding JIT compiler bugs. V8 commit `81155a8` [20] fixed a type confusion vulnerability (Chromium Issue 367818758) that was not discovered through conventional code coverage-directed fuzzing campaigns. The vulnerability occurred in Turbofan and arose from neglecting the edge case that a WasmStruct can also be a JavaScript receiver, as shown in Figure 1. This type confusion allowed an attacker to exploit prototype chain handling. Code coverage feedback fails to detect such bugs because both the vulnerable case (WasmStruct receiver) and the normal case (JSObject receiver) execute the same code paths. Type coverage addresses this by explicitly tracking which heap object types flow through compiler code, rewarding inputs that exercise different types at the same locations.

III. TypeFuzz's Design

Our type coverage is designed to achieve three objectives:

(O1) Collision-free feedback: We collect type information while avoiding collisions in the coverage map, which may reduce fuzzing efficacy (Section III-A).

(O2) Side-effect-free instrumentation: Hooks must not alter the JIT compiler optimizations, as such alterations may reduce bug-finding effectiveness [3] (Section III-B).

(O3) Maintainability: Our instrumentation aims to be completely automatic, so it can be used on any version of V8 without the need of manual rebasing (Section III-C).

V8 Target Components: To increase data diversity during JIT compilation, we target V8's optimizing compilers, Maglev and Turbofan. Sparkplug is not targeted because it does not perform speculative optimizations. Sparkplug's compilation process is agnostic to heap object types, functioning as a simple non-optimizing compiler. Consequently, type coverage at Sparkplug locations would provide limited insight.

```
1  CompilationDependencies::
2      TransitionDependencyOffTheRecord(
3        MapRef target_map) const {
4    RECORD_MAPREF(target_map, 94);
5
6    if (target_map.CanBeDeprecated()) {
7      return zone_->New<TransitionDependency>(
8        target_map);
9    } else {
10     DCHECK(!target_map.is_deprecated());
11     return nullptr;
12   }
13 }
```

Listing 1: Example type hook instrumenting map transition logic in Turbofan.

A. Type Coverage Definition

We define type coverage as an absolute metric that counts the number of unique (location, type) tuples. A *location* represents a specific point in the compiler code where type-dependent optimization decisions are made (see Listing 1). A *type* refers to V8's fine-grained heap object type (one of 313 leaf types) as encoded in an object's map (see Figure 1). Our type coverage definition can be enumerated at compilation time, thus avoiding collisions (**O1**).

This formulation addresses a limitation of code coverage: while traditional edge coverage tracks which compiler code paths execute, type coverage additionally captures *what data* flows through those paths. For instance, code coverage treats map transitions identically regardless of object type, while type coverage distinguishes these cases.

Type coverage measurements occur at strategically important locations during JIT compilation, such as map transition and deprecation logic. According to GitHub Security Lab analysis of CVE-2021-30632, such locations involve "fairly complex mechanisms with various vulnerabilities found in the past" [21].

Listing 1 shows a concrete example of a type hook inserted into `compilation-dependencies.cc`, a component of Turbofan. The type hook `RECORD_MAPREF` on line 4 measures the type by extracting it from `target_map`, with location ID 94 uniquely identifying this compiler location.

This illustrates the limitation of code coverage that type coverage addresses. With code coverage feedback, reaching this location once would satisfy the coverage metric regardless of which heap object type undergoes the map transition. A fuzzer using only code coverage might trigger this path with a mundane type (e.g., a simple JavaScript object), then discard subsequent test cases involving the same code path with security-relevant types like typed arrays, which have a history of JIT compiler bugs [7]. Type coverage solves this by rewarding the fuzzer every time a new type flows through this location, ensuring comprehensive testing of map transition logic across V8's 313 heap object types.

B. Instrumentation Locations

We insert type hooks at points where the optimizing compilers manipulate heap object references during V8 compilation. By tracing only these code locations and recording type information without modifying compiler state, we preserve the standard JIT optimization passes (**O2**). Specifically, we target locations where three reference types are assigned, reassigned, or passed as function arguments:

- `HeapObjectRef`: General references to heap-allocated objects
- `JSObjectRef`: References specifically to JavaScript objects
- `MapRef`: References to map objects (hidden classes)

While V8 defines additional reference types (e.g., `StringRef`, `BigIntRef`, `FunctionRef`), we do not instrument these because they can only reference objects with a much smaller set of possible instance types, contributing minimally to the explorable state space. For example, `BigIntRef` contributes only one additional state to the explorable state space per location and are therefore not interesting.

While these specific reference types are V8-specific, the instrumentation approach generalizes to other engines. JavaScriptCore's DFG and FTL compilers use `RegisteredStructure` references that wrap `Structure` objects, from which the `JSType` (91 distinct types) can be extracted. Our clang-tidy-based instrumentation (Section IV-B) can be adapted to target these equivalent type-carrying references, as we plan to demonstrate in our extended evaluation on JavaScriptCore.

These reference types appear throughout compiler internals where speculative optimization occurs: property access optimization, map transition handling, dependency tracking, and lowering to machine code. By instrumenting these locations, we capture type information at points where incorrect type assumptions historically led to vulnerabilities.

Our instrumentation places 463 type hooks across Maglev and Turbofan, concentrated in files implementing type-dependent optimizations: `maglev-graph-builder.cc` (145 hooks), `access-info.cc` (59 hooks), `js-native-context-specialization.cc` (54 hooks), and other compiler components handling optimization and lowering. Roughly two thirds (308) are inserted into Turbofan, while the remaining 155 are inserted into Maglev. The numbers are based on V8 commit `5283c8f` (Feb 2025).

C. Automatic Instrumentation

Figure 2 describes our compilation pipeline, which achieves the maintainability objective (**O3**). Our pipeline is designed to automatically detect locations where the JIT compilers manipulate heap object references. The process begins with V8's source code (❶). Before compilation, we utilize clang-tidy, a part of the LLVM toolchain, to fully automate the instrumentation process (❷). Clang-tidy analyzes V8's source code, identifies target locations based on our criteria (assignments, reassignments, and usage as function arguments of the three reference types), and automatically inserts type hooks (❸). This eliminates manual instrumentation and allows TypeFuzz to adapt to V8 updates without code modifications, directly fulfilling our maintainability design goal.

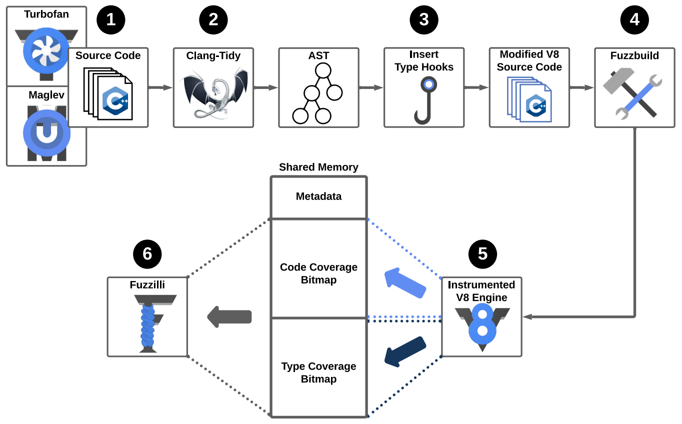

Fig. 2. The simplified build and runtime design of TypeFuzz.

After clang-tidy completes instrumentation, we build the modified source code into an executable using Fuzzilli's build script ❹. This yields the instrumented V8 binary ❺. During runtime, the type hooks extract each heap object's type identifier from its map and record (location ID, type ID) pairs. Any new coverage is written to shared memory set up by Fuzzilli ❻. We extend the shared memory to reserve additional space for type coverage information, allowing Fuzzilli to receive and distinguish between both coverage types.

TypeFuzz uses type coverage feedback by default, though it can also be configured to use hybrid feedback (both type and edge coverage) or edge coverage alone. When V8 executes a test case and triggers JIT compilation, the type hooks record observed (location, type) tuples in shared memory. Fuzzilli reads this information alongside edge coverage and retains test cases that trigger novel tuples, even if edge coverage remains unchanged. This enables the fuzzer to optimize for control-flow diversity (edge coverage) as well as data-flow diversity (type coverage) during JIT compilation.

IV. Implementation

We have implemented TypeFuzz by modifying three software components: V8's optimizing compilers, LLVM's clang-tidy, and Fuzzilli's feedback processing.

A. V8 Modifications

We extended V8 (based on commit `5283c8f`, February 2025) with 1,554 lines of code, primarily consisting of automatically-inserted type hooks. Each hook extracts the heap object type at a specific compiler location and writes a (location ID, type ID) pair to shared memory.

Different reference types require different hook implementations. For `MapRef` objects, we obtain the type by directly reading the instance type from the map. For `HeapObjectRef` and `JSObjectRef` objects, extraction requires access to V8's broker, a compiler component providing safe read-only access to heap metadata during compilation. The distribution of our 463 hooks reflects the prevalence of these reference types in V8's codebase: 341 hooks instrument `MapRef`, 78 instrument `HeapObjectRef`, and 44 instrument `JSObjectRef`, with all eligible references being automatically instrumented rather than selectively chosen.

Our hooks introduce no observable differences between instrumented and unmodified V8 because they operate exclusively during compilation without modifying the JavaScript input or the generated machine code's semantics. This preserves V8's aggressive speculative optimizations, which are essential for triggering JIT compiler bugs. We validated hook correctness by verifying that all expected instrumentations executed during compilation of test workloads.

B. LLVM Modifications

We extended clang-tidy (LLVM commit `27598ab`, February 2025) with 2,470 lines of code implementing custom static analysis checks. These checks operate on V8's abstract syntax tree (AST) to identify instrumentation target locations and automatically insert type hooks.

Each check consists of two components: a matcher that identifies AST patterns (e.g., assignments of `MapRef` variables), and a transformer that modifies matched nodes by inserting appropriate type hooks. The matcher accounts for V8's complex codebase structure, handling scenarios where broker accessors are class members, function parameters, or indirectly obtained through method calls. This automated approach eliminates manual instrumentation and allows TypeFuzz to adapt to V8 updates without modification.

C. Fuzzilli Modifications

We extended Fuzzilli (commit `f31876f`, January 2025) with 435 lines of code to receive and process type coverage information alongside traditional edge coverage. The changes involve giving Fuzzilli the ability to receive type coverage information, log statistics, and distinguish between coverage types. We implemented the type coverage feedback collision-free by mapping each possible (location, type) pair to a unique location in memory. With 463 locations and 313 possible types, this requires extending Fuzzilli's shared memory allocation. When a test case triggers a novel (location, type) tuple, Fuzzilli adds it to the corpus even if edge coverage remains unchanged, thereby guiding exploration toward data-flow diversity during JIT compilation.

V. Preliminary Evaluation

We present preliminary results addressing three RQs:

- **RQ1:** How much additional type coverage does TypeFuzz achieve compared to code coverage? (Section V-C)
- **RQ2:** What bugs does TypeFuzz discover during preliminary evaluation? (Section V-D)

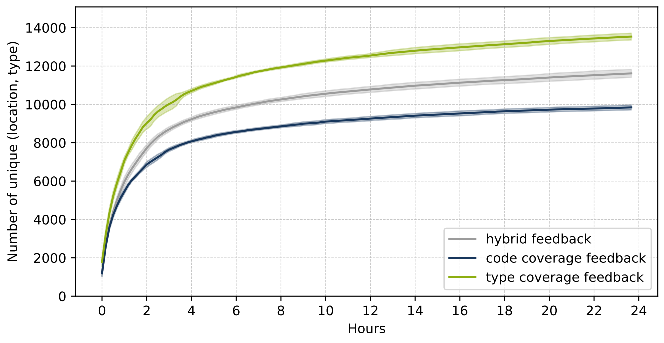

Fig. 3. Type coverage over time during a 24-hour fuzzing campaign, comparing code coverage, type coverage, and hybrid feedback approaches.

- **RQ3:** Does type coverage feedback reduce code coverage? (Section V-E)

Moreover, we discuss a case study of an issue discovered solely through type feedback (Section V-F), which motivates the extended evaluation.

A. Experimental Setup

We conduct experiments on a 192-core machine with 755GB RAM. All fuzzing runs use Docker containers with pinned CPU cores to ensure fair evaluation. For coverage experiments, we use V8 commit `5283c8f` with five Fuzzilli leaf instances and one root instance across 61 cores. We evaluate three feedback approaches: code coverage alone, type coverage alone, and a hybrid approach using both. Each configuration runs five times for 24 hours starting from an empty corpus.

B. Avoiding Queue Bias

During fuzzing with code coverage feedback, samples may achieve new type coverage but not new code coverage. Such samples will not be added to the corpus. If type coverage is calculated only from samples in the corpus, we neglect coverage achieved by discarded samples. This problem is known as queue bias [9]. We avoid this bias by updating coverage counters before the corpus inclusion decision, ensuring all achieved coverage is recorded regardless of which metric determines corpus membership. The same applies symmetrically when measuring code coverage under type coverage feedback.

C. Type Coverage Increase (RQ1)

Type coverage feedback increases the achieved type coverage by 37.5% on average (13,543 unique (location, type) tuples versus 9,850 with code coverage). This demonstrates that type coverage successfully rewards data-flow diversity during JIT compilation beyond what code coverage captures. Figure 3 shows the evolution of type coverage over time during a 24-hour fuzzing campaign.

We observe 133-134 distinct heap object types (out of 313 possible) across all runs regardless of feedback metric, suggesting both approaches explore similar heap object types at a coarse-grained level. The key difference lies in *where*

TABLE I
BUGS DISCOVERED DURING PRELIMINARY EVALUATION. ALL BUGS WERE FOUND WITH BOTH CODE COVERAGE AND TYPE COVERAGE FEEDBACK.

Bug ID	**Component**	**Issue**
399689236	Runtime	JSON.stringify crash
409354670	Maglev	Empty type incorrectly treated as Smi
409905368	Maglev	Post-loop type check failure
413913371	Turbofan	Feedback slot kind mismatch

these types are observed: type coverage feedback tests various JIT compilation stages more thoroughly with different object types. Across the 463 instrumentation locations, type coverage feedback observes 38% more unique types per location on average (37.6 types/location versus 27.2 types/location with code coverage). This increase occurs consistently rather than at a few outlier locations: 223 locations observe more types with type coverage feedback while only 21 observe more with code coverage feedback. This observation suggests the feedback mechanism enables iterative exploration: inputs discovering a type at earlier locations are retained and mutated to propagate that type to different compilation stages.

The most frequently observed types align with common JavaScript usage: `JS_ARRAY` (284 unique locations), `JS_OBJECT` (253 locations), `JS_FUNCTION` (231 locations), `JS_REG_EXP` (213 locations), and `JS_GENERATOR_OBJECT` (211 locations).

Hybrid coverage (combining both metrics) achieves results between the two extremes: it improves type coverage compared to code coverage alone (11,696 tuples on average), though not matching pure type coverage feedback. As shown in Section V-E, hybrid feedback maintains full code coverage while still improving type coverage, suggesting it as a potential compromise approach.

D. Bugs Discovered (RQ2)

During our preliminary evaluation, we discovered four unique bugs in non-experimental features of V8. Table I summarizes these bugs, demonstrating that TYPEFUZZ successfully identifies real compiler vulnerabilities.

All bugs were discoverable with both code coverage and type coverage feedback, indicating substantial overlap in the state space explored by each approach. However, all four bugs were less than two weeks old at the time of discovery, suggesting they were not hard to find and therefore not dependent on exotic (location, type) states. In contrast, bugs that require specific (location, type) combinations, such as Issue 367818758 demonstrated in Section II-D, are hidden in the low-probability regions of the state space. Google's large-scale fuzzing initiative ClusterFuzz already explores parts of this low-probability type state space implicitly through extensive code coverage-directed campaigns. To reach the exceedingly low probability regions where type-specific vulnerabilities reside, we propose doubling the campaign duration in our extended evaluation.

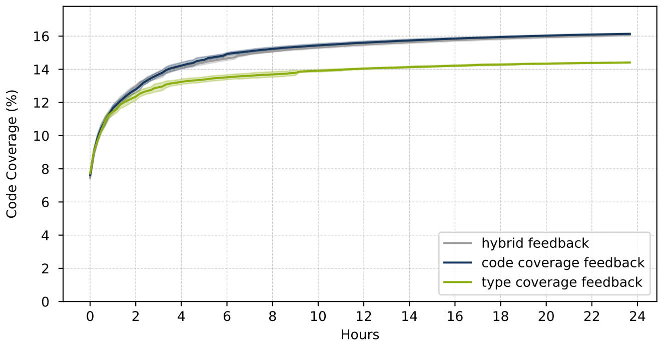

Fig. 4. Code coverage over time during a 24-hour fuzzing campaign, measured in covered edges, with 95% confidence intervals.

These findings align with OptFuzz's V8-specific results [8], where optimization-path guidance and code coverage found similar bug counts despite OptFuzz showing significant advantages on other engines (36 new bugs on JavaScriptCore, SpiderMonkey, and Hermes). This pattern suggests V8 may require extended campaigns and enhanced bug oracles, while cross-engine evaluation tests whether type coverage effectiveness varies by JavaScript engine architecture.

E. Impact on Code Coverage (RQ3)

Type coverage feedback achieves comparable code coverage to traditional code coverage feedback, demonstrating that optimizing for type coverage does not sacrifice control-flow exploration. Figure 4 shows the evolution of code coverage over time during a 24-hour fuzzing campaign.

Across our 24-hour campaigns, type coverage-guided fuzzing achieves two percentage points less code coverage ($\sim$12% relatively less) compared to code coverage-guided fuzzing and hybrid feedback, which both achieve equivalent coverage. This demonstrates that optimizing exclusively for type coverage incurs a modest code coverage penalty, though type coverage still explores substantial portions of the JIT compiler codebase. The hybrid approach achieves code coverage equivalent to pure code coverage feedback while simultaneously improving type coverage, providing an alternative that avoids any code coverage penalty.

F. Revisiting Issue 367818758

In Section II-D, we presented a type confusion vulnerability (Chromium Issue 367818758) that was not discovered through conventional code coverage-directed fuzzing campaigns. We validate the effectiveness of our approach by examining whether type coverage can reach the compiler state associated with this vulnerability.

In our five runs with code coverage, none of the runs tested that location with a WASM struct or WASM object. In contrast, two out of five type coverage-directed runs tested that location with WASM arrays, and four out of five type coverage-directed runs tested it with WASM structs, both of which are suitable to exploit the vulnerability.

This observation is not by chance. In fact, Figure 3 clearly shows that type coverage-directed runs reach 37.5% more of these (location, type) states overall.

The motivating example demonstrates that our approach reaches security-critical compiler states missed by code coverage, validating that our (location, type) metric effectively guides exploration toward type-diverse scenarios. However, reaching the vulnerable state is necessary but not sufficient for bug detection: the original vulnerability caused silent heap corruption without triggering a crash or assertion. Detecting such bugs requires a semantic oracle, motivating our proposed integration of differential testing (Dumpling) in the extended evaluation.

VI. Proposed Extended Evaluation

Our preliminary results show type coverage increases dataflow diversity by 37.5%, yet all four bugs were discoverable with both metrics. This overlap is consistent with recent alternative feedback evaluations on V8 [8], motivating extended campaigns and enhanced bug oracles to investigate whether deeper exploration reveals unique benefits.

Duration and Oracle: We will extend fuzzing duration from 24 hours with 61 cores allocated (Section V-A) to 72 hours with 121 cores allocated. Moreover, we will conduct the experiments by integrating Dumpling mode for differential testing [5]. We will update all components to their latest versions for the extended evaluation. Our setup is necessarily large as testing JavaScript engines requires massive computing resources. Given our plan to include another engine and increase the fuzzing campaign duration, we do not intend to increase the number of repetitions as well, especially since the repetition variance is low and the margins between different configurations already large. This addresses two limitations: (1) Figure 3 shows type coverage continues growing at 24 hours, suggesting insufficient exploration, and (2) type-specific optimization errors may not violate memory safety, requiring semantic bug detection.

Engines: We will evaluate both V8 (313 types) and JavaScriptCore (91 types) to test whether effectiveness varies by engine architecture, motivated by OptFuzz's 3× advantage on JSC vs. V8 [8]. We exclude SpiderMonkey because its type system only defines 44 types, making the explorable state space quite small. Using type coverage as the sole feedback would likely lead to early stagnation, and when combined with code coverage, code coverage would dominate corpus evolution. This effect is already observed in our preliminary results and would be even more pronounced with SpiderMonkey's smaller type space.

Experimental Setup: We will run 72-hour campaigns with Dumpling mode enabled across three feedback conditions on both V8 and JSC: type coverage, code coverage, and hybrid coverage. Each condition will run five times per engine. The code coverage baseline is essential for isolating type coverage's contribution: any unique bugs found by type coverage can be definitively attributed to the feedback metric rather than increased duration or enhanced oracle.

Extended Instrumentation: We will expand our instrumentation to include additional reference types (e.g., `NameRef`, `StringRef`, `ContextRef`) beyond our current selection of `HeapObjectRef`, `JSObjectRef`, and `MapRef`.

Overhead Analysis: We will provide an evaluation of the overhead introduced by type coverage instrumentation in comparison to code coverage instrumentation in terms of instrumentation time and fuzzing throughput.

A. Success Criteria

We define falsifiable criteria to evaluate type coverage:

SC1 (Unique Discovery): Type coverage discovers ≥ 1 new bug on V8 not found by code coverage baseline (same duration, same oracle). By extending the fuzzing campaign and integrating a logic bug oracle (i.e., Dumpling), we have the unique opportunity to measure whether different feedback mechanisms lead to substantially different bug discovery. Furthermore, we will compare time to discovery and provide a case study of found bugs. We will categorize discovered bugs as crashes, assertion failures, or miscompilations (detected via differential testing).

Expected Outcome: By meeting SC1, we expect to observe that different feedback can lead to different classes of bugs discovered, validating type coverage's practical benefit beyond coverage metrics.

SC2 (Cross-Engine): Type coverage demonstrates advantages on JavaScriptCore (higher unique bug count than observed on V8). The instrumentation approach described in Section IV-B can be adapted to instrument other JS engines, e.g., JavaScriptCore.

Expected Outcome: Meeting SC2 indicates engine-independent effectiveness. More specifically, all the benefits observed for V8 can potentially be transferred to other JS engines, thus showing the generality of our approach.

Meeting none of our success criteria suggests type-confusion bugs are triggerable through multiple paths, making specific type guidance unnecessary, or our instrumentation does not capture relevant type variation. Overall, all outcomes provide valuable insights for feedback metric research, including potential negative results.

VII. Discussion

We discuss potential explanations and how our proposed extended evaluation addresses them.

a) Potential Explanations for Preliminary Overlap: We hypothesize that the overlap in discovered bugs stems from significant overlap in the explored regions of V8's state space between code coverage and type coverage directed runs. Both metrics guide the fuzzer toward JIT compiler internals where bugs concentrate (property access optimization, map transitions, deoptimization logic). While type coverage explores substantially more type-varied states (37.5% increase), many bugs may be triggerable through multiple type combinations once the fuzzer reaches the vulnerable code location.

Recent work [8] observed similar patterns when comparing optimization-path guidance with code coverage on V8, despite showing other advantages. This suggests alternative feedback effectiveness may depend on target architecture. However, our analysis of Issue 367818758 provides direct evidence that type coverage reaches security-critical compiler states that code coverage misses: none of the five code coverage runs tested the vulnerable location with WASM types, while type coverage successfully exercised it with both WASM arrays (two of five runs) and WASM structs (four of five runs). This demonstrates that type coverage can expose type-specific vulnerabilities that remain hidden under code coverage-guided exploration.

b) Implications for Fuzzing Research: Code coverage's effectiveness for V8 may reflect V8-specific architectural properties rather than universal principles across all JavaScript engines. Our JavaScriptCore evaluation tests this hypothesis.

These results provide valuable guidance for future research: alternative feedback metrics should demonstrate bug-finding improvements, not merely coverage diversity improvements, before deployment. The theoretical limitations of code coverage, such as missing data-flow diversity, do not always lead to practical problems in every domain, and effectiveness may vary significantly by target architecture.

c) Limitations and Future Work: TYPEFUZZ targets V8's x86-64 build. Extending TYPEFUZZ to other JavaScript engines requires adapting instrumentation to their respective type systems. JavaScriptCore and SpiderMonkey comprise 91 and 44 types respectively, compared to V8's 313 types, resulting in smaller explorable state spaces. We propose to extend the evaluation to JavaScriptCore as an intermediate target, and leave SpiderMonkey to future work.

Our clang-tidy modifications are unlikely to be merged upstream, as they are highly specific to V8 instrumentation. However, this automated approach is more maintainable than manual instrumentation since adapting to new V8 versions requires re-running the tool rather than manually updating hooks. We will release statistics about its robustness across different V8 versions.

VIII. Related Work

A. JavaScript Engine Fuzzing

Early JavaScript engine fuzzers [22] generated JavaScript using a grammar without coverage feedback, discovering over 1,000 bugs in Mozilla's SpiderMonkey [23]. AFL [24] inspired grammar-aware fuzzers [25], [6] that combine code coverage with context-free grammars.

Fuzzilli, the basis of this work, addresses this problem with FuzzIL, a structure-aware IL (intermediate language) for JavaScript with a lightweight type system [2]. DIE [26] uses aspect-preserving mutations to maintain type and structure during fuzzing.

Recent work has employed differential testing to detect semantic JIT compiler bugs. JIT-Picker [3] instruments JavaScript code to compare execution with and without JIT compilation. FuzzJIT [7] uses templates to trigger JIT compilation and compare states, while Dumpling [5] instruments V8 itself rather than generated code.

B. Alternative Feedback Metrics

Most fuzzers use code coverage (line, edge, basic block, or function coverage) as their primary feedback metric [27], [28]. However, the limitations of code coverage have motivated research into alternative metrics.

GreyOne [29] and MemFuzz [30] incorporate data-flow information inspired by taint analysis and memory access profiling to guide fuzzing toward interesting data-flow patterns. Centipede [31] combines multiple feedback sources (call stacks, bounded paths, comparison arguments, anomaly detection) to extend the period during which fuzzing remains guided before feedback signals saturate. For JIT compiler fuzzing specifically, OptFuzz [8] guides Fuzzilli using optimization paths through multiple JavaScript engines' JIT compilers. OptFuzz demonstrated significant advantages on JavaScriptCore, SpiderMonkey, and Hermes, discovering 36 new bugs that baseline approaches (including code-coverage-based Fuzzilli) failed to find. However, results varied by engine: for V8 specifically, OptFuzz and Fuzzilli found similar numbers of bugs (both discovered two bugs, sharing one common bug). LOOL [32] uses optimization logs to guide fuzzing of GraalVM's JIT compiler.

While type coverage increases data-flow diversity, we did not observe bugs discoverable exclusively with this metric in our preliminary evaluation. The overlap in bugs discovered by different metrics indicates that reaching vulnerable compiler code paths matters more than the specific feedback metric used to reach them. This observation is consistent with Centipede's findings [31] that combining multiple feedback sources extends the period during which fuzzing remains effective before signals saturate.

TYPEFUZZ is the first systematic evaluation of type-aware feedback for JavaScript engine fuzzing, demonstrating that while type coverage can be measured and increased, it does not necessarily improve bug discovery over traditional code coverage in preliminary short-duration campaigns.

IX. Conclusion

We present TYPEFUZZ, a type coverage-guided fuzzing system, representing the first systematic evaluation of type-aware feedback for JavaScript engine fuzzing. Our preliminary evaluation shows 37.5% increased type state exploration and four bugs discovered, with substantial metrics overlap.

We propose an extended evaluation combining differential testing (Dumpling mode), extended campaigns, and cross-engine evaluation on JavaScriptCore to investigate whether deeper exploration reveals unique bug classes.

Our work contributes the first systematic infrastructure for type-aware fuzzing of JavaScript engines and provides a rigorous methodology for evaluating whether data-flow diversity improvements translate to practical bug-finding advantages.

Acknowledgment

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] Samuel Groß, "The V8 Sandbox," https://v8.dev/blog/sandbox, 2024.

[2] 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*, 2023.

[3] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "JIT-PICKING: Differential Fuzzing of JavaScript Engines," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[4] C. Salls, C. Jindal, J. Corina, C. Kruegel, and G. Vigna, "Token-Level Fuzzing," in *Proceedings of the 2021 USENIX Security Symposium*, 2021.

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

[6] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-Aware Greybox Fuzzing," in *Proceedings of the 2019 IEEE/ACM International Conference on Software Engineering*, 2019.

[7] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler," in *Proceedings of the 2023 USENIX Security Symposium*, 2023.

[8] J. Wang, Y. Kang, C. Wu, Y. Hu, Y. Sun, J. Ren, Y. Lai, M. Xie, C. Zhang, T. Li *et al.*, "OptFuzz: Optimization Path Guided Fuzzing for JavaScript JIT Compilers," in *Proceedings of the 2024 USENIX Security Symposium*, 2024.

[9] 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 *Proceedings of the 2024 IEEE Symposium on Security and Privacy*, 2024.

[10] CVE News, "CVE-2024-7971: Type Confusion in V8," https://www.cve.news/cve-2024-7971/, 2024.

[11] ——, "CVE-2025-2135: Type Confusion in V8," https://www.cve.news/cve-2025-2135/, 2025.

[12] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," 2012.

[13] V8 Project Authors, "Regression test for issue 346086168," V8 Git Repository, 2024, v8 version 14.2.197-pgo. [Online]. Available: https://chromium.googlesource.com/v8/v8.git/+/refs/tags/14.2.197-pgo/test/mjsunit/regress/regress-346086168.js

[14] Samuel Groß, "CVE-2020-16009," https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2020/CVE-2020-16009.html, 2021.

[15] Sergei Glazunov, "CVE-2021-30551," https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2021/CVE-2021-30551.html, 2021.

[16] Man Yue Mo, "CVE-2021-30632," https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2021/CVE-2021-30632.html, 2021.

[17] ECMAScript, "ECMAScript 2025 Language Specification," https://tc39.es/ecma262/, 2025.

[18] V8, "String Heap Object Types," https://source.chromium.org/chromium/chromium/src/+/main:v8/src/objects/instance-type.h;l=116;drc=fb50f44f04a661e75cf628f1f797d1c1a999df29, 2025.

[19] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *Proceedings of the 2020 USENIX Workshop on Offensive Technologies*, 2020.

[20] V8, "Commit 81155a8," https://chromium-review.googlesource.com/c/v8/v8/+/5901846, 2024.

[21] Google Project Zero, "CVE-2021-30632 Analysis," https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2021/CVE-2021-30632.html, 2021.

[22] Jesse Ruderman, "Introducing jsfunfuzz," https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, 2007.

[23] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *Proceedings of the 2012 USENIX Security Symposium*, 2012.

[24] GitHub, "AFL Archive," https://github.com/google/AFL, 2025.

[25] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for Deep Bugs with Grammars," in *Proceedings of the 2019 Network and Distributed System Security Symposium*, 2019.

[26] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing JavaScript Engines with Aspect-preserving Mutation," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, 2020.

[27] LLVM, "LLVM Sanitizers," https://releases.llvm.org/4.0.1/tools/clang/SanitizerCoverage.html, 2025.

[28] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[29] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GreyOne: Data Flow Sensitive Fuzzing," in *Proceedings of the 2020 USENIX Security Symposium*, 2020.

[30] N. Coppik, O. Schwahn, and N. Suri, "MEMFUZZ: Using Memory Accesses to Guide Fuzzing," in *Proceedings of the 2019 IEEE Conference on Software Testing, Validation and Verification*, 2019.

[31] K. Serebryany, "Rich Coverage Signal and the Consequences for Scaling," in *Proceedings of the 2023 International Fuzzing Workshop*, 2023.

[32] F. Schwarcz, F. Berlakovich, G. Barany, and H. Mössenböck, "LOOL: Low-Overhead, Optimization-Log-Guided Compiler Fuzzing," in *Proceedings of the 2024 ACM International Fuzzing Workshop*, 2024.