Automatic Fuzz Drivers for JavaScript with Type Distributions

Mayant Mukul
University of British Columbia

I. Motivation

Testing libraries with modern fuzzers conventionally involves writing an entry point into the library, called the *fuzz driver*, which invokes library functions (endpoints) in a sequence that meaningfully exercises the library. Manually writing and maintaining effective fuzz drivers is tedious and requires expert knowledge of the library. Existing automatic library fuzzing techniques do not generalize to languages like JavaScript, the most popular programming language among professional developers [1].

II. Background

Prior works like FUDGE [2], FuzzGen [3] and GraphFuzz [4] assume compile-time type checks and invoke endpoints with strict predefined argument types. However, JavaScript allows invoking functions with arbitrary arguments [5], so developers write additional code to validate input types at run-time. To test if a library is robust to unexpected input, a JavaScript fuzzer should invoke endpoints with arguments of unexpected type.

We believe GraphFuzz provides the cleanest presentation of the problem. It encodes fuzz drivers for C++ libraries as directed acyclic graphs – nodes represent endpoint invocations and edges represent data-flow that respects argument types. GraphFuzz generates graphs through mutations that preserve type compatibility between nodes. We relax these mutations by considering *likely* signatures for endpoints instead of strict function signatures.

III. Approach

We introduce the notion of *type distributions* to compactly express likely function signatures (Figure 1a). Instead of a distribution over a flat set of types, we save conditional probabilities discriminated on the kind of type (primitive or complex) with additional metadata for complex types. These distributions can be hand-tuned or inferred using type inference approaches like Guess What [6].

The input to our tool is a list the endpoints and their type distributions. Mutations use type distributions to pick new endpoints (Figure 1). An interpreter receives graphs and executes them, providing coverage feedback to the fuzzer.

IV. Results

We re-implement GraphFuzz for JavaScript with our extensions and evaluate on 5 libraries with handwritten fuzz drivers maintained as part of OSS-Fuzz [7]. We found **two bugs** in our benchmarks. First is a bug where a combination of an unexpected input type and a particular configuration option causes a stack overflow. The handwritten driver uses default options and does not exercise this branch while TypeScript annotations use `any` and allow the unsupported type. Second is a bug where the dynamic type of a returned value deviates from its declared TypeScript type when a function is called with an empty `Uint8Array`. The handwritten fuzz driver for this library does not test empty inputs. These bugs highlight the difficulty of writing and maintaining fuzz drivers that thoroughly exercise the library by hand.

V. Conclusion

In this work we propose loosely typed fuzzing for JavaScript libraries and demonstrate the utility of generating diverse fuzz drivers. Future work will explore further improving code coverage, automatic inference for type distributions, and other program representations that can express control-flow constructs.

References

[1] (2024) Stack overflow developer survey. [Online]. Available: https://survey.stackoverflow.co/2024/technology#1-programming-scripting-and-markup-languages

[2] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 975–985.

[3] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "{FuzzGen}: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2271–2287.

[4] H. Green and T. Avgerinos, "Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1070–1081.

[5] (2024) Ecmascript language specification. [Online]. Available: https://262.ecma-international.org/#sec-ecmascript-function-objects-call-thisargument-argumentslist

[6] D. Stallenberg, M. Olsthoorn, and A. Panichella, "Guess what: Test case generation for javascript with unsupervised probabilistic type inference," in *International Symposium on Search Based Software Engineering*. Springer, 2022, pp. 67–82.

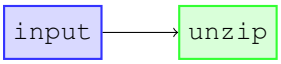
[7] K. Serebryany, "{OSS-Fuzz}-google's continuous fuzzing service for open source software," 2017.

```
"arguments": [{
  "kind": { "string": 0.5, "class": 0.5 },
  "constructorIfClass": { "Uint8Array": 1.0 }
}],
"return": {
  "kind": { "string": 0.5, "class": 0.5 },
  "constructorIfClass": { "Uint8Array": 1.0 }
}
```

(a) Type distribution for example functions `zip` and `unzip` that can receive and return `string` or `Uint8Array` with equal probabilities.

```
const input = /* string input from fuzzer */
const data = unzip(input)
```

(b) A simple fuzz driver



(c) Graph for fuzz driver in (b)

```
const input = /* string input from fuzzer */
const compressed = zip(input)
const data = unzip(compressed)
```

(d) Fuzz driver after mutation

(e) Graph for fuzz driver in (d)

Fig. 1: An example of a mutation that inserts a new node. It searches for an endpoint that is likely to receive and return a `string` to insert between nodes `input` and `unzip` in (c). `zip` fits this constraint, and the result is a new graph (e) that represents a new fuzz driver (d).