IDBFuzz: Web Storage DataBase Fuzzing with Controllable Semantics

Jingyi Chen[1,2], Jinfu Chen[1,2,*], Saihua Cai[1,2,*], Shengran Wang[1]
[1]*School of Computer Science and Communication Engineering, Jiangsu University*
[2]*Jiangsu Key Laboratory of Security Technology for Industrial Cyberspace, Jiangsu University*
Zhenjiang, Jiangsu, China
jychen@stmail.ujs.edu.cn, jinfuchen@ujs.edu.cn, caisaih@ujs.edu.cn, shrwang@stmail.ujs.edu.cn

***Abstract*—Despite great progress in fuzzing browser APIs, systematic approaches for testing web storage techniques remain absent. IndexedDB, the most popular NoSql database in modern browsers, brings unique challenges for fuzzing its API due to its asynchronous event-driven feature and strict phase separation. Current browser fuzzing techniques frequently struggle to generate nested event flows and invocations, which significantly impacts semantic correctness. Moreover, they often rely heavily on the try-catch block to suppress exceptions, which introduces substantial performance overhead. We propose IDBFuzz, the first fuzzing approach tailored for the IndexedDB API, which effectively tackles the challenge of capturing the execution context and event semantics inherent to IndexedDB, as well as handling large persistent objects. We design a seed generator based on intermediate representation (IR) that decouples layered IR skeletons from input object generation. With the aid of a global database snapshot, IDBFuzz can generate semantically controllable seeds, enabling the efficient production of high-quality test cases that significantly improve coverage.**

***Index Terms*—fuzzing, web storage, web API, IndexedDB.**

I. Introduction

Recent years have seen a growing focus on fuzzing techniques aimed at web APIs, which has become an essential means for examining browser security [1]–[4]. For instance, Minerva [5], a representative whole-browser fuzzing technique, employs dynamic dependency analysis to identify interactions between various web APIs, generating closely correlated call sequences to reveal potential logic flaws within components. In parallel, the research community has developed various domain-specific fuzzers that concentrate on specific subsets of web APIs. Fuzzers like Domato [6] and FreeDom [7] focus on testing structural manipulations of the document object model (DOM) and their rendering processes. Additionally, targeted fuzzing frameworks have emerged for various interfaces, including WebGL [8], WebAudio, ServiceWorker [9], and WebGPU [10].

Despite the numerous successes achieved in detecting browser internals, a systematic methodology for fuzzing the built-in persistent storage component that manages structured client-side data remains lacking. IndexedDB API stands out as the most widely used storage API in modern browsers [11]. It exhibits unique characteristics, including a version migration mechanism, a transaction isolation model, and asynchronous event-driven execution chains. Its API usage is governed by strict phase separation throughout the operation lifecycle. For example, schema operations like `deleteObjectStore()` can only be defined or modified during the `upgradeneeded` event, while data operations within transactions are tightly coupled with asynchronous event scheduling [12]. Such features require an approach to accurately capture phase-specific contexts and event-chain dependencies.

```
1  try {var v4 = v2.onfocusin;} catch(e){}
2  try {v6.onversionchange = v4;} catch(e){}
```

Fig. 1. A simplified piece of a seed generated by Minerva [5].

However, most existing browser fuzzers can generate syntactically valid test cases for the IndexedDB API but fail to satisfy the aforementioned semantic requirements on input dependency and contextual behavior. For example, Minerva constrains assignment targets to type-consistent variables, but struggles to simulate complex IndexedDB event flows. As shown in Figure 1, the `onversionchange` event indiscriminately binds `onfocusin`, the `EventHandler` from the UI event. Such binding bypasses the IndexedDB event chain, leaving database-specific callbacks inactive. Moreover, they overlook object lifetimes and transaction boundaries, causing frequent exceptions. To guarantee the error-free execution of generated test cases, a commonly used yet brute-force approach is to wrap all API calls within `try-catch` blocks in order to suppress exception propagation [5], [7], [13]. Although this improves the success rate of execution, it introduces significant performance overhead.

In response to the asynchronous event-driven nature and object-oriented storage features of IndexedDB API, we are committed to developing IDBFuzz, a fuzzing approach specifically designed for browser-side structured storage interfaces. This approach generates precise asynchronous operation flows through a customized intermediate representation (IR) and decouples the IR skeleton and input objects generation to achieve controllable semantics. It refers to maintaining database snapshots and hierarchical IR semantic constraints to ensure that

*Corresponding author

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00346

the generated test cases strictly follow the stage semantics and context dependencies. By combining coverage feedback, it can generate effective seeds with both diverse event-driven control flow and data flow with interleaved read/write operations across transactions and *object stores*, thereby increasing the likelihood of triggering deep and subtle behaviors in the IndexedDB runtime. By far, we have discovered and reported two bugs[1] in Chromium's implementation of this API family, one of which, although previously unknown and recently fixed, cannot be disclosed due to restricted visibility.

II. Background

IndexedDB is a low-level client-side database API designed to store large amounts of data in a structured key–value form [11]. It is constrained by the policy of the same origin, ensuring isolation between different origins. Each origin may host multiple *object stores*, and each *object store* supports versioning mechanisms that enable controlled schema evolution. An *object store* organizes *items* as key–value pairs, where keys may be numbers, strings, dates, or compound keys represented by arrays. IndexedDB also provides secondary indexes for query optimization, asynchronous transactions that avoid UI blocking, and cursor-based iteration for sequential data access.

IR [14] is a formal abstraction between high-level source code and low-level execution that captures program semantics in a structured and analyzable form, which enables modeling of IndexedDB's asynchronous, event-driven features and systematic testing of its complex API behaviors. Building on this, we design an IR-based fuzzing framework for the API.

III. IDBFuzz

The proposed IDBFuzz is a structured fuzzing framework tailored to explore and validate the event-driven behavior of the IndexedDB API. Figure 2 shows its overall architecture, which is composed of three main modules: (i) a schema parsing module that utilizes MDN documentation [11] and the WebIDL specification [15] to automatically abstract a comprehensive IndexedDB API schema, which is further refined into a chain-accessible schemaTree. This process ensures the accuracy of subsequent single-API invocations and type inference. (ii) A seed generation module that leverages our customized intermediate representation to model typical IndexedDB operation flows. This essentially forms an IR without input data, namely IR skeleton. An item driver is also introduced to implement an independent mechanism for item generation and management, which decouples the IR skeleton from the actual input data. Items can be flexibly matched and filled based on the current context information within the IR skeleton. (iii) A converter composed of a lifter, which transforms the generated skeleton into executable JavaScript code, and an HTML wrapper, which embeds the lifted code into a complete HTML template to produce the final executable HTML file. The fuzzing engine mutates both the IR skeleton and items, using coverage feedback to guide exploration toward diverse structures likely to trigger new paths.

[1]https://issues.chromium.org/issues/442392370

A. Intermediate Representation (IR)

1) IR Design: Our IR disentangles data generation from control flow construction. It consists of the IR skeleton that describes the structural operations and concrete input items managed by a separate item corpus. The skeleton is composed of a set of nodes representing basic syntactic units such as expressions, statements, functions, and variables, as shown in Part I of Figure 3. To ensure syntactic correctness, we introduce an auxiliary environment called IRContext, which is responsible for tracking variable declarations, type information, and the set of visible objects across lexical scopes. This guarantees valid variable usage and consistent reference resolution throughout the IR generation process. We further construct a layered IR skeleton to encode the event-driven control flow (as shown in Part II of Figure 3), which includes four types of layers: invocation, registration, execution, and access. We also introduce a coverage-guided data generation component called item driver. During fuzzing, this component dynamically generates valid object items based on a database snapshot IDBContext (as shown in Part III of Figure 3). These items are then injected into the placeholders reserved in the IR skeleton to generate the complete seeds.

2) Achieving Controllable Semantic Correctness: The main issue with current fuzzers that generate test code for the IndexedDB API is their lack of awareness regarding the database state and the context of events. This oversight leads to operations that violate semantic constraints, resulting in many invalid or ignored requests. As a result, these fuzzers heavily depend on `try-catch` blocks to manage runtime errors and ensure successful execution. To address this limitation, we introduce the notion of controllable semantics, which ensures that generated test cases strictly follow the stage semantics and contextual dependencies of IndexedDB, thereby reducing invalid operations and minimizing the reliance on `try-catch`. Its concrete realization is as follows.

We propose IDBContext, a metadata snapshot mechanism that is maintained throughout the entire lifecycle of the database. Its primary role is to ensure that *object store* operations adhere to schema semantics, thereby producing controllable, valid, and structurally rich data access flows. First, during the schema construction phase, IDBContext records metadata such as *object stores*, *indexes*, *transactions*, and *key* sets in real time, forming a semantically consistent global view. This snapshot supports cascading operations: once an *object store* or *index* is deleted, its related structures, such as *index* lists and registered *keys*, are also removed, preventing semantic errors or context pollution caused by stale references. Second, during actual transaction execution, IDBContext synchronously tracks the operation flow within each transaction and ensures that operations from aborted or committed transactions do not contaminate the global view. This design enables us to generate semantically strong test code in a controllable manner. For example, a `get` operation is generated only when the snapshot contains a valid *key*, rather than by constructing parameters solely based on data types.

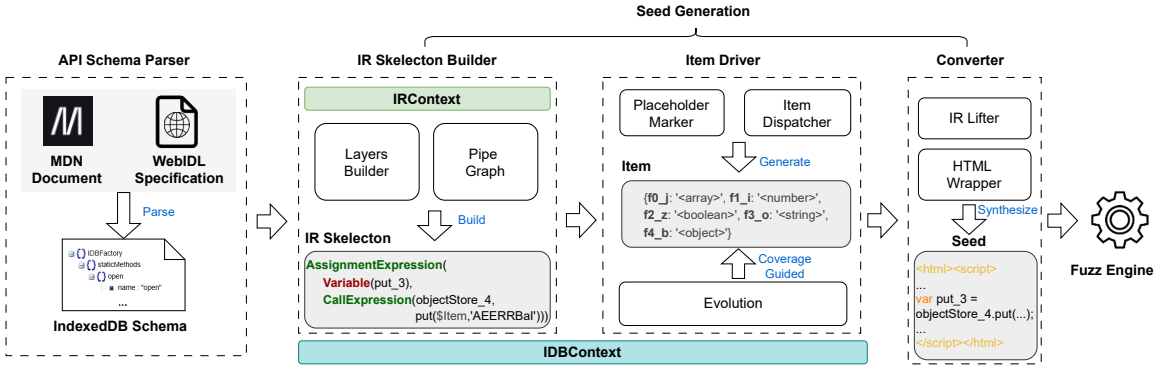

Fig. 2. The overall architecture of IDBFuzz.

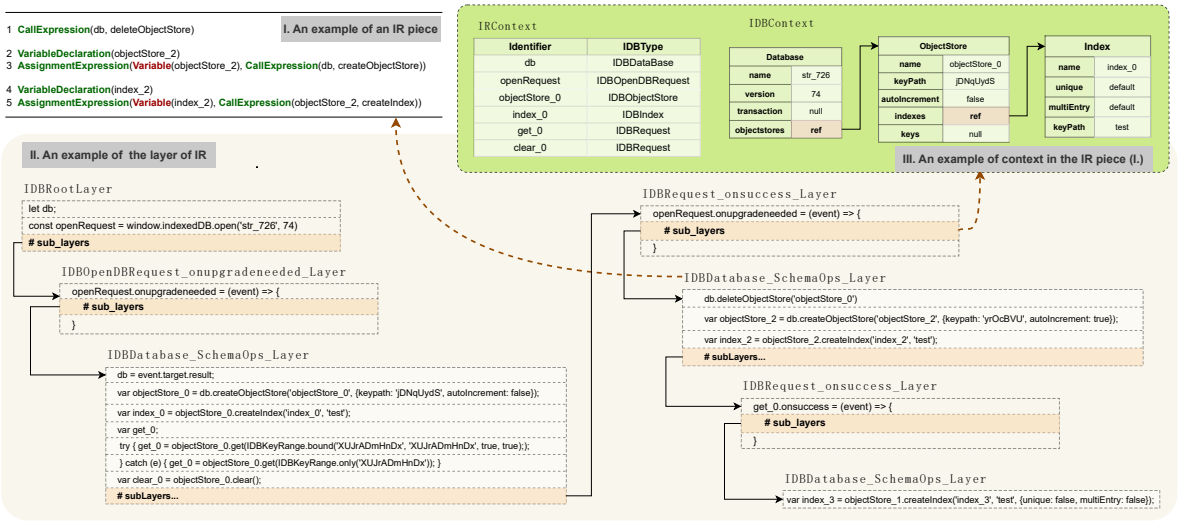

Fig. 3. Illustration of the IR design using schema initialization as an example. Part I shows the IR skeleton composed of basic syntactic nodes. Part II demonstrates the layered control flow structure: (1) database initialization triggers `indexedDB.open()` and registers event handlers like `onupgradeneeded`; (2) schema operations perform structure updates such as `createObjectStore` and `deleteIndex`; (3) data manipulation executes core *object store* APIs and transactional logic; (4) event flow generation models asynchronous nesting using both IDBContext and IRContext. Part III shows how concrete items are generated and injected based on the dual contexts.

Moreover, we construct a context-aware layered IR consisting of over 20 semantic layers. Based on domain knowledge, these layers are first grouped into four functional categories: the invocation layer, registration layer, execution layer, and access layer. These categories naturally exhibit dependency relations, as the registration layer depends on invocation, and both execution and access layers also rely on invocation. We then refine these categories by extracting API dependencies from schemaTree, yielding a complete structure of semantic layers. The four categories correspond respectively to event initiation, event listener registration, data operations, and data queries within transactions. This layered design provides each test path with a clear semantic position and enables robustness evaluation by injecting invalid nodes, such as placing schema-violating operations in the wrong layer to trigger exceptions. This layered design, combined with the snapshot, enables the IR generation to accurately identify the schema state and event context required for each operation during the generation phase, and to place it into the appropriate semantic layer.

3) Efficient Generation of Rich Data Flows: To construct complex access chains and data propagation paths among input objects during transactional interactions, we focus on the `IDBObjectStore` interface, the only interface in the IndexedDB API that allows data read and write operations. We treat all instance methods of `IDBObjectStore` as pipe ends; the sequential relationships between these ends are considered pipes, and multiple pipes form a pipe flow. We construct a directed weighted graph, called a pipe graph, to capture dependency strength among these methods in typical execution sequences. Based on this graph, we can randomly generate pipe flows with arbitrary invocation lengths, data dependencies, and semantic diversity as needed, enabling

the construction of targeted test inputs that drive complex transactional interactions. Specifically, we extract all instance methods of `IDBObjectStore` from the constructed schemaTree to define the pipe ends in the graph. When initializing edge weights, we consider the varying testing value of different method combinations in transactional scenarios. We assign higher weights to operation sequences with clear read-write dependencies, medium weights to common self-loops and mixed write patterns, and lower weights to read-only combinations that typically lack strong dependencies. These initial weights are then dynamically adjusted during fuzzing using the Upper Confidence Bound strategy [16], [17] according to new coverage, unique crashes and novel data access behaviors.

B. Item Driver

Effective fuzzing the IndexedDB API relies not only on the diversity of API call sequences, but also on the ability to generate high-quality, structured input data. IndexedDB is built upon a transactional *object store* system, typically implemented on top of key-value storage engines such as LevelDB [18] or SQLite [19]. Input objects must conform to specific schema constraints, including *keyPath* configurations, *index* definitions, and compound key structures. Also, IndexedDB supports complex usage scenario beyond primitives, such as ImageData, Blob, and large objects.

To meet the above requirement for input data, we design the item driver component to manage a collection of data items that is decoupled from the IR skeleton. Specifically, IDBContext provides database snapshot information to guide the appropriate referencing and insertion of items within test cases. During IR construction, write operation nodes use type labels as placeholders to indicate insertion positions. When generating a test case, the main engine requests candidate items matching the specified *keyPath* based on these labels and fills them into the corresponding operation nodes to form complete test statements. During fuzzing, if replacing only the filled data while keeping the structure unchanged, it can result in new coverage, the item is considered to have contributed to path exploration, and is added to the item corpus. To control the overall size, the item corpus is equipped with a coverage-guided replacement strategy that retains only items with high coverage gains, gradually evolving into a diverse and semantically effective input set.

IV. Case Study

We conducted a preliminary experiment to evaluate the effectiveness of IDBFuzz. The experiment was run on a machine equipped with an Intel(R) Core(TM) Ultra 7 265K CPU (3.90 GHz) and 48 GB RAM. Among some pioneering browser fuzzers [7], [20], [21], we selected Minerva [5] as our baseline for this case study, as it explicitly claims to have found a bug related to IndexedDB API. To evaluate effectiveness, we ran both Minerva and our proposed tool, IDBFuzz, on Chromium in five independent trials, each lasting 24 hours. Table I shows that IDBFuzz consistently covers significantly more edges than Minerva across all five trials, with an average improvement of over 34%.

TABLE I
The numbers of edges triggered in each trial

Fuzzer	1	2	3	4	5	Average
Minerva	154,712	155,003	155,221	155,097	155,387	155,084
IDBFuzz	207,012	208,632	207,944	206,308	209,174	207,814

To evaluate the performance benefits of our approach, we designed a set of micro-benchmark tests. In each round, we generated a transaction that triggered 1,000 `IDBObjectStore` API calls and measured the elapsed time from the start of the transaction to the firing of the complete event. We compared performance under two configurations: (1) wrapping every API call in a `try-catch` block to suppress errors; (2) uses our approach to limit `try-catch` usage and apply it only when necessary. Figure 4 shows that our approach consistently outperforms the configuration 1 across all test rounds. Specifically, it achieves an average reduction of over 29% in execution time, highlighting the efficiency gains from avoiding unnecessary exception handling.

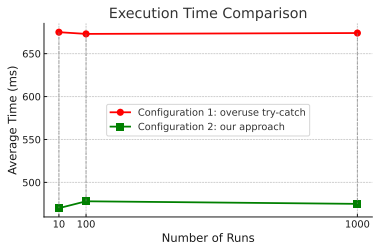

Fig. 4. Execution time comparison under two configurations.

V. Extensibility and Future Work

In terms of extensibility, the methodology of IDBFuzz is broadly applicable, as it is based on the idea of separating the IR skeleton from data generation, which allows abstraction of general operation flows while flexibly substituting specific data. The challenge lies in the heterogeneous operational semantics of different NoSQL databases, such as *keyPath* definition rules, transaction models, or support for nested structures, all of which require targeted adaptations and implementations to ensure correct migration of the approach. As future work, beyond exploring such cross-database application, our first step will be to investigate clearer boundaries and rules for aborting asynchronous programs, which remain vague and may currently waste computational resources.

Acknowledgment

This work was partly supported by the National Natural Science Foundation of China (NSFC) (grant nos. 62172194, 62202206, and U1836116), the Natural Science Foundation of Jiangsu Province (grant no. BK20220515), and the Graduate Research Innovation Project of Jiangsu Province (Grant number: KYCX23_3676).

REFERENCES

[1] J. Lin, Q. Zhang, J. Li, C. Sun, H. Zhou, C. Luo, and C. Qian, "Automatic library fuzzing through api relation evolvement," in *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS 2025)*, 2025.

[2] Z. Hatfield-Dodds and D. Dygalo, "Deriving semantics-aware fuzzers from web api schemas," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 345–346.

[3] G. Duan, H. Zhao, M. Cai, J. Sun, and H. Chen, "Dfl: A dom sample generation oriented fuzzing framework for browser rendering engines," *Information and Software Technology*, vol. 177, p. 107591, 2025.

[4] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé *et al.*, "Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases." in *NDSS*, 2021.

[5] 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*, 2022, pp. 1135–1147.

[6] I. Fratric, "Dom fuzzer," https://github.com/googleprojectzero/domato, accessed: 2025.

[7] 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*, 2020, pp. 971–986.

[8] 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)*, 2023, pp. 1883–1899.

[9] S. Kim, Y. M. Kim, J. Hur, S. Song, G. Lee, and B. Lee, "Fuzzorigin: Detecting uxss vulnerabilities in browsers through origin fuzzing," in *31st usenix security symposium (usenix security 22)*, 2022, pp. 1008–1023.

[10] 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*, 2024, pp. 690–704.

[11] Mozilla Developer Network, "Indexeddb api," https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API, accessed: 2025.

[12] W3C, "Indexed Database API 3.0," https://www.w3.org/TR/IndexedDB/, accessed: 2025.

[13] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, "Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities." in *NDSS*, 2023.

[14] D. A. Lamb, "Idl: Sharing intermediate representations," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 297–318, 1987.

[15] W3C, "Web idl," https://htmlspecs.com/webidl/, 2021, accessed: 2025.

[16] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.

[17] M. G. Azar, I. Osband, and R. Munos, "Minimax regret bounds for reinforcement learning," in *International conference on machine learning*. PMLR, 2017, pp. 263–272.

[18] Google, "Leveldb," https://github.com/google/leveldb, 2011, accessed: 2025.

[19] D. R. Hipp, "Sqlite," https://sqlite.org, 2024, version 3.x, Accessed: 2025.

[20] C. Zhou, Q. Zhang, L. Guo, M. Wang, Y. Jiang, Q. Liao, Z. Wu, S. Li, and B. Gu, "Towards better semantics exploration for browser fuzzing," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 604–631, 2023.

[21] 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*, 2024, pp. 1174–1185.