Characterizing and Taming Non-deterministic Bugs in JavaScript Applications

Jie Wang
State Key Lab of Computer Science, Institute of Software, Chinese Academy of Sciences, China
University of Chinese Academy of Sciences, China
wangjie12@otcaix.iscas.ac.cn

***Abstract*—JavaScript has become one of the most popular programming languages for both client-side and server-side applications. In JavaScript applications, events may be generated, triggered and consumed non-deterministically. Thus, JavaScript applications may suffer from non-deterministic bugs, when events are triggered and consumed in an unexpected order. In this proposal, we aim to characterize and combat non-deterministic bugs in JavaScript applications. Specifically, we first perform a comprehensive study about real-world non-deterministic bugs in server-side JavaScript applications. In order to facilitate bug diagnosis, we further propose approaches to isolate the necessary events that are responsible for the occurrence of a failure. We also plan to design new techniques in detecting non-deterministic bugs in JavaScript applications.**

Keywords—JavaScript, Node.js, non-deterministic bug, empirical study, record and replay, bug detection

I. Introduction

JavaScript has become the most popular language for client-side web applications. According to recent surveys from Stack Overflow [1] and Node.js foundation [2], JavaScript is also surpassing the popularity of other programming languages (e.g., PHP and Java) in server-side applications. For example, Node.js has become a popular server-side JavaScript framework.

In both client- and server- sides, JavaScript adopts event-driven architecture. In this architecture, the events may be generated, triggered and consumed non-deterministically, causing bugs that happens under a special event order. We call such bugs as non-deterministic bugs in this proposal. Non-deterministic bugs can result in unreliable and unpredictable application behaviors, such as crashes in server-side applications and wrong outputs in client-side applications.

Non-deterministic bugs in JavaScript applications have not been well addressed by existing work despite their importance. First, as far as we know, no comprehensive study has been performed about non-deterministic bugs for JavaScript applications, so we cannot well understand non-deterministic bugs' characteristics. Second, there is still a lack of effective tools for taming non-deterministic bugs for JavaScript applications. For example, in debugging, record-replay techniques, e.g., Mugshot [3], can be used to record non-determinism in JavaScript applications, and facilitate failure diagnosis. However, record-replay techniques usually record a long event trace, and it is time-consuming and exhausting to debug with such a long event trace. For non-deterministic bug detection, there is no effective tool that can detect atomicity violation among events. Although the fuzz testing technique [4] is proposed to increase the possibility to expose non-deterministic bugs in Node.js, there is still a lot of room for better approaches to combat non-deterministic bugs.

In this proposal, we target at characterizing and combating non-deterministic bugs in event-driven JavaScript applications. Specifically, we address the following research problems. First, for better understanding non-deterministic bugs, we performed the first *comprehensive study on real world non-deterministic bugs in server-side Node.js applications* [5]. We have carefully studied 57 real-world bug cases from open-source Node.js applications, and analyzed their bug characteristics, e.g., bug patterns, fixing strategies, etc. Second, to facilitate failure diagnosis, we eliminate the side effects of non-determinism by recording the event trace that results in a failure, and then reduce the original event trace to a smaller event trace that is responsible for the occurrence of a failure. In this respect, we present a *dynamic slicing based approach* [6] and a *context-based approach* [7], respectively. Third, we are developing a tool for *detecting atomicity violation bugs* which are very common in Node.js applications. We have proposed several novel patterns to identify atomicity violation bugs.

II. Related Work

Bug studies. Existing bug studies in JavaScript mainly focus on client-side JavaScript applications [8][9]. However, these studies do not cover non-deterministic bugs in JavaScript applications. The study [8] observes that non-deterministic errors are common in web applications, but it does not give an analysis on non-deterministic bugs. Although Node.fz [4] provides an initial study on a small set of non-deterministic bugs (only 12 bugs) in Node.js, we do not know whether the characteristics from these 12 bugs can be generalized to more non-deterministic bugs. There are some studies on concurrency bugs in multi-threaded [10] and distributed systems [11]. However, non-deterministic bugs in server-side JavaScript applications differ from those in traditional systems as they originate from different programming paradigms and execution environments.

Concurrency bug analysis. In recent years, much research effort has been devoted to non-deterministic bugs in event-driven applications, e.g., Android [12][13], client-side [14][15] and server-side web applications [16]. Our study shows that non-deterministic bugs in Node.js have different characteristics in bug patterns, manifestations and fix strategies. Thus, those approaches may be ineffective for Node.js.

Event trace reduction. Dynamic slicing [17] is effective in simplifying traces which computes the dynamic slice by building reachability graph. However, dynamic slicing faces DOM-specific challenges in JavaScript applications. Delta debugging [18] is commonly used for trace minimization and fault isolation. HDD [19] takes advantage of the hierarchical structure of the input, such as XML, to speed up delta debugging. However, HDD does not help our case since the input of the event trace reduction (i.e., event trace) is not structured. Hamouddi et al. [20] adapts delta debugging [18] to reduce event traces for web applications. However, it needs to try many infeasible candidate traces, and is inefficient.

III. Accomplished Work

In this section, we introduce our approaches and results for our accomplished work. First, we perform a comprehensive study on 57 real world bugs in server side JavaScript applications (i.e., Node.js) for better understanding of non-deterministic bugs caused by event ordering (Section III.A). Second, to help facilitate diagnosing non-deterministic bugs, we eliminate the side effects of non-determinism by recording the event trace that results in a failure, and output a shortened event trace that can still reproduce the original failure. To this end, we propose a dynamic slicing based approach (Section III.B). Third, we find that some events are still irrelevant although dynamic slicing approach cannot remove them. Thus, we further propose a context based approach to remove failure-irrelevant events (Section III.C). Although the proposed dynamic slicing based and context based approaches are now implemented for client-side JavaScript applications, they are also applicable for server-side Node.js applications.

A. *NodeCB: A Comprehensive Study on Real World Non-deterministic Bugs in Node.js*

Node.js, as a server-side JavaScript platform, is relatively new, and little is known about non-deterministic bugs in real world Node.js applications. Therefore, we aim to conduct a comprehensive study on such non-deterministic bugs in Node.js applications. Our study tries to answer the following research questions:

- RQ1 (Bug patterns and root causes): What are common bug patterns of non-determinisitic bugs in Node.js? What are their root causes?
- RQ2 (Bug impacts): Do non-determinisitic bugs have severe failure symptoms? What impacts do they have in Node.js applications?
- RQ3 (Bug manifestation): How do non-determinisitic bugs manifest themselves in Node.js? How are non-determinisitic bugs triggered, e.g., the timing condition, input conditions?
- RQ4 (Bug fix strategies): How do developers fix non-determinisitic bugs in Node.js applications? Are there any common fix strategies?

In order to collect sufficient bugs for our study, we directly collect non-determinisitic bugs from all Node.js projects in GitHub. First, we use concurrency-related keywords "concurrent", "race", "synchronization", "atomic", "mutex", "transaction", "deadlock", "compete" and "starve", which is a union set of existing work [11][21][22], to search candidate non-determinisitic bugs. We aim to filter out the issues that are about code written in JavaScript, marked as bugs, already in closed state, and have fixing solutions. We obtain 1,583 bugs after this step. Then we manually check these bug reports and finally collect 57 real world non-determinisitic bugs in 53 open-source Node.js projects. The involved projects are popular (avg. 2,426 stars), well-maintained (avg. 1,516 revisions and 1,152 issues), complicated (avg. 10,390 lines of code) and from various categories (including 12 server-side applications, 6 desktop applications, and 35 libraries).

Result. We thoroughly study these 57 non-determinisitic bugs, and answer the above four research questions. Our main findings are as follows:

- Non-determinisitic bugs in Node.js can be caused by atomicity violation (65%), order violation (30%), and starvation (5%). Existing work mostly focuses on order violation in event-driven applications [12][13][14]. This suggests that more research should be conducted on atomicity violation in Node.js.
- Most non-determinisitic bugs contend against shared variables (54%), databases (26%) and files (14%). This suggests that, besides shared variable [12][13], non-determinisitic bug detection approaches should pay more attention to shared resources like databases and files.
- APIs in Node.js are written in an asynchronous and event-driven way. They usually have unclear API protocols (e.g., event order and atomicity specifications). The specifications are violated in 28 (49%) of studied non-determinisitic bugs, indicating that developers may misunderstand the specifications of asynchronous APIs.
- Most non-determinisitic bugs were fixed by a small set of fix strategies. But, only a small portion (23%) of bugs were fixed by simply adding synchronization (e.g., nested callbacks), which is the main approach used by existing automated bug fixing approaches. This indicates that further automated bug fixing approaches are needed.

B. *JSTrace: Fast Reproducing Web Application Bugs*

Record-replay techniques, e.g., Mugshot [3], can be used to record non-determinism in client-side JavaScript applications, and reproduce a failure. However, the recorded event trace can be very long after a long run, and lots of events in the recorded event trace may be irrelevant to the failure. Debugging with such a long trace is exhausting and time-consuming. A recent study shows that a shortened event trace can significantly increase programmers' efficiency in failure diagnosis and fault localization. We propose a dynamic slicing based approach, *JSTrace*, to remove events irrelevant to the occurrence of the failure.

We have two key insights to remove failure-irrelevant events. (1) If a recorded event never triggers any listener registered by users, we can safely remove it. (2) If an event *e* does not affect the variables that will be used by the erroneous event's handler directly or indirectly, then *e* can be removed. To determine whether a variable may affect the variables in the erroneous event's handler, we provide novel models to abstract the JavaScript and DOM manipulation instructions to precisely capture data dependencies. We also build the dependency between JavaScript instructions and DOM instructions. Based on the captured data dependencies, we build dependency relationships between events and based on this, our slicing algorithm searches for the failure related events which are depended by the failure directly or indirectly.

Result. We have implemented the tool JSTrace, which is an enhanced record-replay tool that can significantly remove irrelevant events while keeping the trace reproducible. We have evaluated JSTrace on 10 real-world bugs from 7 popular web applications that belong to different domains. The result shows that all 10 bugs are successfully replayed, and can remove most of the failure-irrelevant events (96%).

C. *EvMin: Context-Based Event Trace Reduction*

JSTrace adopts dynamic slicing to trace the precise program dependence and discards the events that are not depended by a failure. However, not all remaining events in JSTrace are necessary to reproduce the failure. E.g., suppose an original event trace: e_1: a=1; e_2: a=a+1; e_3: if(a>0) throw new Error(). The event e_2 is failure-irrelevant since {e_1, e_3} can faithfully reproduce the failure.

Delta debugging [20] can be used to minimize failure-inducing events. However, delta debugging is slow because it may generate lots of infeasible candidate subtraces that may trigger syntactical errors (e.g., ReferenceError). We observe that, if an event in the reduced subtrace can be replayed, its context (i.e., variable usage information, such as the existence of its accessed DOM, the type of accessed variables) should keep the same as its corresponding event in the original trace. Based on this observation, we propose *EvMin*, an effective and efficient approach to remove failure-irrelevant events from an event trace.

EvMin iteratively generates a candidate subtrace of the original event trace and reruns the subtrace until it finds a failure-reproducible trace (i.e., a trace that can reproduce the original failure). In order to effectively and efficiently find a failure-reproducible trace, we utilize the context (i.e., preconditions) of each event to guide the generation of subtraces. If an event *e* is selected in a candidate subtrace, we require that the context of an event in a candidate subtrace should keep compatible with the context of its corresponding event in the original trace. In this way, we can generate shorter traces and avoid generating syntactically-infeasible candidate event traces. Specifically, for an event e_i in the original trace, if it is selected (marking it as e_i') in the candidate event trace, we say e_i and e_i' have compatible contexts if the following two conditions are satisfied: (1) All the variables used by e_i and e_i' are declared in the same scope with respect to their corresponding traces. (2) All the variables used by e_i and e_i' have the same type with respect to their corresponding traces. Consider that the trace of needed events for a failure is usually short [23] (i.e., usually no more than 6), EvMin generates candidate traces from short to long so that it can find the failure-reproducible trace earlier.

Result. We implemented EvMin for client-side JavaScript applications. We evaluated EvMin on 10 real-world bugs from 7 popular web applications that belong to different domains in terms of effectiveness and performance. We also compared EvMin with delta-debugging [20] in terms of time and generated event traces. The result shows that EvMin can remove all irrelevant events for the evaluated subjects. The performance of EvMin is acceptable. EvMin generates 72% less event traces, and costs 84% less time overhead than delta debugging.

IV. Planned Work

A. *Event Trace Reduction in Node.js Applications*

Non-deterministic bugs in server-side applications usually result in severe impacts according to our previous study (Section III.A). Besides, server-side applications usually run for a very long time and lead to a long event trace, and the events that are caused by different requests may be heavily interleaved. Thus, identifying the relevant events that are responsible for a failure and reproducing it to developers is very important for server-side applications. We plan to develop dynamic slicing and delta debugging approaches, and apply them on server-side JavaScript applications. We plan to carefully redesign the record-replay tool for Node.js and tackle the complicated event model of Node.js. We also need to design new approaches to make it scale to a long trace.

Envisioned Result. We plan to evaluate our solution on popular real-world open-source Node.js applications. The approach is expected to be effective and yet efficient in reducing event traces in Node.js applications, especially long traces.

B. *Atomicity Violation Bug Detection in Node.js*

According to our previous study, about two thirds of bugs are atomicity violation (i.e., AV), which indicates an urgent need for tools to detect atomicity violation bugs. The main challenges to detect atomicity violation bugs in Node.js are as follows: (1) The atomicity intention (i.e., some events should be processed uninterruptedly) is unknown. (2) In Node.js, condition variables are widely used to check whether some unexpected events have been processed, or store program states that can be used to recover from unexpected event processing. However, these condition variables can introduce many false positives for bug detection.

We are now developing AVDetector to detect atomicity violation bugs. The reason why atomicity bugs are common is that developers often use cooperative multitasking and partition their operations into multiple asynchronous steps to avoid blocking the event loop, leading to callback chains. Our solution is inspired by the observation that most atomicity violation bugs happen due to a false assumption of atomicity across callback chains. The interleaving between callback chains are very likely to introduce atomicity violation bugs. Thus, we plan to propose a pattern-based approach to identify atomicity violation bugs. For example, for a read and write operated on the same variable *v* in the callbacks caused by a user request, the variable *v* cannot be operated by the callback that caused by another user request. If two asynchronous operations operate on the same resource, e.g., a file, no other operations should operate on the same file. These patterns certainly do not cover all kinds of AV bugs. We plan to propose more patterns to cover more AV bugs.

Envisioned Result. We plan to evaluate AVDetector on popular real-world open-source Node.js applications, and bugs studied in our empirical study. It is expected that AVDetector can effectively detect new non-deterministic bugs.

V. CONCLUSION

Non-deterministic bugs are common in JavaScript applications. In this proposal, we firstly give a conprehensive study about non-deterministic bugs in server-side JavaScript applications. Further, to tame non-deterministic bugs, we propose a dynamic slicing based approach and a context-based approach to reduce/minimize a given event trace for client-side applications, so that developers can use a short event trace for failure diagnosis. We also plan to propose techniques to detect atomicity violation bugs in server-side JavaScript applications.

ACKNOWLEDGMENTS

This work was supported by National Key Research and Development Plan (2016YFB1000803), National Natural Science Foundation of China (61672506, 61702490), and Beijing Natural Science Foundation (4164104).

REFERENCES

[1] "Developer Survey Results 2016." [Online]. Available: http://stackoverflow.com/research/developer-survey-2016.

[2] "New Node.js Foundation Survey Reports New 'Full Stack' In Demand Among Enterprise Developers." [Online]. Available: https://nodejs.org/uk/blog/announcements/nodejs-foundation-survey/.

[3] J. Mickens, J. Elson, and J. Howell, "Mugshot : Deterministic Capture and Replay for JavaScript Applications," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation(NSDI)*, 2010, pp. 159–174.

[4] J. Davis, A. Thekumparampil, and D. Lee, "Node.fz: Fuzzing the Server-Side Event-Driven Architecture," in *Proceedings of the European Conference on Computer Systems(EuroSys)*, 2017, pp. 145–160.

[5] J. Wang, W. Dou, G. Yu, G. Chushu, Q. Feng, and W. Jun, "A Comprehensive Study on Real World Concurrency Bugs in Node.js," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[6] J. Wang, W. Dou, C. Gao, and J. Wei, "Fast Reproducing Web Application Errors," in *Preceedings of International Symposium on Software Reliability Engineering(ISSRE)*, 2015, pp. 530–540.

[7] J. Wang, "Constraint-Based Event Trace Reduction," in *Preceedings of ACM SIGSOFT International Symposium on the Foundations of Software Engineering(FSE16-SRC)*, 2016, pp. 1106–1108.

[8] F. S. Ocariza, K. Pattabiraman, and B. Zorn, "JavaScript Errors in the Wild: An Empirical Study," in *Proceedings of International Symposium on Software Reliability Engineering(ISSRE)*, 2011, pp. 100–109.

[9] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "A Study of Causes and Consequences of Client-Side JavaScript Bugs," *IEEE Transactions on Software Engineering (TSE)*, vol. 43, no. 2, pp. 128–144, 2016.

[10] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2008, pp. 329–339.

[11] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "TaxDC:A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2016, pp. 517–530.

[12] P. Bielik, V. Raychev, and M. Vechev, "Scalable Race Detection for Android Applications," in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA)*, 2015, pp. 332–348.

[13] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race Detection for Event-driven Mobile Applications," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 326–336, 2014.

[14] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race Detection for Web Applications," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 251–262.

[15] W. Wang, Y. Zheng, P. Liu, L. Xu, X. Zhang, and P. Eugster, "ARROW : Automated Repair of Races on Client-Side Web Pages," in *Proceedings of the International Symposium on Software Testing and Analysis(ISSTA)*, 2016, pp. 201–212.

[16] Y. Zheng and X. Zhang, "Static Detection of Resource Contention Problems in Server-Side Scripts," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012, pp. 584–594.

[17] X. Zhang, S. Tallam, and R. Gupta, "Dynamic Slicing Long Running Programs Through Execution Fast Forwarding," in *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2006, pp. 81–91.

[18] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-inducing Input," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 2, pp. 183–200, 2002.

[19] G. Misherghi and Z. Su, "HDD: Hierarchical Delta Debugging," in *Proceedings of the International Conference on Software Engineering(ICSE)*, 2006, pp. 142–151.

[20] M. Hammoudi, B. Burg, G. Bae, and G. Rothermel, "On the Use of Delta Debugging to Reduce Recordings and Facilitate Debugging of Web Applications," in *Proceedings of Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software (ESEC/FSE)*, 2015, pp. 333–344.

[21] G. Pinto, W. Torres, and F. Castor, "A Study on the Most Popular Questions about Concurrent Programming," in *Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools*, 2015, pp. 39–46.

[22] M. Yu, Y.-S. Ma, and D.-H. Bae, "Characterizing Non-deadlock Concurrency Bug Fixes in Open-source Java Programs," in *Proceedings of the Annual ACM Symposium on Applied Computing(SAC)*, 2016, pp. 1534–1537.

[23] G. Li, E. Andreasen, and I. Ghosh, "SymJS: Automatic Symbolic Testing of JavaScript Web Applications," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering(FSE)*, 2014, pp. 449–459.