Enhancing Web App Execution with Automated Reengineering

Kijin An
Supervisor: Dr. Eli Tilevich
Software Innovations Lab, Virginia Tech
ankijin@vt.edu

ABSTRACT

The execution of modern web applications is affected by distribution, mobility, and heterogeneity. The design-time assumptions of web applications rarely correspond to their runtime conditions. As a result, the efficiency, performance, and reliability of web app execution can suffer. This dissertation research addresses this problem by introducing novel automated reengineering techniques. In particular, we put forward a series of novel domain-specific refactorings, semantics-preserving program transformations, that behind-the-scenes improve the efficiency, performance, and reliability of extant web applications. To that end, we apply state-of-the-art program analysis and transformation techniques, extending and adapting them for the domain of web applications. Our ultimate objective is to enhance the execution of real-world web apps to meet the requirements of modern users, so the web can remain the most versatile computing application deployment and delivery infrastructure. The intermediate results of this dissertation research have been accepted for publication as a full research paper in the proceedings of the Web Conference 2020 [3].

KEYWORDS

Software Engineering, Reengineering, Web Applications, JavaScript, Mobile Apps, Program Analysis & Transformation, Middleware

ACM Reference Format:
Kijin An. 2020. Enhancing Web App Execution with Automated Reengineering. In *Companion Proceedings of the Web Conference 2020 (WWW '20 Companion), April 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3366424.3382087

1 PROBLEM

The web has become the most important infrastructure for deploying and delivering computing applications. Web application execution spans across cloud-based servers and, typically, mobile clients that communicate with each other over the network. When designing and implementing web applications, developers make assumptions about which execution conditions would be present at runtime. If these assumptions hold true, then the applications perform efficiently and reliably. However, it is a confluence of servers, clients, and the networks connecting them that can affect web application execution, while predicting the exact combination of these runtime parameters is hard if not impossible. Hence, when design-time assumptions fail to match the actual runtime conditions, web application execution becomes plagued with performance, efficiency, and reliability problems. In addition, edge computing resources are being increasingly adopted to address sensor data deluge and the resulting bandwidth limitations of wide-area networks. However, existing web applications are not designed to take advantage of edge-based resources, thus missing out on the performance optimization opportunities these resources provide.

For example, a mobile web app can be engineering to invoke a remote service that executes some application functionality. Because the computational resources of cloud-based servers are superior to those of mobile devices, remote execution offers performance advantages, as long as the overhead of network communication is not prohibitively expensive. However, if the app is executed over a poor network, the communication overhead can offset the performance benefits of cloud-based processing, with the resulting performance becoming unsatisfactory. To adapt app execution unreliable or limited networks, the app can be reengineered to be able to execute the remote service locally, so the service could be invoked remotely or locally depending on the network connection currently in place.

However, reengineering web applications is non-trivial, as achieving efficient and reliable execution requires that applications adapt to dissimilar networks and devices. Furthermore, the majority of advanced software engineering techniques and tools are designed for centralized applications that execute on a single machine. Therefore, developers are faced with the necessity to enhance the execution of web applications in complex ways, but have only inadequate software engineering tools in their disposal.

For example, refactoring, the most common semantics-preserving program transformation, has been applied mostly to centralized applications. It would be impossible to apply known refactoring transformations to a typical web application, distributed across the web. That is, one can apply existing refactorings separately to the client and server parts of an app, but the resulting transformations are not guaranteed to preserve the semantics of the app's distributed execution spanning the client and server sites.

2 STATE OF THE ART

Improving various aspects of web application execution has been the target of numerous prior research efforts. Since our work applies novel software engineering approaches, techniques, and tools, we limit our coverage of the related state of the art to this area.

Our reference implementation of Client Insourcing, relates to advanced program analysis techniques for JavaScript, due to its target domain—cross-platform mobile applications. Our approach follows declarative program analysis frameworks that statically analyze JavaScript code by means of a constraints solver [16, 24, 27, 37]. The JavaScript language constructs for programming event-based applications that wait for dispatches events or message asynchronously.

Some advanced static analysis approaches apply formal reasoning for *callback* and *promises* of web apps based on a calculus [28, 29]. Existing dynamic analysis tools [24, 33] have a scalability problem to analyze entire JavaScript program. Dynamic symbolic execution (DSE) symbolically executes a JavaScript program by applying concrete input values [31] for a faster analysis. For more advanced DSE, MultiSE [34] uses a value summary in Jalangi2 to effectively generate testing input values of a JavaScript program to speed up dynamic symbolic execution.

In JavaScript, choosing one programming implementation over another can make a significant difference performance effect on the overall application [14, 15]. An approach presented in [32] empirically identified reappearing patterns of inefficient JavaScript programs in open source community, so common performance bottlenecks can be automatically detected and fixed by using software engineering techniques.

Some prior approaches automatically change the locality of execution in existing application, a highly complex process, as centralized and distributed execution models differ from each other in terms of their respective latency, concurrency, and failure modes [39]. Researchers and practitioners have greatly studied to facilitate the task of rendering local functionalities remote to take advantage of remote resources. For example, offloading local functionality to the cloud has also been supported as an automated refactoring technique [19, 20, 40, 41].

Client Insourcing can be seen as a variant of program synthesis [6–9, 11, 12, 17, 35], an active research area concerned with producing a program that satisfies a given set of input/output relationships. CodeCarbonReply and Scalpel [5, 36] integrate portions of a C/C++ program's source in another C/C++ program by leveraging advanced program analysis techniques. The programmer's effort is only limited to annotate the code regions to integrate, and then the tool automatically adapts the receiving application's code to work seamlessly with the moved functionality. Client Insourcing belongs to a category of refactoring transformations that change the locality of application components for various reasons. One prominent direction in this research is *application partitioning*, which is an automated program transformation that transforms a centralized application into its distributed counterpart [5, 25, 26, 36, 38]. Another approach that leverages compiler-based techniques is the ZØ compiler [13], which automatically partitions CSharp programs into distributed multi-tier applications by applying scalable zero-knowledge proofs of knowledge, with the goal of preserving user privacy.

Several middleware-based approach has been proposed to reduce the costs of invoking remote functionalities. APE [30] is an annotation based middleware service for continuously-running mobile (CRM) applications. APE defers remote invocations until some other applications switch the device's state to network activation. Similarly, to reduce the overhead of HTTP communication, events for HTTP requests in Android apps are automatically are bundled into a single batched network transmission [22, 23]. The e-ADAM middleware [21] optimizes energy consumption by dynamically changing various aspects of data transmission scheme. My approach can also adapt the realities of executing full-stack JavaScript mobile apps to optimize the remote execution with a cost function.

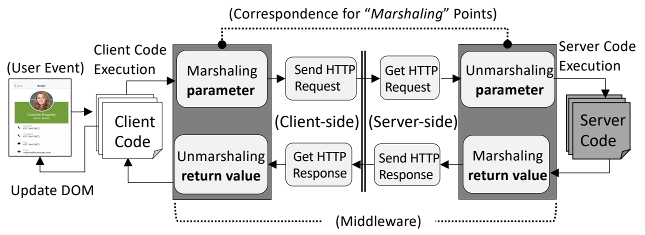

Figure 1: Web App's Execution Model

3 APPROACH

This dissertation research is concerned with creating novel automated program transformations that enhance the execution of web applications. Refactoring is an automated program transformation that changes the source code of an application while preserving its semantics. The key novelty of this research is a novel domain-specific refactoring —*Client Insourcing*—that transforms a web application into its semantically equivalent centralized variant, in which the remote parts are glued together and communicate with each other by means of regular function calls. The equivalent variant is then for adaptive and perfective modifications and subsequent automated redistribution, thereby enhancing web app execution, with a particular emphasis on performance, energy efficiency, and reliability.

Our technical contributions lie in addressing the substantial technical complexities of implementing the Client Insourcing Refactoring, so it could become a viable addition to the toolset of web developers. As our primary target, we focus on full-stack JavaScript applications, in which both the client and server parts are written in this language. The monolingual nature of such applications lowers the implementation burden, so the same techniques and tools can be applied to all application parts.

Client Insourcing automatically moves all remote functionalities to run locally, removing all middleware functionality, so the resulting centralized equivalent can serve as a convenient proxy for all corrective, perfective, and adaptive modifications using proven existing techniques and tools [1]. Several technical obstacles stand on the way of realizing Client Insourcing as a viable software engineering technique. First, in distributed web apps, written in scripting languages, it is often hard to precisely determine all the dependent code of a remote functionality. All the dependencies have to be identified, even if they are scattered across multiple implementation units. In addition, typical server-side code of web applications makes heavy us of third-party libraries and frameworks that are specific to the server, such as server-side persistent storage. These server-side dependencies require special processing and adaptation to be able to to migrate them to the client.

Second, remote functionalities are invoked by means of distribution middleware, which exposes APIs whose protocols and conventions differ from invocations in the same address space. These APIs tend to differ widely depending on their vendor: the JavaScript ecosystem features numerous dissimilar distribution frameworks and libraries. Rather than encoding domain-specific preconditions, our approach uses domain agnostic semantic to identify the entry/exit points of middleware functionality.

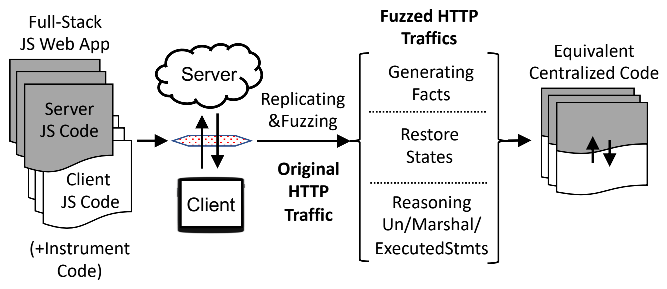

Figure 2: Overall Process for JS-RCI [3]

One of the key functions of a middleware system is marshaling and unmarshaling method parameters and return values. Marshaling converts program values to a data format suitable for transmission; unmarshaling reverses the process by converting the transmitted data format to regular program values (See Figure 1). Our approach captures and replays HTTP traffic to analyze the marshaled parameters and unmarshaled return values from the client and the server, respectively.

4 METHODOLOGY

We follow the software engineering methodology of creating a novel domain-specific refactoring and demonstrating its value, applicability, and limitations through a series of case studies of enhancing web application execution. Our new domain-specific refactoring—Client Insourcing—is realized as a reference implementation, called JavaScript Remote Client Insourcing (JS-RCI) [3]. JS-RCI integrates advanced software engineering techniques, such as program synthesis, which identifies which application functionality satisfies a given remote execution's input and output relationship. JS-RCI also makes use of constraint solving to extract JavaScript server-side functions. The constraint solving functionality is based on declarative logic-based programming, which represents JavaScript program statements and their relationships to each other as logical facts and predicates. As a specific example, consider the relationship that expresses the transitive dependency across the statements, executed between the entry and exit points of a remote service:
$ExecutedStmts(stmt_n, val_{umar}, val_{mar}) \leftarrow$

$$(DataDep(stmt_n, stmt_1) \wedge Marshal(stmt_1, v_1, val_{mar})) \wedge$$
$$(\neg DataDep(stmt_n, stmt_2) \wedge UnMarshal(stmt_2, v_2, val_{umar})).$$

To determine such entry/exit points, JS-RCI instruments the app's source code statements, which implement HTTP commands, as well as applies advanced testing techniques (e.g., *fuzzing*). Specifically, to identify these points with minimal false-negatives, JS-RCI fuzzes the original HTTP traffic by padding the HTTP header and body data with random data. Since many of these testing techniques requires that the subject be executed repeatedly in a deterministic way, JS-RCI also makes it possible for stateful servers to be executed idempotently by automatically undoing their state changes. Overall process of JS-RCI is shown in Figure 2.

5 RESULTS

We have successfully applied JS-RCI to numerous real-world apps that use different middleware frameworks to implement their client/server (Tier I/Tier II) communication and different third-party libraries for database operations (Tier III).

- **Tier I:** JQuery, Ajax, fetch, axios, AngularJS/TS
- **Tier II:** Express, koa.js, and Restify
- **Tier III:** MySql, Postgres, and knex.js

In the following discussion, we report on the initial results of applying JS-RCI to enhance various aspects of web app execution.

5.1 A Technique for Debugging Web Apps

One application of Client Insourcing is to facilitate the debugging of web apps. The underlying idea is to be able to debug the semantically equivalent centralized variant of a faulty app. Our tool CanDoR [2] applies Client Insourcing to create a centralized app variant that can be debugged by means of the numerous existing state-of-art debugging tools for centralized JavaScript programs. Once the programmer fixes the bug in the centralized variant version with these tools, CanDoR applies the resulting fixes to the actual client and server parts of the original web application. To that end, CanDoR automatically generates input scripts for GNU Diff, which executes these scripts against the source files of the original full-stack JavaScript application by using GNU patch. For certain types of bugs, this debugging approach proves to be effective and reducing the required debugging efforts.

CanDoR reduces the complexity of debugging web apps. It makes it possible to debug the app's business logic locally, so the app can be executed repeatedly, with the debugging output examined in place. As a result, the required debugging effort decreases. For example, CanDoR reduces the time taken to execute a debugged functionality by more than **90%** on average. Given that debugging typically involves repeated execution, having much faster subjects to debug improves the the debugging process's efficiency.

5.2 Optimizing Remote Execution Granularity

Another application of Client Insourcing is restructuring the distribution of web apps to maximize their performance and efficiency. Our approach, named D-Goldilocks [4], follows *the Goldilocks Principle* [10, 18] to maximize the benefits of distribution. That is, the granularity of a remote service should not be too fine or too crude; it must be just right. D-Goldilocks re-architects web apps to adjust their distribution granularity as a means of improving performance and efficiency. To that end, using a semantically equivalent centralized application variant, D-Goldilocks profiles the variant's performance, with the following cost function $C(r)$, determining how to reshape the original distribution r to improve performance and efficiency.

$$C(r) = \alpha \cdot latency(r) + (1-\alpha) \cdot \sum resource(r).$$

Extant automatic refactorings (e.g., *partition* and *batch*) are then applied to reshape and redistribute the original remote functionality into a service with a changed granularity. In the end, D-Goldilocks identifies a distribution that minimizes the $C(r)$ function.

D-Goldilocks saves programming effort by automatically reshaping the granularity of the remote functionalities of web apps.

To to determine what the optimal combination of functions is, D-Goldilocks generates all possible combinations of individually invoked functions. For some subjects, D-Goldilocks can save almost 1.6×10^6 lines of code required to discover the optimal granularity.

5.3 Adapting Web Execution at Runtime

To achieve the best performance for all combinations of client and server devices and network connections, a web app would have to be distributed in a variety of versions. My ongoing research is concerned with adaptively redistributing functionalities of a full-stack mobile JavaScript app to optimize its performance and energy consumption. I have been working on a new framework, Communicating Web Vessels (CWV). In a CWV-enabled web app, the client dynamically instruments and monitors web's app execution for "$/service\_r$". In response to a deterioration in network conditions, the client can request that the server-side JavaScript code and program state of a remote service be moved to the client, so the service can be invoked locally as a regular function call ($insource$("$/service\_r$")). When the network conditions become favorable, the moved service starts to be invoked remotely again ($revert(r)$). Invoking remote services locally or remotely helps achieve the best possible response time of the web app under the current network conditions.

Preliminary results for CWV show that it effectively adapts web apps for dissimilar networks and two different mobile devices. CWV offers a low performance overhead, so CWV-enabled apps improve their responsiveness by adapting to the current runtime conditions. CWV-enabled web apps can also improve their energy efficiency by as much as **10%** for poor network conditions. As the longer total execution time still causes larger overall energy consumption even though the device switches into the low power mode during the idle states.

6 CONCLUSIONS AND FUTURE WORK

The ever-changing realities of the modern web as the infrastructure for deploying and delivering applications creates new challenges for ensuring that web applications provide a satisfactory user experience. In particular, modern users expect web applications to be responsive, reliable, and energy efficient. To meet these objectives, web application developers need powerful approaches, techniques, and tools. This dissertation research innovates in the software engineering space to create novel automated reengineering approaches that enhance web application execution. In particular, we introduce a novel domain-specific refactoring, supported by state-of-the-art program analysis and transformation techniques, and apply these refactoring to address some of the most salient problems of web app execution.

To complete this dissertation research, we plan to add the domain of edge applications to the list of the successfully accomplished case studies of our technologies. Although edge computing offers novel execution optimization opportunities, it would be non-trivial to adapt existing web applications to take advantage of edge-based resources. This research can be naturally applied to enable such adaptation, offering a unique technical perspective for solving this important problem of web app engineering.

ACKNOWLEDGMENTS

This research is supported by the NSF through the grants # 1650540 and 1717065.

REFERENCES

[1] Kijin An. 2019. Facilitating the Evolutionary Modifications in Distributed Apps via Automated Refactoring. In *Web Engineering*. Springer International Publishing, 548–553.
[2] Kijin An and Eli Tilevich. 2019. Catch & Release: An Approach to Debugging Distributed Full-Stack JavaScript Applications. In *Web Engineering*. 459–473.
[3] Kijin An and Eli Tilevich. 2020. Client Insourcing: Bringing Ops In-House for Seamless Re-engineering of Full-Stack JavaScript Applications. In *Proceedings of the Web Conference 2020*.
[4] Kijin An and Eli Tilevich. 2020. D-Goldilocks: Automatic Redistribution of Remote Functionalities for Performance and Efficiency. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering(SANER)*.
[5] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. ACM, New York, NY, USA, 257–269.
[6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 95–110.
[7] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. 2016. Sampling for Bayesian program learning. In *Advances in Neural Information Processing Systems*. 1297–1305.
[8] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 420–435.
[9] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 422–436.
[10] Norman E Fenton and Martin Neil. 1999. A critique of software defect prediction models. *IEEE Transactions on software engineering* 25, 5 (1999), 675–689.
[11] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 229–239.
[12] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining model learning and model checking to analyze TCP implementations. In *International Conference on Computer Aided Verification*. Springer, 454–471.
[13] Matthew Fredrikson and Benjamin Livshits. 2014. ZØ: An Optimizing Distributing Zero-Knowledge Compiler. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 909–924. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/fredrikson
[14] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. 357–368.
[15] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. 94–105.
[16] Salvatore Guarnieri and Benjamin Livshits. 2009. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code.. In *USENIX Security Symposium*, Vol. 10. USENIX, Montreal, Canada, 78–85.
[17] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.
[18] Les Hatton. 1997. Reexamining the fault density component size connection. *IEEE software* 14, 2 (1997), 89–97.
[19] Michael Hilton, Arpit Christi, Danny Dig, Michał Moskal, Sebastian Burckhardt, and Nikolai Tillmann. 2014. Refactoring local to cloud data types for mobile apps. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*. ACM, 83–92.
[20] Young-Woo Kwon and Eli Tilevich. 2014. Cloud refactoring: automated transitioning to cloud-based services. *Automated Software Engineering* 21, 3 (01 Sep 2014), 345–372. https://doi.org/10.1007/s10515-013-0136-9
[21] Young-Woo Kwon and Eli Tilevich. 2014. Configurable and adaptive middleware for energy-efficient distributed mobile computing. In *Proceedings of the 6th International Conference on Mobile Computing, Applications and Services (MobiCASE)*. IEEE, 106–115.
[22] Ding Li, Shuai Hao, Jiaping Gui, and William GJ Halfond. 2014. An empirical study of the energy consumption of Android applications. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 121–130.
[23] Ding Li, Yingjun Lyu, Jiaping Gui, and William GJ Halfond. 2016. Automated energy optimization of HTTP requests for mobile applications. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 249–260.
[24] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proceedings of the 22nd ACM*

SIGSOFT International Symposium on Foundations of Software Engineering (Hong Kong, China) *(FSE 2014)*. 449–459.

[25] Yin Liu, Kijin An, and Eli Tilevich. 2018. RT-Trust: Automated Refactoring for Trusted Execution Under Real-Time Constraints. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Boston, MA, USA) *(GPCE 2018)*. ACM, 175–187.

[26] Yin Liu, Kijin An, and Eli Tilevich. 2019. RT-Trust: Automated Refactoring for Different Trusted Execution Environments under Real-Time Constraints. *Journal of Computer Languages* (2019), 100939.

[27] Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Baltimore, MD).

[28] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A Model for Reasoning About JavaScript Promises. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 86 (Oct. 2017), 24 pages.

[29] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static Analysis of Event-driven Node.Js JavaScript Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, 505–519.

[30] Nima Nikzad, Octav Chipara, and William G Griswold. 2014. APE: an annotation language and middleware for energy-efficient mobile application development. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 515–526.

[31] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for JavaScript. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 513–528.

[32] M. Selakovic and M. Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 61–72.

[33] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, 488–498.

[34] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, 842–853.

[35] Jiasi Shen and Martin C Rinard. 2019. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 269–285.

[36] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. 2017. CodeCarbonCopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, 95–105.

[37] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. 2016. Static DOM Event Dependency Analysis for Testing Web Applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, 447–459.

[38] Eli Tilevich and Yannis Smaragdakis. 2002. J-Orchestra: Automatic Java Application Partitioning. In *ECOOP 2002 — Object-Oriented Programming*. Springer, 178–204.

[39] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. 1996. A note on distributed computing. In *International Workshop on Mobile Object Systems*. Springer, 49–64.

[40] Xudong Wang, Xuanzhe Liu, Ying Zhang, and Gang Huang. 2012. Migration and execution of JavaScript applications between mobile devices and cloud. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 83–84.

[41] Young woo Kwon and Eli Tilevich. 2013. Power-Efficient and Fault-Tolerant Distributed Mobile Execution *(ICDCS '13)*. IEEE.