Static Call Graph Construction in AWS Lambda Serverless Applications

Matthew Obetz, Stacy Patterson, Ana Milanova
Rensselaer Polytechnic Institute

Abstract

We present new means for performing static program analysis on serverless programs. We propose a new type of call graph that captures the stateless, event-driven nature of such programs and describe a method for constructing these new extended service call graphs. Next, we survey applications of program analysis that can leverage our extended service call graphs to answer questions about code that executes on a serverless platform. We present findings on the applicability of our techniques to real open source serverless programs. Finally, we close with several open questions about how to best incorporate static analysis in problem solving for developing serverless applications.

1 Introduction

Serverless computing is a model where developers pay to execute stateless, event-driven functions in the cloud [32]. This new paradigm offers highly elastic scaling for applications by executing application code in virtual containers that may be shared across multiple function invocations and moved seamlessly between physical servers [29]. As a result, serverless platforms encourage microservice-oriented designs, where serverless functions are deployed separately and accomplish tasks cooperatively. To accomplish this, these services pass messages and subscribe to notifications about events that occur on platform services, such as writes to databases or distributed filesystems [36].

The unique features of this platform have caused a rapid increase in interest from developers in industry who seek to reduce the overhead associated with maintenance and monitoring of traditional servers [3]. However, as developers have begun programming applications for the serverless platform, they have encountered a noticeable gap in tooling [40]. Common questions include how to trace executions during debugging, how to measure application performance, and how to verify program correctness and security. Program analysis seeks to answer these questions by building tools to investigate program behavior. While significant progress has been made toward answering these questions for traditional programs, the tools they have produced have not been extended to the domain of serverless computing. Recent work has suggested that static analysis may also be a beneficial tool for answering questions such as how to provision and allocate serverless resources [22].

Call graph construction is a necessary first step toward developing a wide range of tools for analysis of serverless applications. A call graph is a directed graph where each node represents a function, and each edge represents a point of control flow where one function invokes another [12]. Call graph construction is an instrumental part of many program analyses, such as those for identifying and eliminating unused code [23] and those for detecting potential software vulnerabilities [33] or malicious behavior [6], all of which can greatly aid in refining serverless code.

General approaches to call graph construction exist for the languages used in serverless programs. However, these approaches traverse a program from an identified entrypoint, generating summaries of encountered functions. Calls between functions are resolved using a set of rules that make simplifying assumptions about context or control flow. By contrast, serverless programs are fundamentally event-driven [31]. This means that approaches to generate an accurate call graph must be able to resolve not only function calls, but also resolve *implicit* state transfer that occurs, for example, when a function writes to a message queue or database service, and that service has an event trigger configured to execute more code once the write is completed. These call graph construction approaches must also be able to parse events, which are often declared in an external file with a platform-specific format. As such, standard approaches to building call graphs will generate an incomplete view of the application that fails to describe the higher-level interconnectivity of microservices.

To address this gap, we propose a new approach to call graph construction for serverless applications that augments graphs with information about relationships between serverless functions and the platform services with which they interact. We introduce the notion of an *extended service call graph*, which extends the traditional call graph to include new classes of nodes. These new nodes represent the platform services written to or read by application code to produce control flow that spans multiple disconnected parts of a program. To construct this more complete view of a serverless application, we describe novel means of statically capturing event declarations and locating method calls within programs that cause these events to fire, with focus on applications written in Javascript for the AWS Lambda platform. We choose Javascript as it is the most common language for AWS Lambda programs.

Specifically, the contributions of this work are:

- We propose a novel conception of call graphs that better fits the serverless paradigm and opens new opportunities for static analysis for the serverless platform.
- We describe how platform services that affect program control flow should be represented in the extended service call graph.
- We survey applications of static analysis that can provide useful information for serverless programs.
- We present preliminary data on the structure and feature set of real-world serverless programs written in Javascript, and organize these programs into a set of benchmarks for static analysis. Preliminary results suggest that static analysis is effective for the analysis of serverless programs.

Related work Without access to techniques for reasoning statically about serverless programs, some have opted to consider dynamic runtime analyses to visualize program structure [27, 28], track the flow of sensitive information [1], or measure resource costs [39]. These tools operate by injecting instructions into a program or modifying its runtime to instrument realtime monitoring that allows the analysis to collect information about the behavior of code [35]. Like all dynamic analyses, they require substantial modification and directed exercising of public interfaces to produce results. Since executing programs over an exhaustive set of inputs is generally impractical, these tools offer only a partial view of an application [7]. Perhaps more significantly, the code to execute the dynamic analysis must be colocated with the application on the cloud, resulting in significant execution costs to collect results for an analysis [8, 28]. This suggests that these analyses cannot serve as a general replacement for static analysis in situations where a more comprehensive view of application behavior is required.

The remainder of this paper is organized as follows. Section 2 provides background on serverless applications and call graphs. We discuss how call graphs are extended to support the new classes of nodes that we define in Section 3, then describe how this new model improves several applications of call graphs in Section 4. We establish benchmarks to evaluate our methods and future static analyses in Section 5, and we close with a discussion of open questions in Section 6.

2 Background

Serverless Architecture Serverless applications consist of a set of libraries that export publicly accessible functions. This collection of *serverless functions* is deployed to an environment managed by a serverless provider, who will ephemerally provision computational resources to execute these functions as they are invoked. To fully capitalize on the opportunities for scaling that this provides, developers typically divide their code into several *microservices*, where each serverless function operates as an independent, modular unit that communicates with other microservices by passing messages [38]. Because a microservice contains only the minimal code to execute its specific task, several serverless functions may be chained together to complete what appears externally as a single action.

Listing 1: Sample fragment of a Serverless Framework YAML configuration file. Each entrypoint of the program is named and given a handler that specifies a filepath and method to be invoked. The events collection for each method describes services that can trigger the handler. In this case, the entrypoints are `listenForUserInput`, which observes web requests, and `listenToSNSTopic`, which observes the Amazon Simple Notification Service.

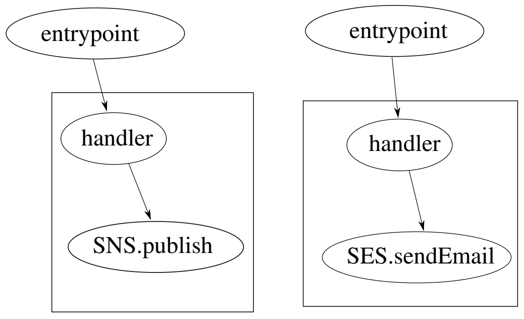
```
functions:
 listenForUserInput:
  handler: sourceFile.handler
   events:
    - http:
   path: /endpoint
   method: post
 listenToSNSTopic:
  handler: secondFile.handler
   events:
    - sns:
    arn:aws:sns:us-east:xxx:mytopic
```

To specify when serverless functions should be invoked, a configuration file is deployed alongside application code. This file identifies specific functions to act as handlers for one or more events. When one of these events takes place, the associated function is *triggered*. For AWS Lambda applications, two common formats for the configuration are CloudFormation templates [17] and Serverless Framework templates [21], both of which are YAML files that list a collection of resources that should be packaged for deployment. We present relevant portions of a sample YAML file for a serverless application in Listing 1 and a sample event handler in Listing 2.

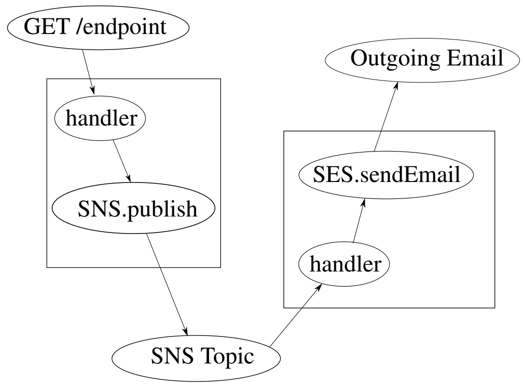
3 Defining Serverless Call Graphs

Suitability of Static Analysis. There are certain features of programming languages that interfere with the ability to create precise and efficient static analysis tools. As such, we need to better understand whether serverless programs regularly employ these features to determine if static analysis is well-suited to this domain. The following observations are based on study of real-world serverless programs that will be explored further in Section 5.

In particular, static analysis struggles to compute precise results efficiently when programs grow very large, or when



(a) Call graph applied to serverless functions individually.



(b) Extended service call graph.

Figure 1: Comparison of traditional call graph and extended service call graph, using the sample application from Listing 1. Pictured at left is the result of call graph analysis on each entrypoint individually; flow from the entrypoint to SDK libraries can be detected, but information about where those libraries may cause other serverless functions to execute is lost. At right we introduce nodes representing functionality hosted on AWS which interacts with code. Using the extended service call graph, it becomes possible to track flow of data from the source of user interaction with a public API (the GET /endpoint) to a sink where outgoing email is served through the Amazon Simple Email Service.

Listing 2: Sample fragment of serverless code from `sourceFile.js`, loaded by `listenForUserInput` in Listing 1. This sample accepts a web request and publishes its content to the "mytopic" Simple Notification Service topic. Note that the ARN of the SNS Topic is a string literal passed in to `publish()`, behavior which is common in serverless programs. This allows for static detection of the resource that is being accessed.

```
let aws = require("aws-sdk")
let sns = aws.SNS()

exports.handler = function(ev,ctx,r) {
 sns.publish({
  Message: ev.body,
  TopicArn:
    "aws:sns:us-east:xxx:mytopic"
 })
 }
```

a program loads a large number of external libraries. Fortunately, serverless platforms impose strict limits on function execution time, encouraging developers to ensure that calls made to external libraries are infrequent and not excessively deep. In practice, we have found that programs contain only a few hundred reachable lines of code for each individual serverless function. When applications grow more complex, this complexity is expressed by adding additional serverless functions, rather than expanding the few that exist. This size is well below even the limits placed by serverless providers, and suggests that microservice-oriented designs may be an even stronger influence on function length than platform limitations.

Static analysis likewise cannot analyze certain interpreted language features, such as when code is stored as a string and then loaded with an `eval()` function. However, in most serverless programs, the only unknown strings that could be evaluated in this way are function parameters provided directly by users. We find such dynamic behavior is uncommon in serverless programs, likely because few users are trusted enough to execute arbitrary code. From these observations, we conclude that static analysis has potential as a promising tool for reasoning about serverless applications.

Extended Service Call Graphs. Traditional callgraphs contain only a single type of node, representing functions that may be executed throughout the lifetime of an application [11]. Even when the concept of call graphs has been applied to microservice architectures, previous work has not considered representation of control flow that travels through external services [26]. Since this type of control flow is central to the serverless programming model, it is necessary to develop call graphs that explictly incorporate these external services.

We propose a new type of call graph for serverless applications that we call an *extended service call graph*. The extended service call graph includes a node for each function, but also nodes specific to the serverless platform. To date, these serverless-specific nodes include: 1) public API endpoints, 2) information stores (such as database tables, message queues, notification topics, and object storage buckets), 3) scheduled tasks, and 4) outgoing email. We aim to extend this list to accommodate the full range of serverless features.

We briefly describe how we construct the extended service call graph below. In the next section, we provide examples of how information may flow through services, and thus, why it is necessary to include this flow in call graphs.

To build the extended service call graph for Javascript AWS Lambda programs, we first traverse the configuration file provided with the application to identify the location of all functions that may serve as entrypoints to the program. We generate a call graph node for each of these entrypoints. In addition, for every defined event that may trigger a given function, we immediately generate a node representing this event identified by the Amazon Resource Name [34] included in the configuration. We then add an edge from the event to the associated entrypoint. Once this pre-processing is complete, we iterate a list of discovered entrypoints and perform call graph construction for each.

To capture the effect of third party libraries without expanding the analysis to include their full source code, we propose the implementation of transfer functions for public methods in the library. These functions provide a high level summary of the effects a library has without the complexity of a full implementation, allowing such stateful libraries to be precisely analyzed. We demonstrate an example of this simplification in Listing 3.

Listing 3: Summary functions for `ajv`, an imported external library. In its full implementation, `ajv` stores named JSON objects with the `addSchema` function, then validates that new objects have the same structure as a particular named schema with the `validate` function. Though the full implementation is stateful, we can summarize the effects of the entire library by stating that `validate` will always return either true or false.

```
function AJV() {}
AJV.prototype.validate=function(n,s) {
   return make('AnyBool');
};
AJV.prototype.addSchema=function(n,s) {
};
```

We must also collect edges that point from serverless functions to services. To do so, we treat the AWS SDK library as a special analysis object when we statically detect that it is imported. Then, when we encounter a method call that would write to a service, we produce a corresponding edge in the graph. To determine which specific resource is being accessed, we track the set of possible values that could flow into the identifying parameters of the SDK method. In order for this technique to produce precise results, the resource name must be defined as a string literal or constructed from known values elsewhere in the program. However, we find that this is nearly universally the case for real world programs we have analyzed, and that accepting resource names as external inputs is rare.

A simplified example of an extended service call graph is shown in Figure 1, alongside the corresponding traditional call graph. The standard call graph in Figure 1(a) cannot capture information flow from the potentially sensitive user input at `GET endpoint` to the potentially dangerous sink at `Outgoing Email`; in contrast, the extended service call graph in Figure 1(b) captures the link through service `SNS Topic`. We note that our extended service call graph may still miss some edges. In particular, we cannot capture program flows that require direct human intervention, such as a lambda sending an email to an administrator requesting that they manually trigger a web endpoint. However, we claim that such flows would be difficult to capture in any program analysis, including ones that utilize dynamic instrumentation.

4 Applications

To further highlight the benefits of our approach, we now discuss potential uses of an extended service call graph to perform static analysis on serverless clouds.

Information flow and security analysis. An important application of static analysis is detecting leakage of sensitive information to untrusted sinks. Serverless applications that interface with mobile phones or sensor devices may request that users upload personal information such as location data. Using standard methods of call graph construction, flow of sensitive inputs can be tracked only within the boundaries of the original entrypoint function. If the sensitive information flows through a platform service before reaching a potentially dangerous sink, the leak will not be detected, as the link from source to sink through the service is not captured in the call graph. Conversely, in extended service call graphs, an information flow analysis may define special semantics for functions connected to service nodes, allowing the analysis to trace flow of sensitive information to other parts of the program where these leaks may occur, as illustrated in Figure 1.

Resource estimation. Determining the computational resources required by programs that execute in the cloud is an active area of research [10]. Accurate estimation of the space, computing power, and running time required for a serverless function to execute is paramount for developers who wish to correctly provision containers for their code. The ability to accurately gauge bandwidth required to pass data between functions is also crucial to making informed decisions about task placement in geo-distributed computing platforms.

The platform services that interact with serverless applications typically operate over highly structured data, which allows for precise determination of its size. When these estimates are coupled with the extended service call graph, we envision it is possible to accurately predict the size of inputs to and outputs from serverless functions. This allows leverage of existing techniques to estimate resources used by various microservices across a serverless application as a function of their input size [14].

Documentation. Call graphs have previously been explored as a method for documenting source code by visually demonstrating to developers function usage across a codebase [37]. These methods have been expanded to include generation of supplemental documentation derived from the call graph, for example by analyzing the structure of a call graph to infer method importance and generate verbal descriptions of functions [30]. We have observed that some serverless applications include manually drawn graphs conveying this information [5, 20], suggesting that developers find this style of documentation useful. In our testing, we are able to statically produce comparable graphs that display the same network of functions and services, but with additional guarantees that our extended service call graph reflects the current state of the code.

Structural analysis and dead code elimination. Research suggests that improving structural attributes of code, such as increasing cohesion and reducing coupling, may have a direct impact on code maintainability and the resulting software quality and performance [2]. Among the simplest of these transformations is the elimination of unused code. Building a separate call graph for each serverless function, without capturing the flow between them, will fail to detect if there are functions that never execute. By including additional context from the extended service call graph, it becomes possible to detect functions that trigger off of a dormant platform service by identifying disconnected subgraphs. One can also detect a platform service that is written to but never accessed by identifying service nodes with no outbound edges. Both of these conditions may indicate maintainability problems and a need to perform refactoring of associated code.

5 Applicability on Real World Programs

To investigate the ability of our tool to produce call graphs for real applications, we pull samples from the AWS Serverless Application Repository [18]. This repository features a large collection of serverless programs, written by volunteer submitters, that can be easily deployed directly from the AWS control panel. Of the applications hosted in the repository, we were able to access source code for 146 projects. Of these, 64 were written in Javascript, 57 were written in Python, and the remainder were written in a variety of other languages, commonly Java, Go, or a typed variant of Javascript. We analyze the Javascript packages in the repository.

Of the 64 Javascript applications available, 15 include instructions for manual configuration or shell scripts in lieu of providing a declarative configuration file. Though it is still possible to detect places in the program where platform services are accessed, our tool is unable to statically determine event triggers configured for these systems. As such, we cannot generate complete extended service call graphs for these packages, even with the use of our analysis tool. Another 13 of the remaining applications are triggered using an API gateway, but do not make any other apparent use of platform services. These programs can be adequately analyzed using existing techniques for call graph construction, which is subsumed by our analysis. Therefore, no value is added by generating an extended service call graph and we choose to omit these applications.

For the 36 remaining applications available in the repository that use platform services in some way and provide configuration files, we note some additional characteristics. The majority are single serverless functions that fetch data from one platform service, transform it some way, and then publish it to a different platform service. While potentially unremarkable on their own, we find that many of these packages appear intended to abstract some small, reusable task that might be desirable in a larger application, for instance, encrypting uploaded files before storing them in an S3 bucket [25]. As such, analysis of these serverless 'libraries' will still have value for serverless platform providers who wish to statically guarantee properties about what tasks a serverless function performs or to implement a permission-based security model.

Seven of these 36 packages are applications with complex chains where data is transferred between serverless functions and services. These are ideal targets for static analysis, as they offer the greatest opportunities for uncovering unintended or notable behavior. We find that in all of these applications, static resolution of the identifiers for resources accessed by application code is possible. Consequently, for all the applications that contain interesting control flow, constructing an extended service call graph statically using our method will produce a result identical to one constructed manually.

Currently, we focus our attention on extending our tool to support the full set of language features that serverless programmers employ. Our tool currently parses only a subset of the seven complete applications described above. However, we are actively working on the implementing the full set of features to support all programs contained in the application repository. We plan to open source the tool.

6 Discussion

Open Issues. Call graphs are a vital first step in performing several types of static analysis. However, defining new classes of nodes that connect serverless applications into a single graph does not completely describe how these nodes behave in situations where they are required to transfer more nuanced information about state. Further exploration is necessary to determine how services such as DynamoDB tables can be modeled to precisely capture a safe over-approximation of the data that may be read out of a table when it is updated by a serverless function. This issue is further compounded by application frameworks such as GraphQL [15], which abstract the data model and obscure which tables are being accessed. Additionally, work done for this paper has been specific to the AWS Lambda platform. Study toward the applicability of these methods to other serverless platforms such as OpenWhisk [4] and Google Cloud Functions [19] is warranted, including closer examination of the modes of configuration for these platforms.

Risks. While regard for serverless computing from industry has been optimistic, the technology that powers these platforms is still in its infancy. There are not yet many large scale open source serverless projects or benchmarks available. This leaves program analysts with few sufficiently complex applications to test the efficacy of their tools. Given the enterprise-oriented and pay-to-execute nature of most serverless implementations, there is potential that high quality open source benchmarks may be slow to materialize, hindering research efforts in the field. Similarly, the rapid evolution of this technology means that analysis tools may have limited applicability by the time they reach the public if the serverless paradigm undergoes substantial change in the future.

There is also risk that fundamental limitations of static analysis may make answering certain types of questions intractable for serverless programs. Notably, these programs tend to be written in interpreted scripting languages such as Python and NodeJS [3], which heavily feature abstract constructs that have historically limited static analyses, such as first-class functions, untyped objects, and file imports that are resolved at runtime [9]. This limits the ability to use existing tools for Javascript analysis [13, 16, 24], as these tools tend to have difficulty scaling to large programs or do not support the full set of modern Javascript features. There is potential that tools created specifically for the serverless platform may suffer from similar setbacks, though the current set of available programs suggest this is not the case.

Need for Applications. In this paper, we leverage observed properties of serverless applications to devise a novel method for constructing call graphs that represent the entire structure of these programs. Validation of the utility of the extended service call graph model, and of the applicability of static analysis as a whole, requires determining the degree to which these properties hold across a variety of real-world applications. As such, the program analysis community would benefit from the availability of larger open source serverless applications that could serve as suitable benchmarks, as well as additional information about real and envisioned use cases for serverless computing. These applications and information can also provide guidance about the types of analyses and tooling that will be most useful to the serverless community.

Acknowledgements

This work was supported in part by NSF award CNS-1553340.

References

[1] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure Serverless Computing Using Dynamic Information Flow Control. *arXiv preprint arXiv:1802.08984*, 2018.

[2] Erik Arisholm. Empirical Assessment of the Impact of Structural Properties on the Changeability of Object-Oriented Software. *Information and Software Technology*, 48(11):1046–1055, 2006.

[3] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless Computing: Current Trends and Open Problems. *Research Advances in Cloud Computing*, pages 1–20, 2017.

[4] Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. Cloud-Native, Event-Based Programming for Mobile Applications. In *International Conference on Mobile Software Engineering and Systems*, pages 287–288, 2016.

[5] Evan Chiu. serverless-galleria, 2018. URL: `https://github.com/evanchiu/serverless-galleria`.

[6] Mihai Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. Technical report, Wisconsin University-Madison, 2006.

[7] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A Systematic Survey of Program Comprehension Through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[8] Adam Eivy. Be Wary of the Economics of "Serverless" Cloud Computing. *IEEE Cloud Computing*, 4(2):6–12, 2017.

[9] Michael Furr, Jong-hoon David An, and Jeffrey S Foster. Profile-Guided Static Typing for Dynamic Scripting Languages. In *ACM SIGPLAN Notices*, volume 44, pages 283–300, 2009.

[10] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive Elastic Resource Scaling for Cloud Systems. In *International Conference on Network and Service Management*, pages 9–16, 2010.

[11] David Grove and Craig Chambers. A Framework for Call Graph Construction Algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, 2001.

[12] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call Graph Construction in Object-Oriented Languages. *ACM SIGPLAN Notices*, 32(10):108–124, 1997.

[13] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the World Wide Web from Vulnerable JavaScript. In *International Symposium on Software Testing and Analysis*, pages 177–187, 2011.

[14] Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *ACM SIGPLAN Notices*, volume 44, pages 127–139, 2009.

[15] Olaf Hartig and Jorge Pérez. Semantics and Complexity of GraphQL. In *Conference on World Wide Web*, pages 1155–1164, 2018.

[16] IBM. T.j. watson libraries for analysis (wala), 2019. URL: http://wala.sourceforge.net/wiki/index.php/Main_Page.

[17] Amazon Web Services Inc. Aws cloudformation template formats, 2019. URL: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-formats.html.

[18] Amazon Web Services Inc. Aws serverless application repository, 2019. URL: https://serverlessrepo.aws.amazon.com/applications.

[19] Google Inc. Google cloud functions: Event-driven serverless compute platform, 2019. URL: https://cloud.google.com/functions/.

[20] Nordstrom Inc. hello-retail-workshop, 2019. URL: https://github.com/Nordstrom/hello-retail-workshop.

[21] Serverless Inc. Why serverless?, 2019. URL: https://serverless.com/learn/overview/.

[22] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383*, 2019.

[23] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial Dead Code Elimination. *ACM SIGPLAN Notices*, 29(6):147–158, 1994.

[24] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 541–551, 2015.

[25] Dan Lamb. s3-pgp-encryptor, 2018. URL: https://github.com/bmalnad/s3-pgp-encryptor.

[26] Philipp Leitner, Jürgen Cito, and Emanuel Stöckli. Modelling and Managing Deployment Costs of Microservice-Based Cloud Applications. In *International Conference on Utility and Cloud Computing*, pages 165–174, 2016.

[27] Wei-Tsung Lin, Chandra Krintz, and Rich Wolski. Tracing Function Dependencies Across Clouds. In *IEEE International Conference on Cloud Computing*, pages 253–260, 2018.

[28] Wei-Tsung Lin, Chandra Krintz, Rich Wolski, Michael Zhang, Xiaogang Cai, Tongjun Li, and Weijin Xu. Tracking Causal Order in AWS Lambda Applications. In *IEEE International Conference on Cloud Engineering*, pages 50–60, 2018.

[29] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha. A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms. In *IEEE International Conference on Cloud Computing Technology and Science*, pages 162–169, 2017.

[30] Paul W McBurney and Collin McMillan. Automatic Documentation Generation via Source Code Summarization of Method Context. In *International Conference on Program Comprehension*, pages 279–290, 2014.

[31] Garrett McGrath and Paul R Brenner. Serverless Computing: Design, Implementation, and Performance. In *International Conference on Distributed Computing Systems Workshops*, pages 405–410, 2017.

[32] Garrett McGrath, Jared Short, Stephen Ennis, Brenden Judson, and Paul Brenner. Cloud Event Programming Paradigms: Applications and Analysis. In *IEEE International Conference on Cloud Computing*, pages 400–406, 2016.

[33] Marco Pistoia, Satish Chandra, Stephen J Fink, and Eran Yahav. A Survey of Static Analysis Methods for Identifying Security Vulnerabilities in Software Systems. *IBM Systems Journal*, 46(2):265–288, 2007.

[34] AWS Serverless Application Repository. Amazon resource names (arns) and aws service namespaces, 2019. URL: https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html.

[35] Jin Shao, Hao Wei, Qianxiang Wang, and Hong Mei. A Runtime Model Based Monitoring Approach for Cloud. In *IEEE International Conference on Cloud Computing*, pages 313–320, 2010.

[36] Josef Spillner. Practical Tooling for Serverless Computing. In *International Conference on Utility and Cloud Computing*, pages 185–186, 2017.

[37] Arie Van Deursen and Tobias Kuipers. Building Documentation Generators. In *International Conference on Software Maintenance-1999*, pages 40–49, 1999.

[38] Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, et al. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 179–182, 2016.

[39] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In {*USENIX*} *Annual Technical Conference*, pages 133–146, 2018.

[40] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a Chatbot with Serverless Computing. *International Workshop on Mashups of Things and APIs*, page 5, 2016.