Static Call Graph Construction in AWS Lambda Serverless Applications

Your N. Here
Your Institution

Second Name
Second Institution

Third Name
Third Institution

Abstract

This paper describes an approach for statically constructing call graphs for applications that execute in a serverless cloud. We briefly introduce static analysis and explain the role of a call graph in performing such analyses, then discuss the benefits of using such an approach over existing dynamic techniques like those that employ program traces. We then explore the challenges associated with capturing complete program flows in a serverless environment, where state passes between lambda functions through event triggers associated with external data stores. To implement our discovered techniques we present _____, a tool for statically constructing call graphs on AWS Lambda applications written in Javascript.

1 Introduction

A call graph is a directed graph where each node represents some unit of code, usually a function, and each edge represents a path through the program where a given piece of code may cause another to execute, such as a call site where one function invokes another [9]. Closely related to the more detailed interprocedural control flow graph, this intermediate representation of a program has numerous benefits beyond its instinctive use as an aid for manually inspecting the general structure of a program. Call graph construction is an instrumental step in many static analyses, such as those for identifying and optimizing away unused code [17] and those for detecting potential software vulnerabilities [25] or malicious behavior [5]. As such, constructing a call graph for serverless programs is an integral first step in being able to reason about the safety and performance of their operation in abstract terms.

However, classical methods of call graph construction are unable to generate complete call graphs for serverless programs. These methods typically rely on traversal of the program from an identified entrypoint, generating summaries of encountered blocks of code and then resolving potential calls between blocks using a predefined set of rules that may make simplifying assumptions about context or flow [1]. By contrast, serverless programs are fundamentally event-driven [23]. This means that an accurate call graph must be able to resolve not only function calls, but also implicit state transfer that occurs when a lambda writes to a message queue or database that has an associated event. Attempts at classic call graph construction are further confounded by an inability to detect these events, which are often configured in a declarative style specific to a given serverless provider [19], though deployment tools such as the Serverless Framework have begun to present platform-agnostic file-based solutions to this configuration [16].

Without access to techniques for reasoning statically about serverless programs, some have opted to consider runtime analyses to visualize program structure [20,21], track the flow of sensitive information [2], or measure resource costs [29]. However, in the absence of information collected from the source code itself, these tools must instead instrument monitoring that also runs in the serverless cloud. This additional code introduces significant overhead, both in terms of impact on scalability [21], and in monetary cost to execute the analysis code in the cloud [6]. Furthermore, these techniques suffer from limitations inherent to constructing views of a program from path profiling, such as potentially missing infrequently traversed paths like those used for error handling [3]. Together, these limitations significantly restrict the types of questions an analysis is suited to answer and hamper the development of tools that are useful to programmers at the time they are authoring applications.

In order to address this gap, we have begun building _____, a tool for constructing call graphs statically with special consideration for the needs of serverless applications. Specifically, we contribute a language-agnostic procedure for generating service call graphs that incorporate flow to and from common backend services without the use of program traces, and consider implementation details for NodeJS applications on the AWS Lambda platform. To do so, we extend traditional call graph semantics with novel techniques for considering virtual nodes in the call graph which represent platform services and the actions that cause them to trigger lambda functions. We thus advance call graph analysis by providing a means for multiple programs that may even be running in separate execution environments to be integrated into a single extended call graph that provides information about the limited interfaces between which the constituent programs may share state.

2 Characteristics of Lambdas

AWS Lambda programs have several common characteristics which inform the design of a static analysis. Most notably, serverless programs tend to be written in interpreted scripting languages such as Python and NodeJS [4]. These programs heavily feature abstract constructs that have historically confounded static analyses, such as first-class functions, untyped objects, and file imports that are resolved at runtime [8]. Consequently, an analysis designed for serverless applications must be able to support these constructs to be able to detect the full set of possible invocations from a given function. This limits the ability to use existing tools for Javascript analysis [10, 14, 18], as these tools tend to have difficulty scaling to large programs or do not support the full set of modern Javascript features.

Additionally, the event-driven design of the serverless architecture suggests that traditional means of considering dependency between lambda invocations may be impractical. An effective tool for resolving events statically must be able to filter messages which potentially trigger events on content in order to consider precisely which events may be triggered [26]. Previous work has been performedto design event-aware semantics for static analysis in NodeJS [22]. However, this approach relies on explicit subscription to and emission of events, while Lambda event triggers are described declaratively, and a write to a platform service may or may not have an associated event.

3 Design

Because of the limitations for existing Javascript analysis tools, we elect to implement our analysis directly against the abstract syntax tree (AST). An AST converts a concrete program into a nested structure of tokens that describe the syntax of a language formally [30]. To produce this AST, we use the ESPRIMA library [12], which supports translation of even modern and experimental language features. The AST it produces conforms to the ESTree standard [11], which allows the Javascript parser to be easily replaced in the future.

We also opt to use flow-sensitive tracking of program state when resolving possible function calls to build the call graph. Though flow-sensitive analyses tend to give a minor improvement in precision of their results, they are often impractical for analysis of real programs, as tracking the values that can reach any particular program point often does not scale [13]. However, we find that for AWS Lambda programs, if externally included libraries can be conservatively abstracted away, each individual lambda is of manageable scope. This makes flow-sensitive call graph construction tractable even in spite of the path explosion problem it typically faces.

3.1 Configuration Detection

Serverless applications are organized as a set of libraries, supported by a configuration file that identifies specific functions inside these libraries that act as handlers for one or more events [24]. The first step in constructing a call graph is identifying which source files contain code that acts as an entrypoint into the serverless application. For AWS Lambda applications, two common formats for this configuration are CloudFormation templates [15] and Serverless Framework templates [16], both of which are YAML files that list a collection of resources that should be packaged for deployment. By recursing the source tree and identifying all YAML files that conform to either the CloudFormation or Serverless format then filtering the resources defined in those configuration files, we may immediately identify all non-lambda nodes in the service call graph that contain edges to functions inside a lambda, as well as the information needed to locate the triggered source code.

Significantly, the content of this YAML file also provides additional information that informs the design of the analysis. In the case of AWS Lambda programs, platform services such as AWS Simple Queue Service and DynamoDB are identified by Amazon Resource Names (ARNs) which describe the data center and a unique identifier for the associated table or bucket [27]. Empirically, we also observe that writes to these resources from within the source code of Lambda programs tends to identify the target resource with a statically resolvable string constant from within the program, rather than through indeterminate user input. This allows us to precisely capture which tables are being accessed from each lambda, and surface this information to the nodes created in the service call graph.

The configuration YAML also explicitly declares typing for each event trigger, which affords us an opportunity to further differentiate events when defining our static analysis. For call graph analysis in particular, a central question is identifying which events may serve as entry points into the program [31]. These entry points define the set of reachable methods from which the analysis should begin traversing code to build the full call graph [9]. By contrast, some event types, such as those attached to a DynamoDB table, may be assumed to only fire in response to a write which has occurred in another lambda that executed as the result of a different originating event. For our tool we make such an assumption for updates to DynamoDB tables, S3 buckets, and SQS buckets, as we observe that in most AWS Lambda applications writes to the same resources can be found in other lambdas triggered by web requests made to Amazon API Gateway endpoints, suggesting that HTTP traffic is the primary driver for most lambda applications. However, we note that this behavior may warrant tuning on a per-program basis in order to achieve maximally precise results.

3.2 External Libraries

Summarizing the effects of language libraries and third party code is a recurring concern of static analysis, particularly in scripting languages where highly dynamic features and a lack of typing information make reasoning about the effects of function calls difficult [7]. During the construction of a lambda service call graph, this manifests in uncertainty about libraries making unexpected calls to platform services or to other application code for which the library has obtained a reference, as these actions create new edges in the graph that are not readily apparent. We make the assumption that any callback which has been passed in to external code will be called on some branch in the invoked function, but that there will otherwise be no effect on the call graph. This allows us to roughly summarize libraries without directly analyzing them.

We note that this abstraction is a source of unsoundness in our call graph construction, as it is possible for imported code to store a reference to a callback and then execute that callback in another function, even if it is not provided again. While this pattern is common for event subscription frameworks and some object-oriented paradigms, in practice we find that it is generally not the case for most Javascript libraries used in serverless applications. The design and billing scheme of AWS Lambda highly encourages use of a microservice paradigm when writing applications, with runtimes for individual lambdas often measured in milliseconds [28]. We suggest that the stateless nature of lambda functions coupled with these short runtimes may be at least partially responsible for the lack of stateful library calls, making this assumption somewhat safer.

References

[1] Karim Ali and Ondřej Lhoták. Application-only call graph construction. In *European Conference on Object-Oriented Programming*, pages 688–712. Springer, 2012.

[2] 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.

[3] Taweesup Apiwattanapong and Mary Jean Harrold. Selective path profiling. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 35–42. ACM, 2002.

[4] 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. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[5] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. Technical report, Wisconsin University-Madison, 2006.

[6] Adam Eivy. Be wary of the economics of "serverless" cloud computing. *IEEE Cloud Computing*, 4(2):6–12, 2017.

[7] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

[8] 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. ACM, 2009.

[9] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, 32(10):108–124, 1997.

[10] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2011.

[11] David Herman, Ingvar Stepanyan, Nicholas Zakas, Ariya Hidayat, Henry Ficarra, Michael Zhu, Logan Smyth, and Daniel Tschinder. Estree, 2019. URL: https://github.com/estree/estree.

[12] Ariya Hidayat. Ecmascript parsing infrastructure for multipurpose analysis, 2019. URL: http://esprima.org/.

[13] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 113–123. ACM, 2000.

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

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

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

[17] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. *ACM SIGPLAN Notices*, 29(6):147–158, 1994.

[18] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. Practically tunable static analysis framework for large-scale javascript applications (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 541–551. IEEE, 2015.

[19] Kyriakos Kritikos and Paweł Skrzypek. A review of serverless frameworks. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 161–168. IEEE, 2018.

[20] Wei-Tsung Lin, Chandra Krintz, and Rich Wolski. Tracing function dependencies across clouds. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 253–260. IEEE, 2018.

[21] Wei-Tsung Lin, Chandra Krintz, Rich Wolski, Michael Zhang, Xiaogang Cai, Tongjun Li, and Weijin Xu. Tracking causal order in aws lambda applications. In *Cloud Engineering (IC2E), 2018 IEEE International Conference on*, pages 50–60. IEEE, 2018.

[22] Magnus Madsen, Frank Tip, and Ondrej Lhoták. Static analysis of event-driven node.js javascript applications. In *OOPSLA*, 2015.

[23] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE, 2017.

[24] Garrett McGrath, Jared Short, Stephen Ennis, Brenden Judson, and Paul Brenner. Cloud event programming paradigms: Applications and analysis. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 400–406. IEEE, 2016.

[25] 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.

[26] Daniel Popescu, Joshua Garcia, Kevin Bierhoff, and Nenad Medvidovic. Impact analysis for distributed event-based systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 241–251. ACM, 2012.

[27] 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.

[28] 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 *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 179–182. IEEE, 2016.

[29] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 133–146, 2018.

[30] David S Wile. Abstract syntax from concrete syntax. In *Proceedings of the (19th) International Conference on Software Engineering*, pages 472–480. IEEE, 1997.

[31] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 89–99. IEEE, 2015.