Formalizing Event-Driven Behavior of Serverless Applications

Matthew Obetz, Anirban Das, Timothy Castiglia, Stacy Patterson, Ana Milanova

Formalizing Event-Driven Behavior of Serverless Applications

Matthew Obetz, Anirban Das, Timothy Castiglia, Stacy Patterson, and Ana Milanova

Rensselaer Polytechnic Institute, Troy NY 12180, USA
{obetzm,dasa3,castit,pattes3,milana2}@rpi.edu

Abstract. We present new operational semantics for serverless computing that model the event-driven relationships between serverless functions, as well as their interaction with platform services such as databases and object stores. These semantics precisely encapsulate how control transfers between functions, both directly and through reads and writes to platform services. We use these semantics to define the notion of the service call graph for serverless applications that captures program flows through functions and services. Finally, we construct service call graphs for 12 serverless JavaScript applications, using a prototype of our call graph construction algorithm, and we evaluate their accuracy.

Keywords: serverless computing, formal semantics, call graph

1 Introduction

Serverless computing has grown significantly in recent years and so has the need for abstractions and program analysis tools that target serverless applications [8]. Existing abstractions emphasize unique features of the environment where serverless functions execute [6,4]. However, these abstractions do not consider effects of transmitting data to other services and functions. Data transmitted in this fashion triggers new executions of serverless functions that spawn in response to a change in state on their associated service. Without *operational semantics* that capture this behavior, program analysis cannot construct a precise call graph and cannot reason about dataflow between parts of a serverless application. The lack of formal semantics also hinders more advanced reasoning about data privacy, application correctness, and resource usage.

To address this gap, we propose new operational semantics for event-driven serverless computation. These semantics describe how writes and reads to platform services create inter-function control transfer in serverless applications. Our semantics formalize the most common platform services including object stores, databases, notification services, queues, and stateless services. The semantics gives rise to the *service call graph*, which extends the classical call graph to include new nodes and edges. The new nodes represent the platform services written to or read by application code; the new edges represent the writes and reads to services from serverless functions.

We make the following contributions:

- We formulate new operational semantics for the execution of serverless programs. These semantics model (1) interactions of serverless functions with platform services, including event triggers that cause additional functions to execute, and (2) function composition.
- We extend the traditional notion of a call graph with new types of nodes and edges that represent event-driven behavior on serverless platforms. These new nodes and edges capture the inter-function control and state transfer represented in our operational semantics.
- We design and implement an algorithm for constructing service call graphs and evaluate its accuracy on 12 serverless programs collected from GitHub.

Related Work. Our semantics for the lifecycle of a single serverless function are closely related to those used in a recent formalization of serverless computing [6]. That work focused on modeling low-level behavior of serverless systems. Such models are useful for capturing behavior such as program non-determinism that can arise from reading state from previous executions of serverless functions. Our semantics start from this model to describe initiating requests, language-agnostic computation steps, and generated responses. However, the semantics defined in [6] do not capture inter-function communication and program flows that span multiple serverless functions. Specifically, these semantics limit data persistence to a locking transactional key-value store. Our semantics introduce several new state domains that model common services. More importantly, the previous semantics also lack a conceptualization of serverless events, which initiate execution of a serverless function when state is manipulated on a data storage service. We model these interactions with a new collection of event semantics that capture state transfer between serverless components.

The service call graph shares some features of message flow graphs for distributed event-based systems that communicate through publish-subscribe middleware [5], however, retrieval of data from databases and object stores cannot be succinctly captured in publish-subscribe semantics. Our work considers not only notification-based communication, but also messages that pass through other channels available to serverless applications.

In preliminary work [10], we introduced the notion of the service call graph. In this paper, we formalize the definition in terms of our new operational semantics. Further, we design and implement a call graph construction algorithm and present experimental results on 12 real-world serverless applications.

Outline. Section 2 summarizes the operational semantics, and Section 3 presents our serverless call graph construction. We evaluate the call graph construction accuracy in Section 4 and conclude in Section 5. Our technical report [9] presents an expanded discussion on the serverless model and our semantics and algorithms.

2 Semantics for Serverless Computation

We introduce operational semantics for the execution of serverless applications. The goals of these serverless semantics are to: (1) precisely model the semantics of communication between serverless functions and platform services, and (2) capture program flows that are introduced as a result of this communication.

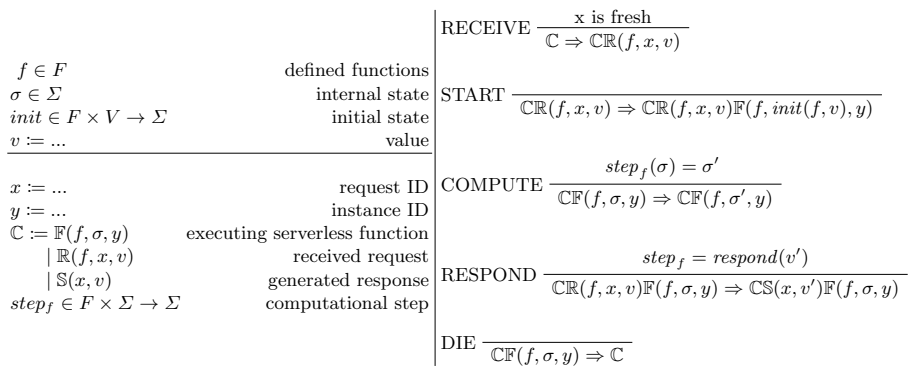

Fig. 1. In-process semantics models the sequence of steps in an individual serverless functions. A full serverless application $\mathbb{C}$ is modeled as a set of requests $\mathbb{R}$, executing functions $\mathbb{F}$, and generated responses $\mathbb{S}$. Functions and requests are appended to $\mathbb{C}$ as they become active, and are removed from $\mathbb{C}$ as they terminate or are responded to.

2.1 In-Process Semantics

In-process semantics are defined in Figure 1. These semantics capture the sequence of steps in an individual serverless function. When an *external* gateway service initiates a request for the execution of the serverless program, the platform applies the RECEIVE rule which adds a new request $\mathbb{R}$. The request contains a serverless function f and a data value v that is passed to the function. Most commonly, RECEIVE represents a request made to a public web endpoint for the serverless application. When an unhandled request exists, the platform applies the START rule which initializes f with an initial state $init(f, v)$ and starts the execution of f. We note that $init(f, v)$ captures both initial state at cold and warm start. COMPUTE models the execution steps in function f. Similarly to [6], COMPUTE is a language agnostic representation of transitions on state σ. COMPUTE absorbs interactions with platform services, e.g., *upload* to object stores (see Section 2.2). A serverless function may issue a response, in which case the platform applies the RESPOND rule. This rule removes the unhandled request $\mathbb{R}(f, x, v)$ from the system and replaces it with a response $\mathbb{S}(x, v')$, where v' is the value provided by the responding serverless function. Responses represent data which is sent back to the external service that initiated the request; they are terminal states and are not used for further computation within the platform. Finally, functions may terminate through the application of the DIE rule. The system reaches a stable state when all requests have been responded to and no serverless functions are still executing.

2.2 Event Semantics

We extend the in-process semantics with an event semantics to capture interaction of functions with platform services, as well as direct invocation. We develop semantics for each service: object stores, databases, notifications, queues, and stateless services. These semantics detail how serverless functions interface with

that specific service during execution. In this section, we detail the semantics of object stores. We include the semantics for the remaining services in the technical report [9]; these semantics follow the structure of the object store semantics, however, each details behavior specific to the service they model.

The semantic rules can be broadly grouped into rules that *write* the state of a service (UPLOAD and REMOVE for object stores; INSERT, UPDATE, and DELETE, for databases; and ENQUEUE for queues), and rules that *read* data from a service into the state of an executing serverless function (READ for object stores, SELECT for databases, and DEQUEUE for queues).

```
functions:
  processor:
    handler: index.process
    events:
    - s3:
      bucket: photos
      event: s3:ObjectCreated:*
```

Fig. 2. An example event configuration. The serverless function `processor` is triggered when an object is added to the `photos` bucket. In the semantics, this event is represented as the fact $e(c, processor)$ where $c = (photos, upload)$.

Our semantics introduce a domain of events E that captures function invocations due to service state transitions. An event $e(c, f) \in E$ consists of two parts: a triggering condition c and an associated serverless function f. Triggering conditions are generally defined by a unique service identifier *sid* and an operation *op* (e.g, *upload* to an object store); we write $c = (sid, op)$. Program configurations unambiguously reference their associated services and the associated serverless functions. We reduce configurations to set of events $e(c, f)$ during static analysis. We present an example configuration in Figure 2. An event is *triggered* when a serverless function performs a step that fires the event condition. For instance, an upload to an object store b will activate all events tied to upload to b. To capture the effect of these triggering events, our semantics introduce the function *trigger*. This function accepts a triggerring condition $c = (sid, op)$, and returns the set of functions f for which there is $e(c, f)$, i.e., the set of functions that will execute when a function runs operation *op* on service *sid*. We note that some types of triggering conditions defined in our semantics are officially supported by serverless platforms but rarely occur in practice, such as the trigger associated with a REMOVE from an object store.

Our semantics distinguish between functions triggered by external requests and functions triggered by events on services. The platform applies RECEIVE followed by START on functions triggered by external requests. It immediately applies START on functions triggered by "internal" events on services. Our semantics allow that any function that is part of the serverless application may issue a response to the external request. RECEIVE and RESPOND define the "boundary" of the serverless application, although functions may continue to execute and modify services after a RESPOND.

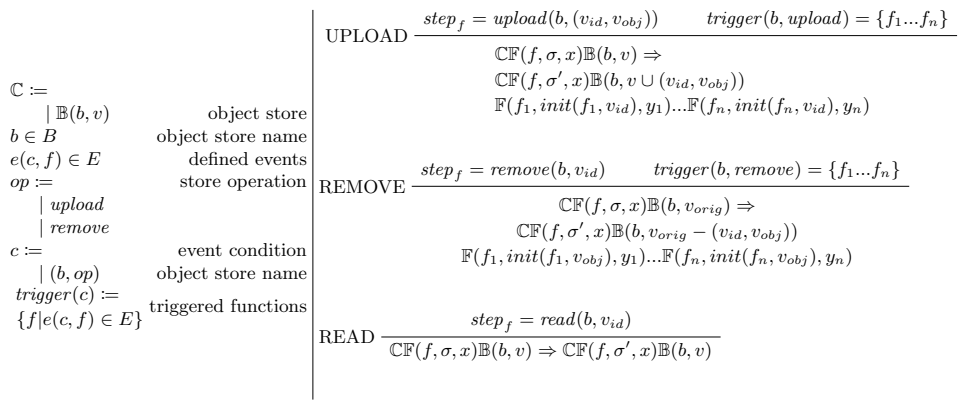

Fig. 3. Object store event semantics.

We define semantics for object stores in Figure 3. Each object store has a unique identifier b in B, the set of object stores defined for the application. Object stores provide a filesystem-like interface for writing and reading data. In the semantics, this interaction is encoded by allowing serverless functions to write or overwrite some value v in a named bucket by applying the UPLOAD rule. When a file is uploaded, all events triggered by state transition on the receiving bucket initialize their respective function(s). Serverless functions can also delete data contained in a bucket through application of the REMOVE rule. When a function retrieves a data value from a bucket, the READ rule accesses the associated data and assigns it to a variable inside the function's local state.

Our event semantics are synchronous in the sense that a request to a service and the execution of the request by the service happen in "one step". This facilitates static reasoning. In practice, a request is decoupled from the execution; we conjecture that the synchronous semantics are sufficient as programs implicitly synchronize events on services: a read in f_2 *is triggered* by a write in f_1. Further, for reads and writes within the same function, standard libraries typically provide only synchronous methods for interacting with platform services. We will formalize sufficiency conditions on programs in future work.

2.3 Platform Behavior Encoded in Semantics

Our semantics are sufficiently expressive to capture features of serverless platforms that impact system state in unintuitive ways. We illustrate below.

Non-finality of RESPOND. Unlike `return` statements in normal functions, responses from a serverless function do not return from the function. Consider the example in Figure 4. This serverless function accepts a URL string and generates a random short slug for that URL. It immediately responds with the generated shortened URL, then afterward, writes the association between the slug and the original URL to a database. Our semantics models the execution of

```
export.shortenUrl = function(event, context, callback) {
  let url = event.body;
  let slug = crypto.randomBytes(8).toString(...).replace(...);
  callback(null, {shortUrl: context.domainName + slug});
  dynamodb.put({
    TableName: "ShortUrls",
    Item: {slug: slug, long_url: url}
  });
}
```

Fig. 4. Example of execution continuing after response. RESPOND is applied when the `callback` passed in to the serverless function is invoked, but a database is written to after this response. Code adapted from the `url-shortener` project [11].

this serverless function by the following transitions ($\mathbb{D}$ represents the database service. INSERT has semantics similar to UPLOAD in Figure 3):

$$
\begin{aligned}
& \mathbb{C}\mathbb{D}(\mathit{ShortUrls}, v) && \\
\Longrightarrow\ & \mathbb{C}\mathbb{D}(\mathit{ShortUrls}, v)\mathbb{R}(f, x, v_1) && \text{by rule RECEIVE}(f,\ x,\ v_1) \\
\Longrightarrow\ & \mathbb{C}\mathbb{D}(\mathit{ShortUrls}, v)\mathbb{R}(f, x, v_1)\mathbb{F}(f, \sigma, y) && \text{by START}(y) \\
\Longrightarrow\ & \mathbb{C}\mathbb{D}(\mathit{ShortUrls}, v)\mathbb{R}(f, x, v_1) && \\
& \quad \mathbb{F}(f, \sigma' = \sigma[\mathit{url} \leftarrow \mathit{ev.body}, \mathit{slug} \leftarrow \mathit{rand}()], y) && \text{by COMPUTE}(f) \\
\Longrightarrow\ & \mathbb{C}\mathbb{D}(\mathit{ShortUrls}, v)\mathbb{S}(x, v')\mathbb{F}(f, \sigma', y) && \text{by RESPOND}(x,\ v' = \sigma'[\mathit{slug}]) \\
\Longrightarrow\ & \mathbb{C}\mathbb{D}(\mathit{ShortUrls}, v' \cup v)\mathbb{S}(x, v')\mathbb{F}(f, \sigma', y) && \text{by INSERT}(\mathit{ShortUrls},\ v') \\
\Longrightarrow\ & \mathbb{C}\mathbb{D}(\mathit{ShortUrls}, v' \cup v)\mathbb{S}(x, v') && \text{by DIE}(y)
\end{aligned}
$$

The application of the INSERT rule affects the final state of the system $\mathbb{C}$ by introducing the value v' to the database $\mathbb{D}$. This insertion occurs even though the serverless function has already generated a response in an earlier step.

Failures and Retried Executions. A serverless function may fail during execution for two reasons: 1) the function code enters an error state as the result of an uncaught exception, or 2) the container runtime kills the function, either because execution has timed out, or because the language interpreter fails with an error. When a function fails, the platform can retry the function by starting a new execution with a clone of the data from the original request [1].

Our semantics capture the effects of failures and retried executions that may impact system state. In particular, serverless functions that are not idempotent may emit messages to platform services that are repeated in retried executions, affecting final system state. In our semantics, these retries are modeled as an application of the DIE rule, followed by a subsequent application of START to handle a still-unsatisfied request. Consider a serverless function that uses the UPDATE rule to increment a view count. It is retried due to a spontaneous failure in the data center where the function is executing. This series of events

are modeled under our semantics as:

$$
\begin{aligned}
& \mathbb{CD}(\mathit{ViewCount}, v) && \\
\Longrightarrow\ & \mathbb{CD}(\mathit{ViewCount}, v)\mathbb{R}(f, x, v) && \text{by RECEIVE}(f,\ x,\ v) \\
\Longrightarrow\ & \mathbb{CD}(\mathit{ViewCount}, v)\mathbb{R}(f, x, v)\mathbb{F}(f, \sigma, y) && \text{by START}(y) \\
\Longrightarrow\ & \mathbb{CD}(\mathit{ViewCount}, v+1)\mathbb{R}(f, x, v)\mathbb{F}(f, \sigma, y) && \text{by UPDATE}(\mathit{ViewCount},\ (v) \rightarrow v+1) \\
\Longrightarrow\ & \mathbb{CD}(\mathit{ViewCount}, v+1)\mathbb{R}(f, x, v) && \text{by DIE}(y) \\
\Longrightarrow\ & \mathbb{CD}(\mathit{ViewCount}, v+1)\mathbb{R}(f, x, v)\mathbb{F}(f, \sigma, y') && \text{by START}(y') \\
\Longrightarrow\ & \mathbb{CD}(\mathit{ViewCount}, v+2)\mathbb{R}(f, x, v)\mathbb{F}(f, \sigma, y') && \text{by UPDATE}(\mathit{ViewCount},\ (v) \rightarrow v+1) \\
\Longrightarrow\ & \mathbb{CD}(\mathit{ViewCount}, v+2)\mathbb{S}(x, v)\mathbb{F}(f, \sigma, y') && \text{by RESPOND}(x,\ \{\}) \\
\Longrightarrow\ & \mathbb{CD}(\mathit{ViewCount}, v+2)\mathbb{S}(x, \{\}) && \text{by DIE}(y')
\end{aligned}
$$

Following these state transitions, *ViewCount* has been incremented twice, despite only a single request being made to the serverless function. Such faults are representative of data inconsistencies that exist in real serverless applications that violate the idempotency recommended by serverless providers [1].

2.4 Platform Supported Function Composition

Function composition frameworks, such as AWS Step Functions, allow developers to statically define pathways for messages through a serverless application. When one of these pathways is defined, the return value of a serverless function implicitly becomes a message passed to the serverless function or service following it in the composition. Our semantics are expressive enough to capture such behavior using the same set of state transitions as other serverless events.

Consider a Step Function composition that defines a chain of two serverless functions, f_1 and f_2. ([9] shows the real-world Step Function declaration that we model in this example.) Our semantics models the execution as follows:

$$
\begin{aligned}
& \mathbb{C} && \\
\Longrightarrow\ & \mathbb{CR}(f_{step}, x, v) && \text{by RECEIVE}(f_{step},\ x,\ v) \\
\Longrightarrow\ & \mathbb{CR}(f_{step}, x, v)\mathbb{F}(f_1, \sigma, y) && \text{by START}(y) \\
\Longrightarrow\ & \mathbb{CR}(f_{step}, x, v)\mathbb{F}(f_1, \sigma', y) && \text{by COMPUTE}(f_1) \\
\Longrightarrow\ & \mathbb{CR}(f_{step}, x, v)\mathbb{F}(f_1, \sigma', y)\mathbb{F}(f_2, \sigma'', y) && \text{by INVOKE}(f_2,\ v') \\
\Longrightarrow\ & \mathbb{CR}(f_{step}, x, v)\mathbb{F}(f_2, \sigma'', y) && \text{by DIE}(f_1,\ \sigma',\ y) \\
\Longrightarrow\ & \mathbb{CR}(f_{step}, x, v)\mathbb{F}(f_2, \sigma''', y) && \text{by COMPUTE}(f_2) \\
\Longrightarrow\ & \mathbb{CS}(x, v'')\mathbb{F}(f_2, \sigma''', y) && \text{by RESPOND}(x,\ v'') \\
\Longrightarrow\ & \mathbb{CS}(x, v'') && \text{by DIE}(f_2,\ \sigma''',\ y)
\end{aligned}
$$

This execution illustrates an important difference between standalone serverless functions and those defined as part of a composition chain. The platform starts the Step Function chain by issuing a request $\mathbb{R}(f_{step}, x, v)$ by RECEIVE. Only the final serverless function in the chain RESPONDs to the request. The "return" of all other functions in the chain is encoded into an event rule that INVOKEs the next function in the chain. Since compositions are static, the target of each INVOKE in the composition is known. To preserve the connection

to the originating Step Function request that started $\mathbb{F}(f_1, \sigma, y)$, f_2 inherits the identifier y from f_1 when it is invoked (the semantics assume that y reflects the request identifier x). Thus, the lifecycle of the first function in the composition chain is RECEIVE, START, COMPUTE, DIE; the lifecycle of the final one is START, COMPUTE, RESPOND, DIE.

3 Service Call Graphs

Our semantics enable construction of a *service call graph* that explicitly models interaction between services and serverless functions. The service call graph extends the classical call graph by adding nodes that represent platform services and edges that represent reads from services, writes to services, and transfer of control to functions triggered by state transition on services. Our graphs treat an entire intra-function call graph as a single node in order to clearly capture the interaction between serverless functions and platform services.

Construction of the service call graph proceeds in two phases: *configuration analysis* and *code analysis*. Configuration analysis processes configuration files and identifies the serverless functions and services for the given application. Each serverless function $f \in F$, and each service $b \in B$ (object store), $d \in D$ (database), $q \in Q$ (queue), and $t \in T$ (notification topic) becomes a node in the service call graph. In addition, configuration analysis also identifies the set of events $e(c, f) \in E$ and triggering conditions $c = (sid, op)$ (recall Section 2 for the explanation of e and c); each event $e(c, f) \in E$ where $c = (sid, op)$ gives rise to an edge from sid to f.

Code analysis processes each serverless functions $f \in F$. It constructs the standard interprocedural control flow graph (ICFG) of f (here, "interprocedural" refers to the local helper functions in f). The analysis tracks the set of service identifiers sid that flow to call sites in the ICFG corresponding to rules of the event semantics (such as UPLOAD, ENQUEUE, INSERT, or NOTIFY). At each such call site, the analysis adds an edge from the current serverless function f to each service sid that may reach the call site corresponding to the event rule.

4 Call Graph Implementation and Evaluation

We implement service call graph construction as an extension of the Type Analysis for JavaScript (TAJS) framework [7]. We employ a branch of TAJS that supports reasoning about asynchronous behavior [12]. Our analysis includes code that summarize the effects of third party libraries, including the AWS SDK. We constructed summaries of library functions to overcome limitations in TAJS that prevented us from performing standard whole-program analysis. We searched GitHub for repositories that included serverless configuration files that defined more than one serverless function, sorted by repository popularity. We analyze the top 12 applications that fit these criteria. To evaluate the accuracy of our generated call graphs, we compare the output of our analysis against call graphs drawn by manual inspection of the programs.

Table 1. Service Call Graph results.

Application	Lines of Code	# Functions	Sound?	Missed Edges
hello-retail	2288	14	Y	0
citizen-dispatch	865	3	N	6
galleria	641	5	Y	0
rating-service	412	2	Y	0
LEX	323	2	Y	0
lending-app	258	4	Y	0
url-shortener	172	3	Y	0
zen-beer	155	4	Y	0
greeting-app	99	2	Y	0
lane-breach	98	2	N	1
wombat	88	2	Y	0
serverless-chaining	28	2	Y	0

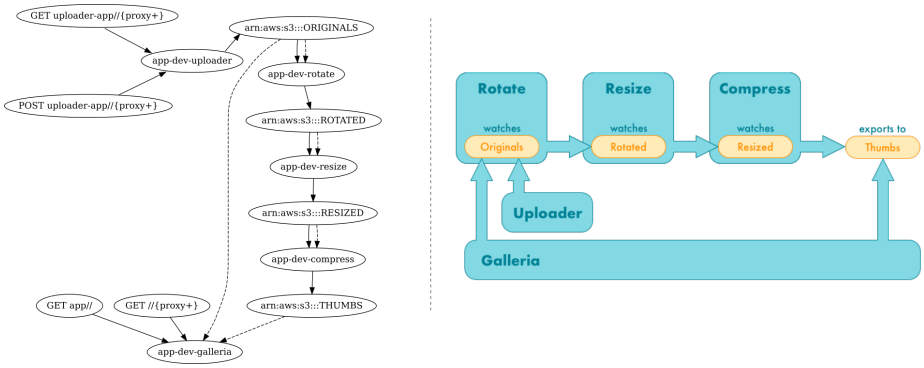

Fig. 5. Comparison of service call graph generated by our analysis for the galleria serverless application [3], and pipeline diagram provided in the repository's user documentation. In the call graph at left, the GET and POST API gateway events trigger the `app-dev-uploader` serverless function. This function then writes to the `ORIGINALS` S3 bucket, which in turn triggers the `app-dev-rotate` serverless function. This function reads from its triggering bucket then writes to a `ROTATED` bucket. The process repeats for two more image processing functions before the final image is uploaded to `THUMBS`.

Table 1 presents the analysis results. For 10 of the 12 applications, our analysis produced a service call graph *identical to the ground truth*. One such comparison is shown in Figure 5. For two applications, our analysis missed edges. In the case of `lane-breach`, the missed edge corresponded to a web request made directly to another function through the external web API. We note that it is not possible, in general, to determine whether a web address belongs to the application under analysis or a third-party web site. Fortunately, this behavior represents a discouraged pattern [2]; the program could be made more efficient using a direct invocation, which would be captured through our INVOKE rule.

In the case of `citizen-dispatch`, the analysis missed edges from serverless functions to a set of database tables that corresponded to database queries made by third-party library calls. This program violated our assumption that third-party libraries do not interact with services. Though constant service identifiers

flow to the library calls, it is difficult to statically infer which tables will be accessed by a particular call due to the nature of the query inference engine. Future versions of our tool could safely over-approximate this behavior by assuming that any library call has the potential to query all tables. If we could perform standard whole-program analysis, interactions with the database through the library would have been soundly detected. (Whole-program analysis is trivially supported in tools for languages such as Java, but it is not supported by TAJS due to the difficulty of analyzing JavaScript.)

5 Conclusion

We introduced new operational semantics for serverless computing and demonstrated how these semantics give rise to the service call graph. Finally, we presented a prototype of our service call graph construction algorithm and showed its efficacy on real-world serverless programs. In future work, we will construct analyses for improving performance and security of serverless applications.

References

1. Amazon Web Services: AWS Lambda Documentation (2019), https://docs.aws.amazon.com/lambda/index.html
2. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al.: Serverless Computing: Current Trends and Open Problems. In: Research Advances in Cloud Computing, pp. 1–20 (2017)
3. Chiu, E.: Serverless galleria (2019), https://github.com/evanchiu/serverless-galleria
4. Gabbrielli, M., Giallorenzo, S., Lanese, I., Montesi, F., Peressotti, M., Zingaro, S.P.: No More, No Less - A Formal Model for Serverless Computing. In: Int. Conf. Coordination Models and Languages. pp. 148–157 (2019)
5. Garcia, J., Popescu, D., Safi, G., Halfond, W.G.J., Medvidovic, N.: Identifying Message Flow in Distributed Event-Based Systems. In: Symp. Foundations of Software Engineering. pp. 367–377 (2013)
6. Jangda, A., Pinckney, D., Brun, Y., Guha, A.: Formal Foundations of Serverless Computing. PACMPL **3**(OOPSLA), 149:1–149:26 (2019)
7. Jensen, S.H., Møller, A., Thiemann, P.: Type Analysis for JavaScript. In: International Static Analysis Symposium (SAS). LNCS, vol. 5673 (8 2009)
8. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N.e.a.: Cloud Programming Simplified: A Berkeley View on Serverless Computing. Tech. rep., University of California at Berkeley (2019)
9. Obetz, M., Patterson, S., Milanova, A.: Formalizing Event-Driven Behavior of Serverless Applications. CoRR **abs/1912.03584** (2019), http://arxiv.org/abs/1912.03584
10. Obetz, M., Patterson, S., Milanova, A.: Static call graph construction in AWS lambda serverless applications. In: HotCloud (2019)
11. Onan, M.: url-shortener (2019), https://github.com/mdonan90/url-shortener/blob/master/create/index.js
12. Sotiropoulos, T., Livshits, B.: Static Analysis for Asynchronous JavaScript Programs. CoRR **abs/1901.03575** (2019), http://arxiv.org/abs/1901.03575