Towards Inter-service Data Flow Analysis of Serverless Applications

Giuseppe Raffa
Royal Holloway University
London, UK
giuseppe.raffa.2018@live.rhul.ac.uk

Jorge Blasco
Universidad Politécnica
Madrid, Spain
jorge.blasco.alis@upm.es

Dan O'Keeffe
Royal Holloway University
London, UK
daniel.okeeffe@rhul.ac.uk

Santanu Kumar Dash
Royal Holloway University
London, UK
santanu.dash@rhul.ac.uk

***Abstract*—The recent advent of serverless applications has created a need for static analysis tools to analyse them. However, the event-driven architecture of serverless applications, along with the black-box nature of the services they invoke, make static analysis challenging. In this work, we propose a novel approach to statically analysing serverless applications, with a focus on the identification of data flows that can lead to code injection and information leakage. To reach our goal, we first design a new suite of microbenchmarks, which we publicly release. The microbenchmarks are based on documented serverless-specific vulnerabilities and the characterization of an existing dataset. We then introduce our static analysis approach and show how it can factor in the effect of platform services and event-triggered code execution by extracting relevant information from both infrastructure and application code. This information is used to obtain a synchronous equivalent of the underlying asynchronous system, which can be inspected with a general-purpose static analysis tool. Preliminary evaluation results using a prototype implementation of our approach and the microbenchmark suite confirm the potential of our analysis technique.**

***Index Terms*—Serverless computing, Data flow analysis, Inter-procedural static analysis**

I. Introduction

Serverless computing is widely used in the software industry to develop a large variety of applications. In the serverless paradigm, enterprises rely on hardware and platform services managed by a cloud provider, e.g., Amazon [1], to cut costs and reduce operational overheads. However, as pointed out by other researchers [2] and industry experts [3], [4], the shared responsibility model [5] for cloud security implies that developers cannot outsource all security-related tasks. Indeed, recent data breaches [6] show that tools available to secure serverless applications are still evolving.

Serverless applications typically feature a large attack surface, because they are, in general, stimulated by a high number of sources and events. As a result, analysis frameworks proposed in recent academic studies [7]–[9] are focused on dynamic information flow. Despite their reasonably low overhead, such frameworks require the deployment of additional code, and, crucially, do not provide support for static analysis. The latter allows inspecting an application prior to deployment, thus detecting security vulnerabilities earlier in the development cycle. However, due to the complexity of the platform services and the event-driven nature of serverless applications, static analysis is particularly challenging and requires domain-specific models and approximations [10].

In this paper, we present our initial steps towards systematic and inter-service static analysis of serverless applications. To achieve this objective, we first design a novel suite of security-oriented microbenchmarks[1]. The majority of existing serverless benchmarks [11], [12], in fact, are focused on performance evaluation and cost effectiveness. As shown by previous research [13], microbenchmarks are useful in testing and comparing security tools. Our suite takes into account ❶ serverless-specific cases of code injection [3], [14] and information leakage [15], along with ❷ cloud service and API-related characteristics of the AWSomePy dataset [16].

We then propose an initial, two-pillar approach to the static analysis of serverless applications. First, we combine information extracted from the infrastructure code with the application code. Second, we instrument the application code in order to obtain a synchronous-like program execution flow. This enables the developer to find security vulnerabilities statically by running a general-purpose tool capable of identifying data flows. We also implement our approach in a prototype analysis framework, which we evaluate against our microbenchmarks. Despite its current limitations, the evaluation shows that our approach is feasible, as it can detect security-sensitive intra-procedural and inter-procedural data flows.

II. Background

A. Serverless Computing Model

The serverless computing model allows running application code by leveraging services made available by a cloud provider. Therefore, serverless is a misnomer, as the execution of the deployed applications still relies on servers, but their provisioning and maintenance are delegated to another organization [5]. Providers also set limits on the amount of time available for user-developed code to complete its execution. As a result, serverless applications are typically implemented as a collection of small stateless functions triggered by *events*, e.g., a HTTP request or a database update.

While chaining together custom code and platform services with events simplifies the development process, serverless applications are more difficult to analyse statically than standard

[1] https://github.com/giusepperaffa/serverless-security-microbenchmarks

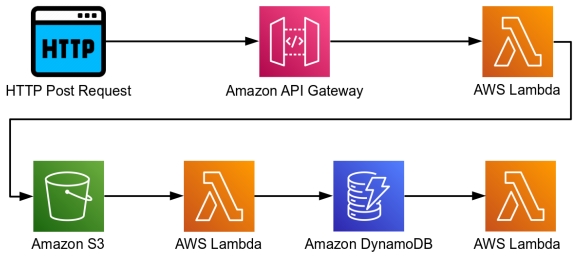

Fig. 1. Example application with an event-driven data flow. The blocks AWS Lambda identify the three event-triggered handlers included in the application.

applications. Since serverless applications routinely involve a high number of sources and events, different types of information and multiple data structures have to be considered. This inevitably makes the detection of security-sensitive data flows more error-prone and time-consuming. In addition, static analysis of these applications cannot include the code that implements platform services, as this is not available to the developer. Consequently, models and approximations are unavoidable [10].

As shown by the characterization of the AWSomePy dataset [16], another challenge is that developers often rely on configuration and management-oriented services as well as programmatic creation of cloud-based resources. While these platform features provide flexibility, they also add data flows and interactions between serverless functions that are difficult to inspect without deploying the application in the cloud.

B. Motivating Example

Fig. 1 illustrates the high-level architecture of an example application with a code injection vulnerability[2]. The application is configured through a YAML file, which specifies the application permissions as well as handlers and event-related parameters (Listing 1).

```
iamRoleStatements:
    - Effect: "Allow"
      Action:
        - "s3:ListBucket"
        - "s3:GetObject"
        - "s3:PutObject"
```

Listing 1. Application permissions specified in the configuration file.

The user-controlled entry point of the application is a HTTP Post request. This triggers the execution of a handler that uploads a file identified through the body of the request to a cloud-based repository (Listing 2). Upon detecting the new file, the S3 service processes the event and runs the second handler, which extracts information from the uploaded file and stores it in a DynamoDB database. This triggers a third handler that uses the newly-stored item in the database as part of a system command called via `os.system` (Listing 3).

While manual analysis of the application, aided by the architectural diagram, allows detecting a data flow from user input to the security-sensitive sink `os.system`, automatic identification is not possible with existing general-purpose static analysis tools for two reasons. First, ❶ such tools do not take into account the effect of triggered events, e.g., the fact that the API call `upload_file` in Listing 2 triggers the execution of another handler. This information is only present in the configuration file, which must therefore be incorporated into the analysis. Second, ❷ a non specialized tool would not be capable of modelling the `event` input arguments expected by the handlers, due to a lack of cloud service-specific information. As a result, the security-sensitive data flow of our example application would not be detected.

```
def onHTTPPostEvent(event, context):
    msgBodyDict = event['body']
    uploadFileFolder = 'uploads'
    uploadFileName = msgBodyDict + '.txt'
    s3BucketKey = os.path.join(uploadFileFolder,
        uploadFileName)
    localFile = os.path.join('/tmp', uploadFileName)
    s3Client = boto3.client('s3')
    s3Client.upload_file(localFile, os.environ['
        BUCKET_NAME'], s3BucketKey)
    return
```

Listing 2. Handler triggered by a HTTP request.

```
def onDynamoDBStream(event, context):
    authorsInfo = event['Records'][0]['dynamodb']['
        NewImage']['authors']['S']
    titleInfo = event['Records'][0]['dynamodb']['
        NewImage']['title']['S']
    eventData = authorsInfo + titleInfo
    os.system('echo %s' % eventData)
    return
```

Listing 3. Handler triggered by a DynamoDB database update.

III. Microbenchmarks Suite

To evaluate the proposed static analysis technique against representative applications, we design a preliminary set of security-oriented microbenchmarks. We next discuss the design principles behind our microbenchmark suite (§ III-A) as well as the types of vulnerability that it contains (§ III-B).

A. Design Overview

The design of our security-oriented microbenchmark suite is based on documented cases of code [3], [14] and database injection [15]. The former enables the execution of system commands in the infrastructure hosting the application, whilst the latter leads to unintended disclosure of information from a cloud-based database. While these are not serverless-specific vulnerabilities, they are among the most critical risks for serverless applications recently identified by the Cloud Security Alliance [4]. The analysis of the aforementioned cases, therefore, is crucial to understanding possible real-world scenarios relevant to serverless applications.

To better model the ways in which serverless applications are (mis-)programmed, we also rely on the characterization of AWSomePy, a collection of 145 Serverless Framework and AWS-compatible Python applications. This allows us to identify the cloud services and APIs with the highest number of

[2]The application is compatible with AWS [1] and the Serverless Framework deployment tool [17]. This does not affect the generality of our observations.

TABLE I
CONSIDERED CLOUD SERVICES ALONG WITH NUMBER OF RELEVANT AWSOMEPY REPOSITORIES AND MOST FREQUENTLY USED API.

Services	No. of Repositories	API
S3	59	`put_object`
DynamoDB	47	`put_item`
SQS	21	`send_message`
SNS	11	✗

occurrences, which are then included in our microbenchmarks. The AWSomePy statistics, summarized in Table I, show that S3, DynamoDB, SQS, and SNS are the first, the second, the fifth and the sixth most frequently used services, respectively. The characterization provides the APIs with the highest number of occurrences for all the four targeted services, apart from SNS. In the latter case, we use the API `publish`, which is commonly called to access this notification service.

Our microbenchmarks do not yet include the third and fourth most frequent AWSomePy services, i.e., Lambda and SSM, respectively. From a security perspective, such configuration and management-oriented services need to be taken into consideration. However, they introduce data flows that are difficult to detect statically, and so we leave to future work the implementation of suitable strategies.

Table II summarizes the microbenchmarks. It specifies whether they are intra-procedural or inter-procedural, the relevant AWS services, the type of vulnerability as well as the results of our evaluation (§ V). The microbenchmarks' names include the `boto3` API[3] that is part of the expected data flow. The microbenchmarks include both types of `boto3` interface object, i.e., `client` and `resource`, as they require different processing of the code under analysis. More specifically, since a `resource` provides a high-level object-oriented interface to a cloud service [18], this is normally followed by the initialization of an *intermediate* object, e.g., `Bucket` in the microbenchmark `api-put-object-bucket-assign`, on which the targeted API is called.

B. Information Leakage Example

We categorise the vulnerabilities in our microbenchmark suite into two types: code injection and information leakage. In total our suite currently consists of 7 code injection and 3 information leakage vulnerabilities. An example of a code injection vulnerability is the motivating example from the previous section (§ II-B), which is also the microbenchmark `api-put-item-boto3-client` in our suite. An example of inter-procedural data flow that can lead to information leakage is provided by the microbenchmark `api-publish-wrong-bucket-key`. The architectural diagram in Fig. 2 shows that a HTTP Post request is again the user-controlled entry point of the application. The request is processed by the API Gateway, and triggers the execution of a handler. The latter inspects the request body and identifies

[3]The `boto3` library is the AWS Software Development Kit for Python.

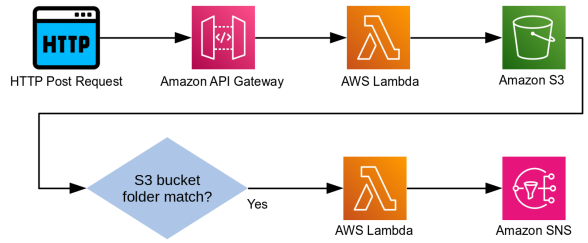

Fig. 2. Microbenchmark `api-publish-wrong-bucket-key`.

a file that is uploaded to a S3 bucket. If the upload operation stores the file in a specific folder, identified by the `prefix` tag of the configuration file, then another handler is executed. This processes the user-provided information, which is included in an email notification sent via the AWS SNS service.

However, the current microbenchmark implementation forces the first handler to access a different folder, thus preventing the execution of the second. We make this choice to enhance the quality of our test suite, as this is a case that might lead to a false positive result. Without sufficiently accurate mechanisms to process infrastructure and application code, in fact, the analysis tool might not be able to discriminate data flows going through different folders within the S3 bucket.

IV. PROTOTYPE ANALYSIS FRAMEWORK

To statically analyse serverless applications, we propose a novel approach that combines information extracted from the application configuration, i.e., the infrastructure code, with the application code. As illustrated by our motivating example (Listing 1), the configuration of a serverless application contains permissions, events and handler-related parameters. These can be used to instrument the application code and obtain a synchronous-like program execution flow. A general-purpose tool capable of identifying data flows can then be used to analyse the instrumented application.

We translate this idea into a prototype analysis tool. In the remainder of this section, we describe its architecture (§ IV-A) and its current implementation (§ IV-B).

A. Architecture

Fig. 3 illustrates the architecture of our five-step analysis pipeline for the identification of security-sensitive data flows. We begin by extracting the set of permissions in the application configuration, which determine whether an API call is allowed (❶). In our motivating example, this means processing the infrastructure code in Listing 1, which specifies the permission necessary (`s3:PutObject`) to execute the API `upload_file` in the handler `onHTTPPostEvent` (Listing 2).

We then extract events and handlers-related information (❷). By relying on a configuration file that lists the events raised by a given API call[4], we are also able to identify the handlers executed as a consequence of that API call. In our

[4]This configuration file is platform-specific, not application-specific.

TABLE II
SUMMARY OF MICROBENCHMARKS SUITE (CI=CODE INJECTION, IL=INFORMATION LEAKAGE, INTER=INTER-PROCEDURAL, INTRA=INTRA-PROCEDURAL).

Microbenchmark	INTER	INTRA	S3	DynamoDB	SQS	SNS	CI	IL	Result
api-publish-wrong-bucket-key	✔	✗	✔	✗	✗	✔	✗	✔	✗
api-put-item-boto3-client	✔	✗	✔	✔	✗	✗	✔	✗	✔
api-put-item-via-file	✔	✗	✔	✔	✗	✗	✔	✗	✗
api-put-item-wrong-table	✔	✗	✔	✔	✗	✗	✔	✗	✗
api-put-object-boto3-client	✔	✗	✔	✗	✗	✗	✔	✗	✔
api-put-object-bucket-assign	✔	✗	✔	✗	✗	✗	✔	✗	✔
api-scan-boto3-client	✗	✔	✗	✔	✗	✗	✗	✔	✔
api-scan-table-assign	✗	✔	✗	✔	✗	✗	✗	✔	✔
api-send-message-boto3-client	✔	✗	✔	✔	✔	✗	✔	✗	✔
owasp-serverless-injection	✗	✔	✔	✗	✗	✗	✔	✗	✔

example, this processing step allows linking the API that stores information in the DynamoDB database with the execution of the handler `onDynamoDBStream` (Listing 3). We then define sources and sinks (❸). Unlike the previous steps, this requires processing both infrastructure *and* application code. All user-controlled application entry points, such as the HTTP Post request visible in Fig. 1, are considered as sources. In contrast, sinks within the scope of the analysis, e.g., `os.system` in Listing 3, are vulnerability-dependent.

To decrease the number of false negatives, i.e., undetected data flows, static analysis tools often rely on type-related information. Therefore, a type annotation step is necessary (❹). Extracting type-related information is a generic, i.e., language-independent, analysis step, but its practical implementation is language-specific. This step is particularly challenging for Python applications, since Python's optional type system implies that type annotations are not mandatory. We therefore rely on the automated type inference feature available in our tool chain (§ IV-B). Due to its limitations though, we also manually specify additional type annotations where necessary, e.g., for the `boto3` client object `s3Client` in Listing 2. We leave to future work the implementation of a fully automated type annotation step.

Finally, events, handlers and permissions-related information are used to synthesize handler calls (❺). These are injected into the application code after cloud service API calls that trigger the execution of other handlers by raising events. In our motivating example, this means modifying `onHTTPPostEvent` by inserting a synthesized call to the handler triggered by the API `upload_file`. Note that the injected code must include at least approximated versions of the input arguments expected by the handlers. This step of our analysis pipeline is not yet fully automated, but we aim to address this in our future work.

B. Implementation

Based on the architecture described in § IV-A, we have created a prototype implementation of a novel analysis tool targeting applications deployed using the Serverless Framework to AWS Lambda. The previously illustrated steps are translated into a collection of Python modules integrated into a command-line utility, which *semi-automatically* modifies the

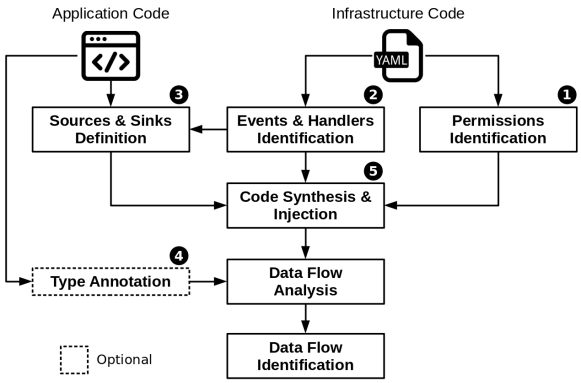

Fig. 3. Serverless Application Static Analysis Pipeline.

code under test. In particular, the type annotation and code synthesis and injection steps are not yet fully automated. Despite this, the output provided by the tool can be used to launch a data flow analysis using Pysa [19]. As Pysa is a Python-specific tool, we currently only support applications implemented in Python.

Pysa ships with a set of *rules* that detail the data flows to be identified. Such rules rely on categorized sources and sinks, which are, in turn, included in Pysa *models*. While we make use of default Pysa rules dedicated to the vulnerabilities of interest, our analysis tool incorporates *custom* Pysa models for the relevant APIs provided by the `boto3` library.

V. EVALUATION

The results of the tests performed with our microbenchmarks are shown in Table II. In seven cases out of ten, our analysis tool is able to detect the expected data flows, which are detailed in the microbenchmarks' documentation. However, given the work-in-progress nature of our tool, it fails to pass three microbenchmarks.

Two failed tests can be classified as *false positives*. The first is `api-publish-wrong-bucket-key`. As previously explained (§ III-B), the handler `onHTTPPostEvent` uploads a file to a S3 bucket folder that is different from the one specified in the `prefix` tag of the application configuration file. Therefore, no handler is triggered as a result

of the `upload_file` API. Our analysis tool, however, does not currently process the `prefix` tag, and, consequently, a non-existing data flow is detected in this case.

The microbenchmark `api-put-item-wrong-table` is the second false positive. Similarly to the previous case, this is due to how we process the application configuration file. In particular, at present, permissions-related information is not associated with specific cloud resources. Consequently, when the function `updateDynamoDBTable` attempts to update a database to which it has no access, an erroneous data flow is detected.

By contrast, the third failed test, i.e., the microbenchmark `api-put-item-via-file`, is a *false negative*, given that Pysa does not identify the expected data flow. This is because of how the tool conducts taint analysis of the code in Listing 4, which is part of the handler `onS3Upload`. While the variables `outputFileObjA` and `outputFileA` acquire the taint from the intended source and the final sink, respectively, Pysa does not merge the partial data flows, despite adding context manager-specific models. We have contacted the Pysa community to determine the root cause of this issue, but, at the time of this writing, it is still unresolved.

```
outputFileA = os.path.join('/tmp', 'outputFileA.txt'
    )
with open(downloadPath, 'r') as inputFileObj, open(
    outputFileA, 'w') as outputFileObjA:
    inputFileContentsList = inputFileObj.readlines()
    outputFileObjA.write(inputFileContentsList[0])
updateDynamoDBTable(outputFileA, outputFileB)
```

Listing 4. Code affected by a taint propagation issue.

VI. Related Work

Serverless Security Frameworks. Datta *et al.* [8] propose Valve, which makes use of taint labels to provide monitoring and enforcement of policy-based information flows. While requiring the deployment of dedicated code, this framework features minimum runtime overhead and does not require modifications to the application, the latter being an important difference with Trapeze [7]. By contrast, Kalium [9] is specifically designed to enforce control-flow integrity and detect manipulations to the order of execution of serverless functions. We emphasize that these frameworks are different from ours, as they require a customized execution environment and cannot inspect a serverless application at coding time.

Flow Analysis Frameworks. FlowDroid [13] is an Android-specific static taint analysis tool that integrates both context and object-sensitivity algorithms. These, along with a model of the Android framework, allow identifying data leaks and malicious applications. Among the language-specific tools, Doop [20] is a Java-focused framework configured declaratively. Similarly to FlowDroid, it supports object-sensitive analyses. By contrast, PolyCruise [21] enables dynamic information flow analysis of multilingual software systems. None of these frameworks, however, capture the specifics of modern serverless platforms.

VII. Conclusion and Future Work

In this work, we present a new approach to statically analysing serverless applications. To underpin its development, we design a novel suite of security-oriented microbenchmarks, which we publicly release. The tests performed with a preliminary implementation of our analysis technique show that the expected data flows are detected for seven out of ten microbenchmarks. This result confirms the feasibility of the proposed approach.

In future work, we aim at fully automating the type annotation as well as the code synthesis and injection steps of our analysis pipeline by processing the abstract syntax tree of the application code. Moreover, in order to minimize the number of false positives, we plan to improve the extraction of information from the application configuration file. Finally, we intend to expand the capabilities of our tool by supporting a higher number of APIs together with configuration and management-oriented services, and to evaluate its performance on real-world applications.

References

[1] AWS. (2023). [Online]. Available: https://aws.amazon.com/
[2] J. Lepiller, R. Piskac, M. Schäf, and M. Santolucito, "Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2021.
[3] J. Daly. (2020). [Online]. Available: https://www.jeremydaly.com/event-injection-protecting-your-serverless-applications/
[4] CSA. (2019). [Online]. Available: https://cloudsecurityalliance.org/blog/2019/02/11/critical-risks-serverless-applications/
[5] J. Katzer, *Learning Serverless*, 1st ed. O'Reilly Media, Inc., 2020.
[6] PortSwigger. (2022). [Online]. Available: https://portswigger.net/daily-swig/insecure-amazon-s3-bucket-exposed-personal-data-on-500-000-ghanaian-graduates
[7] K. Alpernas, C. Flanagan, S. Fouladi, L. Ryzhyk, M. Sagiv, T. Schmitz, and K. Winstein, "Secure serverless computing using dynamic information flow control," *Proc. ACM Program. Lang.*, 2018.
[8] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, and A. Bates, "Valve: Securing function workflows on serverless computing platforms," in *Proceedings of The Web Conference 2020*, 2020.
[9] D. S. Jegan, L. Wang, S. Bhagat, and M. Swift, "Guarding serverless applications with kalium," in *USENIX Security 23*, 2023.
[10] M. Obetz, A. Das, T. Castiglia, S. Patterson, and A. Milanova, "Formalizing event-driven behavior of serverless applications," in *Service-Oriented and Cloud Computing*, 2020.
[11] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *USENIX ATC 18*, 2018.
[12] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "Faasdom: A benchmark suite for serverless computing," in *DEBS '20*, 2020.
[13] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *PLDI '14*, 2014.
[14] OWASP. (2021). [Online]. Available: https://owasp.org/www-project-serverless-top-10/
[15] A. Bhargav. (2018). [Online]. Available: https://medium.com/appsecengineer/dynamodb-injection-1db99c2454ac
[16] G. Raffa, J. B. Alis, D. O'Keeffe, and S. K. Dash, "Awsomepy: A dataset and characterization of serverless applications," in *SESAME '23*, 2023.
[17] SF. (2023). [Online]. Available: https://www.serverless.com/
[18] R. Bolovan, 2018. [Online]. Available: https://realpython.com/python-boto3-aws-s3/
[19] Pysa, 2023. [Online]. Available: https://pyre-check.org/docs/pysa-basics/
[20] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *OOPSLA '09*, 2009.
[21] W. Li, J. Ming, X. Luo, and H. Cai, "PolyCruise: A Cross-Language dynamic information flow analysis," in *USENIX Security 22*, 2022.