USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

CloudFlow: Identifying Security-sensitive Data Flows in Serverless Applications

Giuseppe Raffa, *Royal Holloway, University of London;* Jorge Blasco, *Universidad Politécnica de Madrid;* Dan O'Keeffe, *Royal Holloway, University of London;* Santanu Kumar Dash, *University of Surrey*

https://www.usenix.org/conference/usenixsecurity25/presentation/raffa

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

CloudFlow: Identifying Security-sensitive Data Flows in Serverless Applications

Giuseppe Raffa
Royal Holloway University

Jorge Blasco
Universidad Politécnica

Dan O'Keeffe
Royal Holloway University

Santanu Kumar Dash
University of Surrey

Abstract

The serverless computing paradigm has significantly changed how modern cloud applications are developed. This model allows developers to focus on application business logic while outsourcing to the cloud provider infrastructure details such as machine provisioning. However, the serverless model also presents new security challenges. Among these, static analysis of application security, a fundamental part of the secure software development lifecycle, becomes more complex due to the presence of event-triggered code and the black-box nature of cloud services.

In this paper, we present `CloudFlow`, a novel framework to statically detect security-sensitive data flows in serverless applications. To achieve this, `CloudFlow` leverages the infrastructure definition provided by the developer to identify the events, permissions and entry points of an application. Using this information and custom models for events and cloud API calls, it instruments the application code, which can then be analysed with general-purpose methods for static analysis. We evaluate our framework against a new suite of 40 microbenchmarks, `CloudBench`. Furthermore, we analyse 104 real-world applications selected from a recent dataset. To the best of our knowledge, this is the largest security-focused analysis of serverless applications to date. Our results show that `CloudFlow` passes all microbenchmarks, apart from three, and detects 11 code injection and information leakage vulnerabilities in real-world applications. Both `CloudFlow` and `CloudBench` are open-source to support future research.

1 Introduction

In recent years, a growing number of enterprises have adopted the serverless computing paradigm to develop a variety of cloud-based applications. Serverless platforms are an attractive option because they simplify the development process by reducing operational overheads. In the serverless model, cloud providers, e.g., Amazon [5], Microsoft [31], and Google [20], are responsible for provisioning, scaling, and maintaining the underlying infrastructure. Furthermore, the numerous backend services [27, 33], such as database management systems and data storage solutions, facilitate the implementation of a diverse range of applications, including image processors, web and chatbot services.

However, recent data breaches [38–40], academic research [13] and reports by industry experts [12, 36] show that securing serverless applications is challenging. According to the well-known shared responsibility model [26], developers are responsible for the security of their cloud applications, which typically feature a large attack surface. This is especially the case for serverless applications, which usually receive inputs from a variety of sources and interact with a wide range of internal and external services and endpoints. To improve the cloud provider-developed tool ecosystems [29, 59], researchers have proposed frameworks focused on dynamic information flow [3, 14, 23], customized access control models [51] and provenance [15]. Moreover, model-based techniques for integration-level testing of serverless applications have also been explored [66–69]. Notably though, these frameworks and testing techniques require the deployment of additional code and runtime information, respectively.

Static analysis enables developers to inspect an application *prior* to deployment. While undoubtedly advantageous, analysing serverless applications statically is challenging for several reasons. First, serverless applications are event-driven, i.e., they consist of a set of short-running stateless functions triggered, for example, by HTTP requests and database updates. This makes static analysis harder because functions are also connected via cloud interactions that are not part of the synchronous execution of the application. Second, application permissions can be configured using a wide range of mechanisms and are generally rather complex. Finally, the code of platform services is proprietary and cannot be analysed by the developer. While taking important first steps, previous works on call graphs [34, 35] and security-sensitive data flows [47] only partially address these challenges, as they do not fully take into account how the cloud infrastructure is defined, and require developers to manually model the interactions with

backend services.

In this paper, we present a fully automated framework, `CloudFlow`, to statically detect security-sensitive data flows in serverless applications. `CloudFlow` begins by processing the infrastructure code, which specifies the required cloud-based resources along with a host of configuration parameters. It then analyses the application code, which defines the business logic of the application. The information gathered enables `CloudFlow` to automatically instrument the code under test, thus transforming the underlying asynchronous system into a synchronous equivalent. The key advantage of this technique is that the instrumented code can be inspected with a general-purpose static analysis tool capable of detecting security-sensitive data flows.

We evaluate `CloudFlow` against a new suite of security-oriented microbenchmarks, `CloudBench`. Furthermore, we analyse 104 real-world AWS applications implemented in Python and compatible with the Serverless Framework [57]. Our results show that `CloudFlow` passes 92.5% of the microbenchmarks and identifies 11 confirmed vulnerabilities in the inspected applications.

In summary, we make the following contributions:

- We propose a novel, automated framework for the static detection of security-sensitive data flows in serverless applications. This allows application developers to identify possible security vulnerabilities before their code is deployed to the cloud.
- We release a new suite of security-oriented serverless microbenchmarks. Our suite can be extended and used to test future static analysis tools focused on identifying code security properties for serverless environments.
- We analyse with our framework a total of 104 real-world applications, obtained from a recent dataset. To the best of our knowledge, this is the largest security-focused analysis of serverless applications published so far.

The paper is structured as follows. After providing the reader with the required background information in §2, which also encompasses a motivating example, we present `CloudFlow` in §3. The results of our microbenchmarks-based evaluation are illustrated in §4, whereas our real-world application analysis is detailed in §5. The obtained results are further discussed in §6. We compare our work with previous research in §7, and we finally provide our conclusions in §8.

2 Background

2.1 Serverless Computing Model

The serverless computing model, also referred to as *FaaS* (Function-as-a-Service), enables the execution of applications that leverage backend services fully managed by a cloud provider [26]. Due to platform-imposed limits on execution times, serverless applications are commonly implemented by integrating several event-triggered, short-running, and stateless functions, also known as *handlers*. These provide the application functionality, and constitute the *application code*. Such applications also require the definition of a set of cloud-based resources, e.g., file repositories and databases. These are specified in the *infrastructure code*, along with permissions, events, and handler-related configuration parameters.

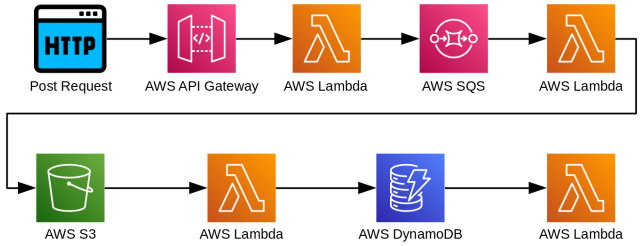

Figure 1: Example application with an event-driven data flow. The four event-triggered handlers are identified by the blocks AWS Lambda.

If supported by the serverless environment, the infrastructure code can additionally include *event filters*. These allow triggering the execution of code only when additional conditions are met, e.g., the update of a specific folder in a cloud-based repository. Furthermore, depending upon the adopted deployment technology, the infrastructure definition can also include *extension plugins*. While simplifying some developer tasks, these affect the definition of cloud resources as well as the application architecture and permissions.

There are both declarative and imperative approaches to the development of infrastructure code, although recent research indicates that declarative approaches are more popular [17, 46, 47]. Regardless of the specific implementation choices, we emphasize that the infrastructure code contains security-related information that is critical for static analysis of serverless applications.

2.2 Motivating Example

To illustrate the serverless-specific challenges of static analysis for security, we next present an example application that contains a security-sensitive data flow that leads to a code injection vulnerability. As shown in the architectural diagram of Fig. 1, its functionality relies on multiple cloud services, along with four event-triggered handlers[1]. A YAML file specifies the required permissions, such as those visible in Listing 1, as well as events and handler-related configuration parameters.

[1]The widely adopted Serverless Framework [57] is the deployment tool of choice for the application. However, the concepts discussed are generic, as they are not platform or deployment technology-specific.

Upon receiving a HTTP Post request from a user, a handler that sends a message to a queue is triggered (Listing 2). As a consequence, the SQS service executes the consumer handler `onSQSMessage` (Listing 3). The handler processes the last received message, which contains user-specified data, and uploads a local file to a cloud-based repository by calling the `upload_file` API. The upload, in turn, triggers the execution of the `onS3Upload` handler (Listing 4), which reads the newly uploaded file and then updates a database by calling the auxiliary function `updateDynamoDBTable`. On detecting this, the DynamoDB service triggers the `onDynamoDBStream` handler (Listing 5), which passes a string initialized from the newly added database item to `os.system`. As a result, the application executes a system command with user-controlled input.

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

Listing 1: Application permissions specified in the configuration file.

```
def onHTTPPostEvent(event, context):
    # After initialization statements
    try:
        sqs = boto3.resource('sqs')
        queue = sqs.Queue(os.getenv('QUEUE_URL'))
        queue.send_message(MessageBody=event['body'])
        message = 'Message accepted!'
    except Exception as e:
        message = str(e)
        statusCode = 500
    return {'statusCode': statusCode, 'body': json.dumps({'message': message})}
```

Listing 2: Handler triggered by a HTTP request.

```
def onSQSMessage(event, context):
    # After initialization statements
    uploadFileFolder = 'uploads'
    uploadFileName = msgBodyDict + '.txt'
    s3BucketKey = os.path.join(uploadFileFolder, uploadFileName)
    localFile = os.path.join('/tmp', uploadFileName)
    s3_client = boto3.client('s3')
    s3_client.upload_file(localFile, os.environ['BUCKET_NAME'], s3BucketKey)
    return
```

Listing 3: Handler triggered by a SQS message.

```
def onS3Upload(event, context):
    # After initialization statements
    s3 = boto3.resource('s3')
    downloadPath = os.path.join('/tmp', os.path.split(fileName)[-1])
    s3.Object(bucketName, fileName).download_file(downloadPath)
    with open(downloadPath, 'r') as inputFileObj:
        inputFileContentsList = inputFileObj.readlines()
    updateDynamoDBTable(inputFileContentsList[0], inputFileContentsList[1])
    return
```

Listing 4: Handler triggered by the S3 API `upload_file`.

```
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 5: Handler triggered by a DynamoDB database update.

Our example, which is one of the microbenchmarks included in `CloudBench`[2], highlights three key challenges for static detection of security-sensitive data flows. First, considering the effect of events raised by cloud API calls is crucial (❶). However, general-purpose tools do not support this since they analyse only the source code, which does not contain any event-related information, e.g., the fact that the API call `send_message` in Listing 2 triggers the execution of the handler `onSQSMessage`. Such information is present exclusively in the infrastructure file, which must therefore be suitably processed prior to the execution of the analysis. Furthermore, events are propagated via objects, i.e., the handlers' `event` inputs, that encapsulate complex, service-dependent data structures. To incorporate these into the analysis, specific models are required. Their absence prevents a general-purpose tool from accomplishing this task.

The second challenge is the multitude of mechanisms available to define the application permissions (❷). For example, the AWS environment features identity and resource policies, along with time-constrained roles and attribute-based access control [59]. Supporting such mechanisms is crucial for providing accurate results.

Finally, the third key challenge is that the developer cannot include in their analysis the proprietary code that implements the cloud-based services (❸). Since these are instrumental in the functionality provided by a serverless application, models are necessary in this case as well.

In summary, the limitations of general-purpose static analysis tools for event-driven serverless applications motivate the need for serverless-specific approaches to identify security-sensitive data flows. Previous research in other ecosystems has shown that some event-driven architectures can be analysed with *augmented* call graphs [28] and specialized testing techniques [16, 18]. However, existing solutions cannot be applied to serverless platforms, as their design is based on

[2] `api-send-message-queue-assign`.

the Node.js and Android programming models. As such, they do not process the complex infrastructure-related information essential to the analysis of serverless applications.

3 CloudFlow

To address the challenges elucidated in §2.2, we propose `CloudFlow`, a fully automated framework for the static detection of security-sensitive data flows in serverless applications. `CloudFlow` transforms an event-driven serverless application into a synchronous equivalent that can be analysed with an existing static analysis tool. To achieve this, `CloudFlow` combines information extracted from the infrastructure and the application code.

In this section, we begin with an overview of `CloudFlow` (§3.1). We then describe in detail how `CloudFlow` processes infrastructure code (§3.2) and application code (§3.3). Finally, we explain how the aforementioned synchronous representation (§3.4) is obtained, and how the static analysis models required to identify security-sensitive data flows are prepared (§3.5).

3.1 Overview

The high-level diagram in Fig. 2 shows the three main stages of `CloudFlow`'s execution, each of which consists of a number of steps. The first stage of `CloudFlow` analyses the infrastructure and unmodified application code. It begins by extracting configuration parameters, e.g., environment variables, from the infrastructure code (*Configuration Parameters*). `CloudFlow` then extracts permissions, events, and handler-related information (*PEH Identification*). Upon completion of these steps, it is possible to identify the allowed cloud API calls, the events processed by the application, and the handlers executed as a consequence of these events. Some of this information is used in our event object models, which are configured later as part of the `CloudFlow` synchronous representation stage (§3.4). The infrastructure code also specifies user-controlled entry points of the application, i.e., *sources*, along with security-sensitive *sinks*, e.g., cloud-hosted repositories and databases, which `CloudFlow` next extracts (*Sources & Sinks Definition*). We note that processing the infrastructure code is the initial step towards addressing challenge ❶. Moreover, extracting permission-related information, such as that shown in Listing 1, helps address challenge ❷.

To complement the infrastructure code processing, we also analyse the application code. This enables us to identify cloud API calls (*Cloud API Calls Identification*) and to extract the information included in their inputs (*Cloud API Calls Models*). These are processed via models that we develop starting from the API documentation. Specifically, we rely on the API documentation to identify relevant API call inputs, such as names of cloud-based resources. We highlight that analysis of these inputs is required to complete the configuration of our event object models, and is, therefore, another step towards addressing challenge ❶. The API call models, instead, are our solution to challenge ❸, as they allow incorporating an approximation of the cloud service functionality into our approach. Finally, we observe that language-specific steps might also be needed to complete the processing of the application code (*Language-specific processing*). This can provide the underlying static analysis tool with additional information that improves the accuracy of static analysis. For example, dynamic languages (e.g., Python) can benefit from an additional automated type annotation step (§3.3).

The second stage of `CloudFlow` focuses on creating a synchronous representation of the analysed application. This enables general-purpose static analysis tools to analyse the application. Starting from the event object model (*Event Object Model*), the framework synthesizes (*Code Synthesis*) and then injects the instrumentation code (*Code Injection*). Specifically, for each cloud API that triggers the execution of a handler, we inject two distinct code statements. The first initialises the event object, whilst the second is an *explicit* handler call where the event object is one of the inputs. This step helps us transform an event-driven asynchronous application into a synchronous one for the purpose of static analysis. Considering the importance of this representation, we provide a detailed example in §3.4.

The last stage of `CloudFlow` creates the static analysis models necessary for detection of security-sensitive data flows. These models are required by the underlying analysis tool to identify sources and sinks. `CloudFlow` is capable of generating application-specific models (*Application-specific Model Generation*). They target, for instance, the public API endpoints of the application under test. In addition, we develop cloud API-specific models, which are, in contrast, application-independent (*Cloud API Model Retrieval*). To minimize the number of false positives, the latter category of models is also processed to select only those relevant to allowed cloud APIs (*Permission-based Model Selection*). Once all the required static analysis models are available, the instrumented application can be analysed by running a general-purpose tool with data flow detection capabilities (*Data Flow Identification*).

`CloudFlow`'s high-level architecture is not specific to a particular cloud platform. However, our current implementation targets AWS Lambda [7] applications deployed using the Serverless framework deployment tool [54]. Similarly, `CloudFlow` can in principle be used with any general-purpose static analysis tool, but we focus for the remainder of this paper on the analysis of Python applications using Pysa, an interprocedural static analysis framework [43]. We next describe in more detail each stage of `CloudFlow`'s execution.

3.2 Infrastructure Code Processing

As stated in §2.2, serverless applications require the definition of cloud resources that will be deployed for the appli-

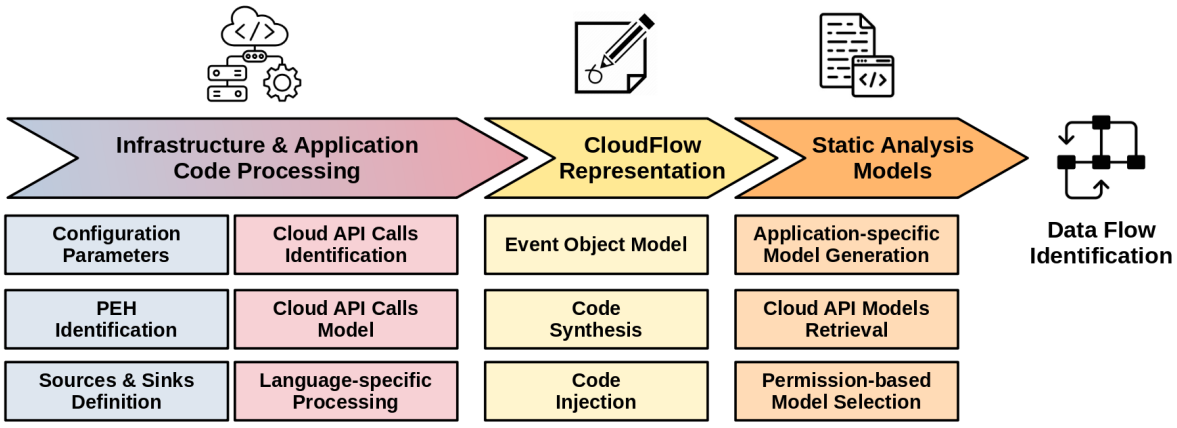

Figure 2: Proposed static analysis pipeline for serverless applications (PEH=Permissions, Events and Handlers).

cation along with their configuration. `CloudFlow` processes these definitions automatically to extract relevant information that will be later used to build the representation necessary to perform data flow analysis. Serverless applications normally define infrastructure code within YAML files. We begin their processing with a custom YAML parser that first stores the infrastructure code in a tree-like data structure. This is then recursively traversed with the strategy illustrated by Zaczyński [9]. During the traversal, we use regular expressions to detect configuration parameters that need to be resolved, e.g., cloud-based repositories' names that depend on the application development stage. To resolve such parameters, we retrieve the necessary values from the YAML file or by accessing, if needed, external files referenced within the infrastructure definition. This step is important to the precision of our analysis, as the Serverless Framework allows specifying resource-related configuration data through parameters included in the infrastructure code. The latter is finally mapped into a new data structure from which we extract security-relevant information. This includes, for example, associations between events and their triggered handlers, and permissions with cloud-based resources. To obtain this information, `CloudFlow` leverages an abstraction layer that embeds the YAML tags and structures used for permissions, events, handlers, plugins, and their configurations.

To better analyse the execution context of the application under test and increase the precision of the analysis, `CloudFlow` also extracts information about environment variables and event filters. Environment variables are frequently used to specify cloud API inputs. For example, the API `upload_file` in Listing 3 is called with the bucket name input set to `os.environ['BUCKET_NAME']`. In contrast, event filters are analysed because they configure the AWS infrastructure to raise events only under specific conditions (§2.1). In the case of the S3 service, for instance, a filter defined as `prefix: uploads/` implies that a given handler is executed only if the event originates from the folder `uploads`.

The functionality provided by the deployment tool can be extended with plugins, which can also be modeled and incorporated in `CloudFlow`. As previously mentioned (§2.1), such plugins can affect both the application architecture and its permissions. For this reason, plugin modeling is an important practical contribution of our framework. Our plugin models combine the logic detailed in their documentation with information obtained from the YAML file. As a result, they encapsulate data structures that can be queried by other components to refine their analyses. `CloudFlow` features full support for the security-oriented plugin that enables handler-level permissions [55]. In addition, we provide basic support for the plugin dedicated to AWS step functions [56], which is among the most frequently used in real-world applications [46].

3.3 Application Code Processing

Serverless applications access the cloud-based platform through language-specific Software Development Kits (SDKs), such as the Python library `boto3` [10]. These provide interface objects, e.g., `sqs` in Listing 2, on which service-specific APIs can be called[3]. Therefore, `CloudFlow` processes the Abstract Syntax Tree (AST) of the application code to identify such objects and, subsequently, the cloud API calls. Their security-relevant inputs are then extracted with documentation-based models. Importantly, to increase the precision of our analysis, we also attempt to statically *resolve* the values of the cloud API inputs by using the configuration parameters obtained from the infrastructure code (§3.2).

Finally, since our current implementation focuses on Python, our framework also adds type annotations as necessary, as the language includes only an optional type system.

[3]The `boto3` library provides two types of interface objects, i.e., `client` and `resource` [48]. `CloudFlow` supports both of them, and manages the differences through configuration files.

The absence of type-related information, in fact, prevents static analysers from identifying issues, thus leading to false negatives. `CloudFlow` relies on the automated type inference feature of Pyre [41], the static type checker on which Pysa is built, to add type annotations. However, Pyre supports only the core language types, such as lists and dictionaries, and cannot be used to insert the necessary `boto3`-specific type annotations. As illustrated by the example provided in §3.4, to overcome this limitation, we include these additional type annotations by modifying the AST nodes of the SDK-provided interface objects.

3.4 CloudFlow Representation

The purpose of this stage is to provide a synchronous-like program execution flow, which we achieve in three steps. First, the previously extracted information is used to populate the relevant event object model[4]. Second, the output provided by the event object model is synthesized as an AST node and then injected as part of an initialization statement. Third, a call to the triggered handler, featuring the event object among its inputs, is AST-synthesized and added to the application code. This three-step process is repeated for each cloud API call that is allowed by the application permissions and that raises events triggering the execution of other handlers.

To further clarify these important steps, Listing 6 shows how we instrument the `upload_file` API call in our motivating example. In this case, the initialization statement (lines 9-11) contains the event name, obtained from the infrastructure code, and information on the cloud-based repository (i.e., bucket) extracted from the API call inputs. This is followed by an explicit call to the triggered handler, i.e., `onS3Upload`, at line 12. Note that the instrumentation code is injected *after* the cloud API call site to emulate a standard function call.

```
1  def onSQSMessage(event, context):
2      # After initialization statements
3      uploadFileFolder = 'uploads'
4      uploadFileName = msgBodyDict + '.txt'
5      s3BucketKey = os.path.join(uploadFileFolder,
        uploadFileName)
6      localFile = os.path.join('/tmp',
       uploadFileName)
7      s3_client: S3Client = boto3.client('s3')
8      s3_client.upload_file(localFile, os.environ[
       'BUCKET_NAME'], s3BucketKey)
9      event = {'Records': [{'eventName': '
       ObjectCreated:*', 's3': {'bucket':
10         {'name': os.environ['BUCKET_NAME'], 'arn
       ': os.environ['BUCKET_NAME'
11         ]}, 'object': {'key': s3BucketKey}}}]}
12     s3uploadhandler.onS3Upload(event, context)
13     return
```

Listing 6: Handler with instrumented API call. Lines 9-11 show the initialization statement injected for the event object.

[4]An event object model contains an approximation of the data structure encapsulated in the platform event. The documentation folder of the `CloudFlow` repository includes event object models in JSON format.

The explicit call to the triggered handler is injected on line 12. Note that the `boto3` interface object `s3_client` includes the type annotation `S3Client` added by `CloudFlow` (line 7).

3.5 Static Analysis Models

`CloudFlow` affords a layer of automation that sits atop Pysa. To the best of our knowledge, the latter is the only Python-specific tool designed to detect inter-procedural and security-sensitive data flows. Pysa relies on sources and sinks classified in categories, e.g., user-provided and remote code execution, and included in *models*. These provide additional contextual information by specifying, for instance, function signatures and class definitions. The targeted data flows are detailed via *rules*, where the relevant sources and sinks are listed, along with optional metadata, e.g., intermediate function calls.

Our framework leverages the set of models and rules shipped with Pysa. While these constitute an important starting point, they are generic and need to be complemented with both application and cloud API-specific models. For this reason, `CloudFlow` automatically generates Pysa models that mark as user-provided sources the event objects processed by all the handlers. This approach, which relies on the analysis of the application code AST to extract the handlers' signatures, allows us to analyse two sets of events. The first includes those intended to be controlled by an user, whereas the second contains events that become user-provided because of erroneous or malicious workflow manipulations [37].

As regards the cloud API-specific models, we develop them after identifying `boto3`-specific security-sensitive sinks, for instance the DynamoDB API `scan` [1]. Such models complement those provided by Pysa for generic security-sensitive sinks, such as `os.system` (Listing 5). It is noteworthy that our custom Pysa models are automatically disabled if they target cloud APIs disallowed by the application permissions. This is an additional serverless-specific feature of our framework that reduces the number of false positives.

4 Evaluation

Previous research shows that security tools can be tested and compared effectively using microbenchmarks [6]. Since existing serverless benchmarks are designed to assess platform performance [30, 64] and cost effectiveness [8], we develop a new security-oriented microbenchmark suite, `CloudBench`. We next introduce `CloudBench` (§4.1), and then present our results (§4.2).

4.1 CloudBench

`CloudBench` consists of 40 AWS and Serverless Framework-compatible Python applications, which are categorized in Table 1. We observe that 30 of them, which represent 75% of

the total number, are inter-procedural. This means that the expected source and sink are in different handlers, and the security-sensitive data flow between them *always* includes event-triggered code. As explained in our motivating example (§2.2), which is one of the microbenchmarks included in this category, inter-procedural data flows are harder to detect. We therefore choose by design to focus on them in order to enhance the quality of our suite. We note that Table 1 also provides a measure of the microbenchmarks' complexity in terms of lines of code (LOC). In the inter-procedural case, these range from 20 to 70.

In addition, `CloudBench` contains nine intra-procedural microbenchmarks. Albeit structurally simpler, as reflected in the lower number of LOC, they include important cases, such as NoSQL injection [1] and variants of an insecure code example described in the OWASP Serverless project documentation [36]. To improve the overall coverage, we also add a simple application with 144 LOC, obtained by modifying a baseline [53] that includes an AWS step function.

Our characterization table also shows that five AWS services are supported[5]. Four of them, i.e., S3, DynamoDB, SQS and SNS, are among the most frequently occurring in the dataset used for our real-world applications analysis (§5). Additionally, the aforementioned simple application includes the Serverless Framework plugin dedicated to the AWS step function service (§3.2). A total of 14 analysed applications rely on this plugin, among the highest number of occurrences in the reference dataset.

Regarding vulnerabilities, also reported in Table 1, twenty-seven `CloudBench` microbenchmarks contain a code injection vulnerability, while five focus on information leakage and integrity. The remaining eight do not contain any vulnerability and are designed to capture cases that can lead to false positive results. We emphasize that the vulnerabilities considered, although not serverless-specific, are listed among the most critical risks for serverless applications by the Cloud Security Alliance [12] and Podjarny *et al.* [37].

Finally, we highlight that `CloudBench` includes in total 30 new microbenchmarks in comparison with the suite recently published by Raffa *et al.* [47]. We expand both the inter-procedural and intra-procedural category, add a simple application, and provide support for features not included in their work, such as permission-relevant deployment tool plugins, event filters, and environment variable-based configuration.

4.2 Results

The results of our microbenchmark evaluation are reported in Table 2. `CloudFlow` succeeds in identifying the expected data flows in 37 cases out of 40. As further detailed in the ablation study below, these results show that our framework improves the baseline provided by previously published work [47]. The three failed inter-procedural microbenchmarks, which include one false positive and two false negatives, are also analysed in the remainder of this section.

Ablation study. The number of microbenchmarks passed grows from 27 in the baseline framework [47] to 37 in the `CloudFlow` evaluation, with the highest increase, i.e., eight microbenchmarks, in the inter-procedural category. To explain this result, we note that the baseline framework does not support the following three `CloudFlow` features: environment variable inspection, plugin and resource analysis. We therefore selectively disable them to identify those that, either singly or in combination with others, determine a change in the result of a microbenchmark execution.

As summarized in Table 2, the most frequently occurring feature is plugin analysis, which enables detection of the expected data flow in four inter-procedural microbenchmarks, one intra-procedural, and the simple application. We note that the plugin analysis feature is sufficient to successfully execute all of them. Our study also shows that resource analysis is the second most frequent feature, with four occurrences in total. Enabling this feature alone identifies the target data flow in a single case. In the remaining three, both resource analysis and environment variable inspection have to be enabled in `CloudFlow` to pass the microbenchmarks.

False positive. We obtain a false positive result with the `api-publish-wrong-bucket-key` microbenchmark. This includes a handler, shown in Listing 7, that is triggered when the application receives a HTTP Post request. After inspecting the body of the latter, the code identifies and then uploads a file to a S3 bucket. However, the microbenchmark includes an event filter, and the upload operation raises an event only if the file is stored in a specific folder, configured via the YAML tag `prefix`. Despite inspecting the inputs of the `upload_file` API, `CloudFlow` is unable to obtain a fully resolved value for the variable `s3BucketKey`. This, in fact, is initialized through a call to `os.path.join`, and cannot therefore be compared with the relevant folder name in the configuration file.

```
def onHTTPPostEvent(event, context):
    msgBodyDict = event['body']
    uploadFileFolder = 'upload-folder'
    uploadFileName = msgBodyDict + '.txt'
    s3BucketKey = os.path.join(uploadFileFolder,
     uploadFileName)
    localFile = os.path.join('/tmp',
    uploadFileName)
    s3_client = boto3.client('s3')
    s3_client.upload_file(localFile, os.environ[
    'BUCKET_NAME'], s3BucketKey)
    return
```

Listing 7: Handler in `api-publish-wrong-bucket-key`.

We note that our framework analyses cloud API calls

[5]The microbenchmarks are organized in their repository according to the main targeted cloud service. All of them though, except for the intra-procedural ones, include multiple cloud services.

Table 1: Summary of the CloudBench microbenchmarks suite (CI=Code Injection, II=Information Integrity, IL=Information Leakage, LOC=Lines Of Code, NN=None).

Category	Amount	S3	DynamoDB	SQS	SNS	Step Funct.	CI	II	IL	NN	Min LOC	Max LOC
Inter-procedural	30	30	14	3	3	0	22	0	2	6	20	70
Intra-procedural	9	6	3	0	0	0	4	1	2	2	5	21
Simple Applications	1	0	1	0	0	1	1	0	0	0	144	144

Table 2: Summary of CloudFlow evaluation (EVI=Environment Variable Inspection, PLA=Plugin Analysis, REA=Resource Analysis).

Microbenchmarks		Baseline		CloudFlow		CloudFlow Feature Contributions		
Category	Total No.	PASS	FAIL	PASS	FAIL	EVI	PLA	REA
Inter-procedural	30	19	11	27	3	3	4	4
Intra-procedural	9	8	1	9	0	0	1	0
Simple Applications	1	0	1	1	0	0	1	0

by first checking whether the application has the required permissions, and then refines the result by extracting additional information, e.g., event filtering-related, from the API inputs. When obtaining such information is not possible, CloudFlow exclusively relies on the API permissions to minimize the number of false negatives. As a result, a non-existing data flow is detected in this case.

False negatives. The first false negative of our evaluation is provided by the api-put-item-via-file microbenchmark. In this case, Pysa is not able to identify the expected data flow due to a taint propagation issue affecting the code in Listing 8, included in the handler onS3Upload. While the taints from the intended source and sink are correctly propagated to the variables outputFileObjA and outputFileA, respectively, the two partial data flows are not merged. We note that context manager-specific models, which are now part of CloudFlow, are not sufficient to rectify this issue. After contacting the Pysa community for support[6], we confirmed that this is a limitation of the static analysis engine, which treats the two variables as independent. Consequently, no taint is propagated between them. For the expected data flow to be successfully detected, the code in Listing 8 would need to be re-structured.

```
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 8: Code affected by a taint propagation issue.

The second false negative result is obtained with the microbenchmark api-publish-boto3-client, which includes the handler onS3Upload also present in our motivating example (§2.2). Surprisingly, only the following three partial data flows are reported by Pysa: (i) between the intended source (located in the handler onHTTPPostEvent) and the with statement in Listing 4, (ii) between the onS3Upload input event and the final sink (located in the handler onDynamoDBStream), and (iii) between the onS3Upload input event and the with statement in Listing 4. Our investigation shows that both the forward and the backward taints associated to the variables downloadPath, inputFileObj, and inputFileContentsList are correct. Since all the above partial data flows are identified through the same Pysa rule, we hypothesize that this false negative is caused by a tool post-processing configuration parameter that we were unable to identify.

5 Real-world Application Analysis

In this section, we present the analysis of 104 real-world applications conducted with CloudFlow. Our objectives are twofold. First, to comprehensively evaluate our framework, we want to ascertain whether it is capable of analysing real-world applications, which implies assessing the caused overhead as well. Second, we aim to gain insight into the types of security vulnerability that can affect such applications. We note that previous research on serverless static analysis does not explore security, instead focusing on call graph generation [34, 35] and performance evaluation [65].

We start with the results of the initial dataset triage (§5.1). Next, we show the identified security-sensitive data flows and vulnerabilities found (§5.2). The latter are further illustrated with two case studies (§5.3). Finally, we present the results of our performance evaluation (§5.4).

[6] https://github.com/facebook/pyre-check/issues/797

5.1 Dataset Triage

Our real-world application analysis is based on the AWSomePy dataset, which consists of 145 Python serverless applications collected in 2023 [46]. To ensure that our analysis is performed on applications without inconsistencies and fully compatible with `CloudFlow`, we conduct an initial triage of this dataset. This triage step filters out 41 applications. As detailed in Table 3, we classify 23 of them as templates (not real applications), since they contain examples demonstrating design patterns or the functionality offered by specific cloud services.

Table 3: Summary of the AWSomePy dataset triage.

Category	Total No.
Template	23
`CloudFlow`-incompatible	6
Pysa-incompatible	6
Infrastructure code inconsistent	4
Application code inconsistent	2

We also identify a total of 12 applications that are incompatible with our framework. Specifically, six of them were developed with an obsolete version of Python or relied on a legacy API system instead of the `boto3` SDK. The other six applications are Pysa-incompatible because the names of files or folders contain either characters or language keywords not allowed in Python identifiers. As a result, the generated Pysa models are invalid. To remedy this problem, we contacted the Pysa community, but, at the time of writing, it is still unresolved[7].

A further four applications have inconsistent infrastructure code because of missing configuration parameters, references to unavailable application modules, and commented out sections. Similarly, we identify two cases of inconsistent application code caused by Python syntax errors and reliance on code not included in the repository.

5.2 Security-sensitive Data Flows

`CloudFlow` detects a total of 205 security-sensitive data flows in the remaining 104 applications analysed. Note that a security-sensitive data flow will not always result in a security vulnerability. For instance, code deployed to perform a data transfer between two AWS S3 buckets may be considered an information leakage vulnerability if the transferred information includes personal or sensitive data. Since the nature of the information cannot be deduced via static analysis, it is therefore marked as a security-sensitive data flow that has to be manually reviewed to verify if it leads to a security vulnerability. Because of this, all the data flows identified by CloudFlow were manually validated, and then categorized as shown in Table 4. Our analysis reveals that 11 of them, present in five different applications, are actual vulnerabilities. We note that four are detected via custom Pysa models automatically generated by our framework, while none of them include code injected by `CloudFlow`.

In contrast, a total of 64 data flows that could not be classified with certainty are included in the category *Potential Vulnerability*. We opt for this classification in two cases. First, when the data flow involves a large number of function calls, which may or may not be event-triggered. Second, when previously deployed and configured cloud-based resources are used. The remaining data flows (130) are categorized as *No Vulnerability*, since we find evidence that they are not a security concern.

To gain additional insight into the vulnerabilities, we further classify them into three types as illustrated in Table 5. It is noteworthy that three applications exhibit five code injection vulnerabilities caused by a HTTP request-triggered data flow. As shown in our motivating example (§2.2), these exist because user-controlled, unsanitized parts of the HTTP requests are used to initialize commands run via standard library functions, such as `subprocess.Popen` and `subprocess.check_output`. Interestingly, we also identify one application where code injection is possible because unchecked command-line (CLI) parameters are used to modify shell scripts executed via `os.system`.

The remaining four vulnerabilities, classified as *Exception Traceback Leakage*, are significantly different from those described above. They are detected by `CloudFlow` because traceback objects, which contain detailed information about raised exceptions, are included in HTTP requests, and hence visible to the user. While not directly exploitable, this pattern should be limited to the testing phase, as it can provide attackers with useful information.

5.3 Vulnerability Case Studies

To illustrate the effectiveness of the proposed approach, this section describes two of the aforementioned vulnerabilities (§5.2). Since this required manually analysing the corresponding data flows, we also explain how `CloudFlow` facilitates this process.

Code Injection via HTTP Case Study. The code in Listing 9 illustrates a case of code injection via HTTP. The application handler defined at line 1 is triggered by a HTTP request. Following this, the event object is parsed to extract a S3 bucket key and a set of options (line 2). The key is used to download a file from the bucket (line 3), whereas the options are validated by calling `param_validation` (line 5). The name of the downloaded file, i.e., `source_filename`, is checked to ensure that it identifies a valid image, and then passed to `image_transform` along with the validated options

[7] https://github.com/facebook/pyre-check/issues/899

Table 4: Summary of the security-sensitive data flows identified in the AWSomePy dataset (App.=Application).

Category	Total No.	App. No.	Inter-proc.	Intra-proc.	Injected Code	`CloudFlow` Models
Vulnerability	11	5	6	5	0	4
Potential Vulnerability	64	15	56	8	1	30
No Vulnerability	130	31	105	25	1	101

Table 5: Summary of the vulnerabilities detected in the AWSomePy dataset (App.=Application).

Vulnerability Type	Total No.	App. No.	Inter-proc.	Intra-proc.	Injected Code	`CloudFlow` Models
Code Injection via CLI	2	1	2	0	0	0
Code Injection via HTTP	5	3	4	1	0	4
Exception Traceback Leakage	4	1	0	4	0	0

(line 6). This function initializes the strings within the list `args`, which holds the command executed subsequently with `subprocess.check_output` (line 14).

Our analysis of the auxiliary functions `parse_event` and `is_valid_image` reveals that neither of them sanitizes the file name. Consequently, if the latter contains malicious commands, an attacker can trigger their execution with a HTTP request. `CloudFlow` detects this vulnerability because the handler `event` input is one of the processed sources. This shows the importance of automatically generating application-specific static analysis models (§3.5).

```
1  def handler(event, context):
2      s3_key, raw_ops = parse_event(event)
3      source_filename = download_s3_obj(BUCKET,
       s3_key)
4      if is_valid_image(source_filename):
5          ops = param_validation(raw_ops)
6          output_img = image_transform(
       source_filename, ops)
7
8  def image_transform(filename, ops):
9      args = ['convert', filename]
10     # Obtain filename extension
11     # Process ops and init output path
12     output = path + '.' + ext
13     args.append(output)
14     im_result = subprocess.check_output(args)
15     return output
```

Listing 9: Code injection via HTTP case study.

Exception Traceback Leakage Case Study. As shown in Table 5, one of the applications exhibits four intra-procedural vulnerabilities classified as exception traceback leakage. One of these is visible in Listing 10, which includes an event processing function of the application. The code shows that the function runs a set of processing and logging commands after the `try` statement (line 2). If an exception is raised during the execution of such commands, the traceback object `e` is returned unfiltered as part of the `Response` object (line 5). This will include detailed information on the exception, which can be leveraged by an attacker.

```
1  def event_process(missing_event) -> Response:
2      try:
3          # Processing and logging commands
4      except Exception as e:
5          return Response(str(e), status=500)
```

Listing 10: Exception traceback leakage case study.

Data Flow Manual Analysis. Our framework generates a summary report specifying the source and sink locations, along with the Pysa rule, for each of the identified data flows. The rules feature self-explanatory descriptions, e.g., *command-line arguments injection may result in RCE*, thus clarifying the reasons behind the detections. `CloudFlow` creates its report by processing the Pysa taint analysis results with the APIs provided by the SAPP tool [52]. To further inspect a given data flow, e.g., by tracking the taints of selected variables or generating a call graph, users can also leverage the Pysa debugging tools [45].

Although these features help identify vulnerabilities, they do not eliminate the need for manual analysis. In our case, it took on average approximately 20 minutes to inspect each data flow. However, we had no prior experience with the analysed applications. We expect the time required by an application's developer knowledgeable would be significantly lower.

5.4 Performance Evaluation

To evaluate the overhead caused by our framework, we report the execution times for the analysed AWSomePy applications. Our measurements are shown in Fig. 3, where both the total and the `CloudFlow`-specific execution times are displayed as a function of the number of lines of code (LOC) in the repository. Note that the presented values are obtained by averaging the results recorded during five `CloudFlow` executions. All our measurements were performed by running our framework

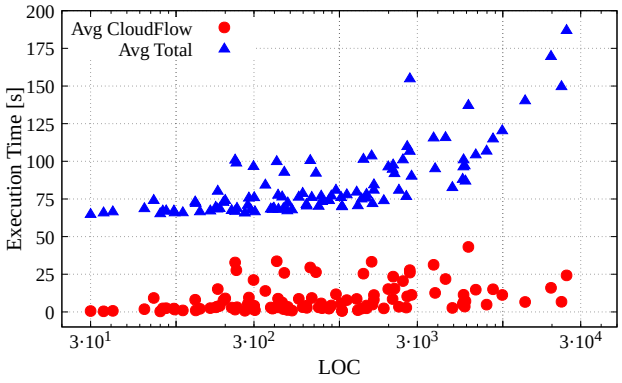

Figure 3: Total and `CloudFlow`-specific execution times vs LOC of the analysed AWSomePy repositories. Each value is the average of five `CloudFlow` executions. Note the logarithmic scale on the horizontal axis.

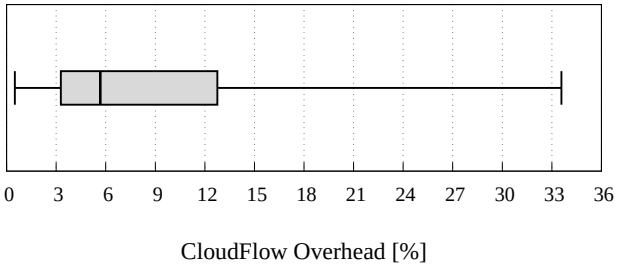

Figure 4: Interquartile range, minimum and maximum values of the recorded `CloudFlow` overhead percentages.

in a Linux Ubuntu 20.04 virtual machine (VM) with 8GB of RAM and a dual-core processor.

We observe that the execution times start increasing when the LOC reaches the approximate value of 3,000. Since the recorded `CloudFlow`-specific values do not exhibit the same trend, the highlighted execution time increase is caused by Pysa. The latter, in fact, performs an automated type inference step[8] and identifies the data flows, two computationally-intensive tasks when carried out on large repositories. Albeit expected, this result confirms the scalability of `CloudFlow`.

To better quantify the framework overhead, we also analyse the percentages of the `CloudFlow`-specific execution times w.r.t. the total. The box plot diagram of Fig. 4, which shows the interquartile range along with the minimum (0.5%) and maximum (33.6%) values, indicates that the `CloudFlow` overhead is generally acceptable. The median of the distribution, in fact, is 5.7%. Furthermore, we observe that for 75% of the analysed applications the overhead is 12.8% or less, with the latter being the value of the third quartile.

[8]Technically, the automated type inference is performed by Pyre, the static type checker on which Pysa is built [42]. Since the same interface, i.e., the `pyre` command, is used to execute both tools, our analysis aggregates the type inference time with the data flow identification time for simplicity.

6 Discussion

Analysis Framework Scope. As confirmed by the results of our real-world applications analysis (§5.2), `Cloudflow` is particularly suited to detecting code injection and information leakage vulnerabilities. Their identification relies on models, either automatically generated by `CloudFlow` or provided by Pysa, that classify relevant sources and sinks as security-sensitive. While this technique addresses several of the serverless risk factors described by the CSA [12], others require different approaches. Observability tools, for instance, are necessary to ensure that functions are adequately monitored and log files with security-relevant events are generated. Other risks identified by the CSA include information leakage caused by cross-execution data persistency and insecure third-party dependencies. We observe that the former depends upon the specifics of the serverless platform, and, as such, is challenging to detect statically in a reliable manner. In contrast, insecure dependencies can be identified with specialized static analysis tools [37]. Our framework therefore provides an effective way of complementing existing serverless security tools, especially those that rely on dynamic analysis.

Infrastructure Code Analysis. There are two main limitations to the infrastructure code analysis currently performed by `CloudFlow`. First, our framework identifies and processes only one YAML file for each application under test. However, 21 of the 104 (20.2%) inspected repositories include multiple infrastructure code files. Our analysis indicates that some of these applications comprise several services, which are orchestrated to provide a complex piece of functionality. Other applications, instead, rely on separate `serverless.yml` files to specify additional infrastructure components. This implies that the analysis performed by `CloudFlow` in these cases is inevitably partial. To address this, we plan to provide the user with the option of choosing the YAML files to be analysed.

`CloudFlow` extracts application permissions either from specific sections of the infrastructure code file, for instance identified by the tag `iamRoleStatements` (Listing 1), or according to the logic of the aforementioned IAM plugin [55]. These are the most widely adopted mechanisms to define application-wide and handler-level permissions, but others exist. To evaluate the accuracy of our real-world applications analysis, we quantify the occurrences of such alternative mechanisms in our dataset. As shown in Table 6, resource policies, which allow defining permissions for cloud resources in terms of trusted principals, are present in eight applications, and are the most frequently used of these mechanisms. Although not currently supported, resource policy-related information can also be processed by extending our framework. Similarly, the analysis of YAML-specified customer managed policies and AWS managed policies, maintained by the provider to meet the requirements of common use cases, can

be integrated as well.

In contrast, processing customer-managed policies and roles defined as account resources would require access to the used infrastructure. Although not feasible for a research study, this can be achieved in enterprise settings by deploying a suitably extended version of `CloudFlow`.

Table 6: Alternative mechanisms to define permissions identified in our dataset (arranged alphabetically).

Mechanism	Specification	Apps
AWS Managed Policies	Provider	7
Customer Managed Policies	Account	4
Customer Managed Policies	YAML	7
Resource Policies	YAML	8
Roles	Account	4

Application Code Analysis. Our framework currently supports nine cloud APIs in total, which are shown in Table 7 along with the corresponding four AWS services. In four cases, i.e., `put_item`, `put_object`, `send_message`, and `upload_file`, this is achieved with cloud API call models, whereas we implement custom Pysa models in the other cases. We focus our attention on this set of APIs because they are among the most frequently used in the AWSomePy dataset. Its characterization shows that 46 different services are used, and each of them is accessed via multiple API calls. This implies that the results obtained with our real-world applications leave out other less popular cloud services. As an example, one of the applications affected by code injection also contains an SQL injection vulnerability that goes undetected due to the absence of models for the AWS RDS service.

`CloudFlow` can be expanded to increase the number of supported services and APIs in a straightforward manner. However, some applications complicate this task by using configuration and management-oriented services to create cloud resources programatically [46]. For these applications, some important pieces of information become available only at runtime, which implies that the application needs to be analysed by combining dynamic and static techniques. Static analysis can still provide valuable information even when such services are used, but since they introduce data flows that are hard to detect statically we leave to future work the implementation of suitable hybrid strategies.

API Documentation Analysis. At present, `CloudFlow` relies on API models obtained from manual analysis of API documentation. The purpose of the analysis is to identify security-relevant inputs, such as the names of cloud databases. Although automated analysis via NLP methods is possible [62], several recent works, e.g., on AWS environments [58] and Android security [49, 63], still employ manual analysis of a limited set of APIs. This also facilitates validation of the resulting models.

Table 7: Cloud APIs supported by `CloudFlow` and corresponding AWS services.

API	S3	DynamoDB	SQS	SNS
`delete`	✔			
`list_objects`	✔			
`list_objects_v2`	✔			
`publish`				✔
`put_item`		✔		
`put_object`	✔			
`scan`		✔		
`send_message`			✔	
`upload_file`	✔			

We note that `CloudFlow` requires only a simple API documentation analysis. Consequently, it is possible to automate this process by integrating an API documentation processing framework, e.g., DocFlow [62], into our analysis pipeline. We leave this task to future work.

Generalizability. `CloudFlow` currently identifies security-sensitive data flows in serverless applications implemented in Python. This is, in fact, the only language supported by Pysa, the static analysis tool on which our current implementation relies. However, we emphasize that the development of the proposed methodology was guided by challenges that are language-independent. As a result, `CloudFlow`'s architecture is generic, and can be applied to any programming language.

Specifically, four steps are required to achieve this. The first step (i) is to identify a toolchain that allows parsing the AST and synthesizing code from it. A language-specific static analysis tool with data flow detection capabilities (ii) is also necessary to replace Pysa (e.g., SootUp [25] for Java applications). The third step (iii) is to configure SDK-specific type annotations, if supported by the target language. In our implementation, these are `boto3`-specific. Finally, step (iv) consists of creating documentation-based API models, which depend on the SDK. These can be automatically generated with an API documentation processing framework.

Furthermore, we highlight that `CloudFlow`'s architecture can also be applied to any cloud provider. This requires a preliminary analysis to ascertain how the application code interacts with the cloud services. In our experience, this is routinely implemented via an SDK for simplicity. Once this is confirmed and the target language is chosen, our architecture can be applied by following the four steps described above.

A possible avenue for future work is a JavaScript-focused extension of `CloudFlow`. According to a characterization of the Wonderless dataset [17], JavaScript is the most popular language for the implementation of open-source serverless applications. However, a recent evaluation by Brito *et al.* of nine mainstream tools indicates that static

detection of security vulnerabilities in JavaScript code remains problematic [11]. In particular, existing tools struggle with the highly dynamic constructs of the language and the complexity of the dependency system. Consequently, although porting `CloudFlow` to JavaScript is feasible and worth pursuing given the popularity of JavaScript in the serverless domain, whether performance comparable to our Pysa-based `CloudFlow` can be achieved is an open question.

Asynchronous Constructs. Modern programming languages feature a range of asynchronous programming constructs to support non-blocking I/O operations (e.g., promise objects, asynchronous functions, and asynchronous iterators). These constructs are challenging to analyse statically, as they require specialized frameworks [61]. The objective of `CloudFlow` is not to support asynchronous constructs but to synthesize and inject infrastructure-dependent handler calls. We then use Pysa to inspect the instrumented application. As with any other general-purpose tool, Pysa relies on over-approximations, e.g., taint broadening [44], for the analysis of asynchronous constructs.

Nonetheless, the validity of the results presented in §5.2 is not significantly affected. Our analysis shows that only two of the 104 analysed AWSomePy applications make use of asynchronous Python constructs, i.e., `async`/`await` keywords and the `asyncio` module. Specifically, we find six occurrences of such constructs spread across five application files, with the overall number of Python files being 3,612. We believe that asynchronous constructs are rare in our dataset because their functionality is provided by the cloud services. We have therefore taken a pragmatic approach and consider only patterns more frequently adopted in serverless applications.

In our future work, we intend to improve the precision of our framework by integrating specialized static analysis schemes, such as the JavaScript-specific one proposed by Sotiropoulos *et al.* [61]. We are not aware, however, of alternative schemes for Python, as recent research focused on this language does not take into account asynchronous constructs [32, 50].

7 Related Work

Serverless Security. Several systems target the infrastructure on which serverless applications rely. Alzayat *et al.* [4] focus their attention on the security issues caused by container reuse, very frequently adopted by FaaS providers to minimize cold-start delays. The devised system, Groundhog, prevents information leakage by restoring a private data-free memory snapshot before the execution of a serverless function. Groundhog has a negligible effect on latency and throughput. Firecracker [2] has similar objectives, but improves the standard container-based strategy with micro virtual machines including a reduced set of resources. In contrast, we focus on the analysis of serverless applications at coding time.

Research has also been conducted on alternative serverless execution environments that deploy dedicated code on an existing platform. Valve [14] monitors policy-based information flows by means of taint labels. Similarly, WILL.IAM [51] relies on permissions graphs to implement a serverless-specific access control model. Unlike Trapeze [3], which implements a language-based approach, neither Valve nor WILL.IAM require modifying the application code. More recently, Kalium [23] enforces control-flow integrity by monitoring the order of execution of serverless functions. None of these frameworks analyse serverless applications statically, the key objective of `CloudFlow`.

Static Analysis of Serverless Applications. Raffa *et al.* [47] propose a prototype-level static analysis framework that relies on manually-injected handler calls and type annotations. `CloudFlow` builds on this work by parsing and modifying the AST of the inspected application. Our framework fully automates the analysis process, builds a specific representation to facilitate the detection of relevant data flows, and supports the analysis of environment variables, event filters, and deployment tool plugins.

Obetz *et al.* [35] present a novel approach to extending the concept of call graph to serverless environments. To support information flow and security analysis, along with automatic generation of documentation, the work introduces the idea of extended service call graph to represent the structure of an application. The authors, who also propose an extension of their graph construction algorithm [34], make use of functions' summaries to consider the effects of platform services and external libraries. In contrast, we rely on event object models, which are defined once for each supported service and automatically populated by processing the infrastructure and the application code. Furthermore, our main objective is the detection of security-sensitive data flows, not the generation of a comprehensive call graph.

Wang *et al.* [65] develop a performance model for serverless that uses data flow and code profiling information from Pysa and code instrumentation. Unlike our work, the authors do not process the infrastructure code, and focus on the application's response time rather than its security.

Formal Semantics for Serverless. Jangda *et al.* [22] propose operational semantics consisting of six rules focused on the low-level behaviour of serverless platforms. Such rules, however, do not provide the event-level details necessary for our work. In contrast, Gabbrielli *et al.* [19] present a serverless-specific formal model based on a Serverless Kernel Calculus that includes both stateless functions and stateful platform services. Obetz *et al.* [34] further improve existing in-process and event semantics in order to capture service-specific characteristics, along with function-to-function and function-to-service data transmission. The latter work provides part of the theoretical foundations of our research,

which translates those notions into a security-focused static analysis tool.

Testing of Serverless Applications. Winzinger *et al.* [66] illustrate how dependency graphs, which combine information extracted from application and infrastructure code, can be used to analyse serverless applications. In addition to identifying workflows, the authors show that this model-based approach enables the developer to evaluate the test coverage of an application [67]. Winzinger *et al.* also propose a serverless-specific testing framework focused on data flows [68]. First, these are modeled by categorizing the values returned by the functions in their dataset. Second, the authors rely on their model to devise a code instrumentation strategy aiming at logging data flow-related information. The work has recently been expanded with a methodology and a prototype tool to automatically generate test cases for serverless applications [69]. However, unlike the approach presented in this paper, all these works rely on models that cannot be fully generated at coding time, as they require runtime information along with static attributes. As a result, data flows can only be analysed dynamically. We also note that the techniques proposed are used for coverage and integration testing and are not designed to detect security-sensitive data flows.

Serverless Benchmarks. The measurement study by Wang *et al.* [64] aims at comparing architecture and resource management strategies in three major serverless platforms. While identifying security risks and vulnerabilities, the work relies on measurement functions rather than security-oriented benchmarks. Back *et al.* [8] design a microbenchmark specifically to evaluate the cost model adopted by four serverless providers. The microbenchmark is, therefore, designed to study exclusively the effect of number of function invocations and memory usage. In contrast, FaaSdom [30] compares serverless platforms in terms of cost and system performance. This benchmark suite, in fact, incorporates metrics focused on latency and throughput. However, while supporting a range of workloads and multiple languages, FaaSdom does not include security-focused tests.

Static Analysis of Interpreted Languages. Interpreted languages have played a key role in the development of cloud and web applications. Therefore, numerous research contributions on static detection of security vulnerabilities in such languages exist. As mentioned above, many of these are JavaScript-specific [11, 61]. For PHP, Xie *et al.* [70] present a novel algorithm that relies on function summaries and symbolic execution to conduct inter-procedural analyses. Pixy [24] increases the precision of data flow analysis by inspecting literals and aliases. Hauzar *et al.* [21] propose an abstract interpretation framework for value and heap analysis. Son *et al.* [60], combine taint analysis with symbolic execution to detect denial of service vulnerabilities and missing authorization checks.

These PHP frameworks are different from `CloudFlow` for two reasons. First, they focus on the dynamic features of PHP. Second, they support security mechanisms used in PHP-based web applications. In contrast, we propose a language-independent approach focused on the characteristics of the serverless paradigm. Given their features, such frameworks could replace Pysa if `CloudFlow` were extended to support PHP. However, according to the Wonderless dataset statistics, this language is not popular in the serverless domain [17].

8 Conclusion

We present `CloudFlow`, a novel approach to statically identifying security-sensitive data flows in serverless applications. `CloudFlow` uses information extracted from infrastructure and application code to construct a synchronous representation of event-driven serverless applications. `CloudFlow` then generates application and cloud API specific static analysis models. Together with the synchronous representation, these can be provided to a general-purpose static analysis tool for data flow detection.

To evaluate `CloudFlow`, we also release `CloudBench`, a new suite of security-oriented microbenchmarks. `CloudFlow` passes 92.5% of the `CloudBench` microbenchmarks, i.e., 37 out of 40, confirming its effectiveness. Furthermore, we use `CloudFlow` to identify security-sensitive data flows in 104 real-world applications obtained from a recent dataset. To the best of our knowledge, this is the largest security-focused analysis of serverless applications to date. `CloudFlow` detects a total of 11 vulnerabilities in the real-world applications we analysed. Seven of these are code injection vulnerabilities. The remaining four are information disclosure vulnerabilities due to exception traceback leaks.

Overall, `CloudFlow` shows that vulnerabilities exist in real-world serverless applications and that it is possible to discover them using static analysis. More broadly, `CloudFlow` demonstrates the value of specialised analysis techniques for this emerging cloud computing paradigm. We hope that `CloudFlow` and `CloudBench` can foster further research into serverless security frameworks that prevent the deployment of insecure applications on public clouds, avoiding the consequences that this may have for organizations and users alike.

Acknowledgments

This research was partially funded by the Spanish Ministry of Science and Innovation, through the State Research Agency (AEI), grant number [PID2023-151996OB-I00]. O'Keeffe was supported by the EPSRC grant EP/Y036417/1 (REDONDA project). Dash was supported by the EPSRC grant EP/W015927/2.

Ethics Considerations

To comply with the GitHub process for coordinated vulnerability disclosure, we contacted the maintainers of two vulnerable applications in November 2024, but we did not receive a reply prior to the submission of this paper. As regards the other three, the validation of our results shows that one of those affected by code injection via HTTP is intentionally vulnerable, and was open-sourced by its authors to illustrate common serverless security issues. In the remaining two cases, we were unable to notify the authors of our findings, as the repositories do not include a security policy and, as they are now archived, raising issues is no longer possible.

Open Science

To comply with the USENIX open science policy, we make publicly available two artifacts on Zenodo[9], along with all the documentation and additional resources required to reproduce our results.

The first is a virtual machine (VM) with the source code for both `CloudFlow` (§3) and `CloudBench` (§4.1), and a preprocessed version of the AWSomePy dataset. We obtained the latter after filtering out files not necessary for our analysis. In addition, our VM includes the installation of our command-line utility (§3.1) as well as Pysa, the underlying static analysis tool used by our framework.

The second artifact is the data flow report generated by `CloudFlow` for our real-world applications analysis (§5). This report contains detailed information about the identified security-sensitive data flows, such as the AWSomePy repository and the lines of code where sources and sinks are located.

References

[1] Dynamodb injection, June 2018. [online] https://medium.com/appsecengineer/dynamodb-injection-1db99c2454ac.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[3] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure serverless computing using dynamic information flow control. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[4] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in faas. EuroSys '23, page 398–415, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Aws solutions, June 2021. [online] https://aws.amazon.com/.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Aws lambda documentation, June 2021. [online] https://docs.aws.amazon.com/lambda/index.html.

[8] Timon Back and Vasilios Andrikopoulos. Using a microbenchmark to compare function as a service solutions. In Kyriakos Kritikos, Pierluigi Plebani, and Flavio de Paoli, editors, *Service-Oriented and Cloud Computing*, pages 146–160, Cham, 2018. Springer International Publishing.

[9] Yaml: The missing battery in python, December 2024. [online] https://realpython.com/python-yaml/.

[10] AWS SDK for Python (Boto3) to create, configure, and manage AWS services, March 2023. [online] https://boto3.amazonaws.com/v1/documentation/api/latest/index.html.

[11] Tiago Brito, Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, José Fragoso Santos, and Nuno Santos. Study of javascript static analysis tools for vulnerability detection in node.js packages. *IEEE Transactions on Reliability*, 72(4):1324–1339, 2023.

[12] The 12 most critical risks for serverless applications, November 2019. [online] https://cloudsecurityalliance.org/blog/2019/02/11/critical-risks-serverless-applications/.

[13] Andrea Continella, Mario Polino, Marcello Pogliani, and Stefano Zanero. There's a hole in that bucket! a large-scale analysis of misconfigured s3 buckets. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 702–711, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. Valve: Securing function workflows on serverless computing platforms.

[9] https://doi.org/10.5281/zenodo.15609299

In *Proceedings of The Web Conference 2020*, WWW '20, pages 939–950, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Pubali Datta, Isaac Polinsky, Muhammad Adil Inam, Adam Bates, and William Enck. ALASTOR: Reconstructing the provenance of serverless intrusions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2443–2460, Boston, MA, August 2022. USENIX Association.

[16] James Davis, Arun Thekumparampil, and Dongyoon Lee. Node.fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 145–160, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Nafise Eskandani and Guido Salvaneschi. The wonderless dataset for serverless computing. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 565–569, 2021.

[18] Xinwei Fu, Dongyoon Lee, and Changhee Jung. nadroid: statically detecting ordering violations in android applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO '18, page 62–74, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, Fabrizio Montesi, Marco Peressotti, and Stefano Pio Zingaro. No more, no less. In Hanne Riis Nielson and Emilio Tuosto, editors, *Coordination Models and Languages*, pages 148–157, Cham, 2019. Springer International Publishing.

[20] Accelerate your transformation with google cloud, June 2021. [online] https://cloud.google.com/.

[21] David Hauzar and Jan Kofron. Framework for Static Analysis of PHP Applications. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 689–711, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[22] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[23] Deepak Sirone Jegan, Liang Wang, Siddhant Bhagat, and Michael Swift. Guarding serverless applications with kalium. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4087–4104, Anaheim, CA, August 2023. USENIX Association.

[24] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6 pp.–263, 2006.

[25] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. Sootup: A redesign of the soot static analysis framework. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–247, Cham, 2024. Springer Nature Switzerland.

[26] Jason Katzer. *Learning Serverless*. O'Reilly Media, Inc., Sebastopol, CA, USA, first edition, 2020.

[27] Xing Li, Xue Leng, and Yan Chen. Securing serverless computing: Challenges, solutions, and opportunities. *IEEE Network*, 37(2):166–173, 2023.

[28] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node.js javascript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 505–519, New York, NY, USA, 2015. Association for Computing Machinery.

[29] Bojan Magusic. *Azure Security*. Manning Publications, Shelter Island, NY, USA, first edition, 2023.

[30] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. Faasdom: a benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS '20, page 73–84, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Power your vision on azure, June 2021. [online] https://azure.microsoft.com/en-gb/.

[32] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static Type Analysis by Abstract Interpretation of Python Programs. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:29, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[33] Jussi Nupponen and Davide Taibi. Serverless: What it is, what to do and what not to do. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 49–50, 2020.

[34] Matthew Obetz, Anirban Das, Timothy Castiglia, Stacy Patterson, and Ana Milanova. Formalizing event-driven behavior of serverless applications. In Antonio Brogi, Wolf Zimmermann, and Kyriakos Kritikos,

editors, *Service-Oriented and Cloud Computing*, pages 19–29, Cham, 2020. Springer International Publishing.

[35] Matthew Obetz, Stacy Patterson, and Ana Milanova. Static call graph construction in AWS lambda serverless applications. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[36] Owasp serverless top 10, June 2021. [online] https://owasp.org/www-project-serverless-top-10/.

[37] Guy Podjarny and Liran Tal. *Serverless Security*. O'Reilly Media, Inc., Sebastopol, CA, USA, first edition, 2019.

[38] Major jobs website left sensitive client data exposed for months, June 2021. [online] https://portswigger.net/daily-swig/major-jobs-website-left-sensitive-client-data-exposed-for-months.

[39] Insecure amazon s3 bucket exposed personal data on 500,000 ghanaian graduates, January 2022. [online] https://portswigger.net/daily-swig/insecure-amazon-s3-bucket-exposed-personal-data-on-500-000-ghanaian-graduates.

[40] Turkish flight operator pegasus airlines suffers data breach, June 2022. [online] https://portswigger.net/daily-swig/turkish-flight-operator-pegasus-airlines-suffers-data-breach.

[41] Getting started with pyre, October 2024. [online] https://pyre-check.org/docs/getting-started/.

[42] Coverage increasing strategies (pyre infer), March 2025. [online] https://pyre-check.org/docs/pysa-coverage/#pyre-infer.

[43] Pysa overview, November 2023. [online] https://pyre-check.org/docs/pysa-basics/.

[44] Taint broadening, November 2023. [online] https://pyre-check.org/docs/pysa-advanced/#taint-broadening.

[45] Debugging tools, November 2023. [online] https://pyre-check.org/docs/pysa-tips/#debugging-tools.

[46] Giuseppe Raffa, Jorge Blasco Alis, Dan O'Keeffe, and Santanu Kumar Dash. Awsomepy: A dataset and characterization of serverless applications. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*, SESAME '23, page 50–56, New York, NY, USA, 2023. Association for Computing Machinery.

[47] Giuseppe Raffa, Jorge Blasco, Dan O'Keeffe, and Santanu Kumar Dash. Towards inter-service data flow analysis of serverless applications. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 654–658, 2024.

[48] Python, boto3, and aws s3: Demystified, October 2018. [online] https://realpython.com/python-boto3-aws-s3/.

[49] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS 2014*, volume 14, 2014.

[50] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1646–1657, 2021.

[51] Arnav Sankaran, Pubali Datta, and Adam Bates. Workflow integration alleviates identity and access management in serverless computing. In *Annual Computer Security Applications Conference*, ACSAC '20, pages 496–509, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Post processor for facebook static analysis tools, May 2025. [online] https://github.com/facebook/sapp.

[53] Community signup serverless service with lambda, python, step function and dynamodb, October 2024. [online] https://www.serverless.com/examples/slack-signup-serverless.

[54] See real world serverless code and architecture examples, August 2021. [online] https://www.serverless.com/examples/.

[55] Serverless iam roles per function plugin, October 2024. [online] https://www.serverless.com/plugins/serverless-iam-roles-per-function.

[56] Serverless step functions plugin, October 2024. [online] https://www.serverless.com/plugins/serverless-step-functions.

[57] Zero-friction serverless development, August 2021. [online] https://www.serverless.com/.

[58] Ilia Shevrin and Oded Margalit. Detecting Multi-Step IAM attacks in AWS environments via model checking. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6025–6042, Anaheim, CA, August 2023. USENIX Association.

[59] Dylan Shields. *AWS Security*. Manning Publications, Shelter Island, NY, USA, first edition, 2022.

[60] Sooel Son and Vitaly Shmatikov. Saferphp: finding semantic vulnerabilities in php applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11, New York, NY, USA, 2011. Association for Computing Machinery.

[61] Thodoris Sotiropoulos and Benjamin Livshits. Static Analysis for Asynchronous JavaScript Programs. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:29, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[62] Marcos Tileria, Jorge Blasco, and Santanu Kumar Dash. Docflow: Extracting taint specifications from software documentation. ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[63] Marcos Tileria, Jorge Blasco, and Guillermo Suarez-Tangil. WearFlow: Expanding information flow analysis to companion apps in wear OS. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 63–75, San Sebastian, October 2020. USENIX Association.

[64] 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, Boston, MA, July 2018. USENIX Association.

[65] Runan Wang, Giuliano Casale, and Antonio Filieri. Enhancing performance modeling of serverless functions via static analysis. In Javier Troya, Brahim Medjahed, Mario Piattini, Lina Yao, Pablo Fernández, and Antonio Ruiz-Cortés, editors, *Service-Oriented Computing*, pages 71–88, Cham, 2022. Springer Nature Switzerland.

[66] Stefan Winzinger and Guido Wirtz. Model-based analysis of serverless applications. In *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*, pages 82–88, 2019.

[67] Stefan Winzinger and Guido Wirtz. Applicability of coverage criteria for serverless applications. In *2020 IEEE International Conference on Service Oriented Systems Engineering (SOSE)*, pages 49–56, 2020.

[68] Stefan Winzinger. and Guido Wirtz. Data flow testing of serverless functions. In *Proceedings of the 11th International Conference on Cloud Computing and Services Science - CLOSER*, pages 56–64. INSTICC, SciTePress, 2021.

[69] Stefan Winzinger and Guido Wirtz. Automatic test case generation for serverless applications. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 77–84, 2022.

[70] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, USA, 2006. USENIX Association.