AWSomePy: A Dataset and Characterization of Serverless Applications

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

Jorge Blasco Alís
Universidad Politécnica de Madrid
jorge.blasco.alis@upm.es

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

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

ABSTRACT

Over the last few years, the serverless computing paradigm has become increasingly popular. Thanks to its cost-effectiveness and the possibility of relying on a cloud provider to manage the underlying infrastructure, companies making use of serverless platforms can fully focus on developing the business logic of their products. However, adopting the serverless model also implies facing several performance, traceability and security-related challenges. Some of them can be tackled by analysing real-world applications and identifying key trends, as these can guide the development of novel models and tools. In spite of this, little up-to-date information on such trends is currently available in the literature.

In this work, we gather and interpret information that can be leveraged to analyse serverless applications. To achieve our goal, we study a set of applications developed in Python for the Amazon Web Services (AWS) platform. We first conduct an architectural analysis to identify serverless-specific parameters, e.g., plugins used in deployment tools and the number of events and handlers. We then perform an application code-level analysis in order to establish which cloud services and APIs developers use most frequently. Our results show that granular definition of handler permissions is not common practice. Furthermore, developers make use of configuration services and programmatic creation of cloud resources, thus adding workflows difficult to analyse statically. Our dataset, AWSomePy, is publicly available to support future research work.

1 INTRODUCTION

The serverless computing paradigm aims at reducing the overhead associated with deployment and monitoring of traditional servers by leveraging a stateless and event-driven model, which relies on platform services offered by a cloud provider [5]. By abstracting away almost all operational concerns, including infrastructure scalability and IT hardware maintenance, enterprises can cut their costs and focus on developing their software products. However, relying on serverless environments, such as AWS [23], Azure [4] and GCP [17], presents new challenges, especially in the areas of performance [25], traceability [9] and security [2, 7, 15, 18].

Both static and dynamic analysis of serverless applications are problematic. Unlike traditional applications, serverless applications routinely receive their inputs from a variety of sources and code execution can be triggered by different kinds of events, such as a database update or a file upload [5]. Recent academic studies have therefore introduced new frameworks for information flow analysis [3, 8, 22, 24] and tracing of potentially malicious events [9]. While demonstrating the correctness in principle of the proposed approaches, the experimental evaluations of such frameworks rely on a very limited set of applications. Thus, they are not optimized to consider architectures and cloud services most frequently used in real-world applications.

Static analysis, in particular, is very challenging in the context of serverless computing. In addition to the high number of events, analysing the code that implements platform services is not possible. Obetz *et al.* [16], who extended the concept of call graphs to serverless applications, show that static analysis is undoubtedly useful, but inevitably has to rely on models and approximations. Similarly to the case of information flow and traceability frameworks discussed above, such models and approximations should ideally be based on key trends and features extracted from a large collection of applications. Unfortunately, existing general-purpose datasets, such as PyTraceBugs [1] and BugsInPy [29], do not take into account the specific characteristics of the serverless paradigm, as they were primarily conceived to support static source code analysis and unit testing for traditional Python applications.

In this work, to guide the development of models and tools for serverless computing, we analyse a dataset of 145 applications obtained from GitHub by adopting and customizing the Wonderless dataset methodology [14]. Given the growing popularity of Python in the serverless domain, we focus our attention on applications implemented in Python for AWS, which is the most widely used development platform [27]. Our results show that permissions are rarely configured on a per-handler basis, despite this being considered a good practice to secure serverless applications. Furthermore, while data storage and NoSQL services are, as expected, the most frequently used, developers rely on configuration and management-oriented services as well. These provide high flexibility, but they also trigger workflows that are difficult to inspect prior to deployment.

In summary, we make the following contributions:

- We publicly release AWSomePy[1], a new dataset of AWS serverless applications implemented in Python and compatible with the Serverless Framework deployment tool.
- We provide a characterization of our dataset, which we obtain by conducting an architectural and an application code-level analysis.

[1] https://doi.org/10.5281/zenodo.7838076

Table 1: Summary of the dataset generation process.

Step	YAML Files	Repositories	Dataset Size
1	9,096	✗	✗
2	7,912	✗	✗
3	✗	7,074	✗
4	✗	811	8.7 GB
5	✗	783	8.5 GB
6	✗	159	1.6 GB
7	✗	147	1.6 GB
8	✗	147	1.6 GB
9	✗	145	1.6 GB

2 DATASET GENERATION

This section describes how the dataset used for our analysis (§ 3) was generated. Starting from the consolidated methodology of the Wonderless dataset [14], which collected GitHub applications compatible with the Serverless Framework deployment tool [28], we generate a new, Python-only version of the dataset in August 2022 by customizing the code provided by its authors [13].

While including a large collection of mature and well-maintained applications, the Wonderless methodology suffers from the limitation that it does not collect repository-specific or application-level metadata. This information can help researchers to identify suitable benchmarks and to assess the complexity of applications in the dataset, as we show in this study (§ 3.1). For this reason, our methodology includes a further step that gathers a set of metadata, which include, among others, the number of stars, events and handlers. Moreover, Wonderless was created in July 2020. Considering the constant evolution of serverless offerings and the latest features of the widely adopted Serverless Framework[2], which facilitates the deployment of serverless applications through declarative infrastructure code, we decided to base our investigation on the most up-to-date dataset possible.

The remainder of this section is dedicated to detailing the nine steps of the dataset generation process, which is outlined in Fig. 1.

Identification of configuration files. The processing pipeline developed by Eskandani *et al.* [14] starts by querying GitHub to identify all the repositories containing at least one `serverless.yml` file, which is used to configure applications to be deployed with the selected framework. As shown in Table 1, which summarizes the output of each processing step in terms of identified files or repositories and dataset size, 9,096 configuration files are detected in step ❶. These are filtered in step ❷ to remove files developed by the Serverless Framework community as well as those included in folders, e.g., *demo* and *test*, indicating that the application is either a template or a toy example. The final number of identified configuration files is 7,912. We emphasize that no repository was cloned during the execution of the first two processing steps.

Identification of repositories. Even though it is reasonable to assume that a vast majority of serverless applications are deployed with only one YAML file, the framework allows for the usage of multiple configuration files. This implies that it is not possible to uniquely identify the repositories URLs by exclusively relying on the gathered `serverless.yml` files. The goal of step ❸ is therefore to obtain such URLs by removing duplicate entries, which enables us to shortlist 7,074 repositories. However, since the focus of this research is the analysis of applications implemented in Python, in step ❹ we filter the repositories by primary language prior to cloning. This was achieved by using a GitHub API that returns several pieces of information about a target repository [12]. The outcome of the latter step consists of 811 repositories with a total dataset size of 8.7 GB.

The removal of invalid YAML files is the objective of step ❺. As observed by Eskandani *et al.*, these include *syntactically* and *semantically* invalid files, with the latter category comprising files that do not specify either the cloud provider or the application functions. This round of filtering causes only a modest reduction in the number of repositories and in the size of the dataset, which decrease to 783 and 8.5 GB, respectively. Congruently with the Wonderless dataset, such quantities are much more significantly reduced in step ❻, which aims at filtering out immature projects, i.e., active for less than one year, and identifies 159 repositories with a dataset size of 1.6 GB.

In order to remove non-real-world applications, the inherited pipeline includes an additional round of filtering, namely step ❼, which is focused on the analysis of repository metadata, such as labels, topics and descriptions. They are compared with a set of keywords, e.g., *demo*, *test* and *example*. In our case, this processing step filters out only 12 repositories, thus bringing the total to 147 without any substantial variation of the dataset size. As for step ❽, Eskandani *et al.* explain that they analysed a list of potential forks, i.e., repositories that feature the same name, but different developers, to identify those to be filtered out. The rationale behind this is that, in all probability, an application developed by forking another shares large portions of the original source code. It is noteworthy that the Wonderless dataset code only generates a list of *candidate* forks, which have to be manually analysed. However, in our case, no such candidates are shortlisted, which implies that the total number of repositories remains unchanged.

Metadata gathering. To support the characterization of the obtained applications, in step ❾ we further customize the dataset generation process with code that extracts repository-specific metadata, such as number of stars, watchers and forks, and processes both the YAML files and the source code to gather information about cloud provider, Serverless Framework version, lines of code, number of events and number of handlers. This analysis step enables us to verify that only 2 out of 147 are non-AWS applications, which we remove. As a result, 145 repositories are included in AWSomePy, which we release along with the associated metadata.

3 DATASET ANALYSIS

In this section, we detail the methodology of our analysis and present the obtained results. We first consider configuration and

[2]According to Datadog, in 2021 the Serverless Framework was adopted by 90% of the surveyed organizations using AWS [26].

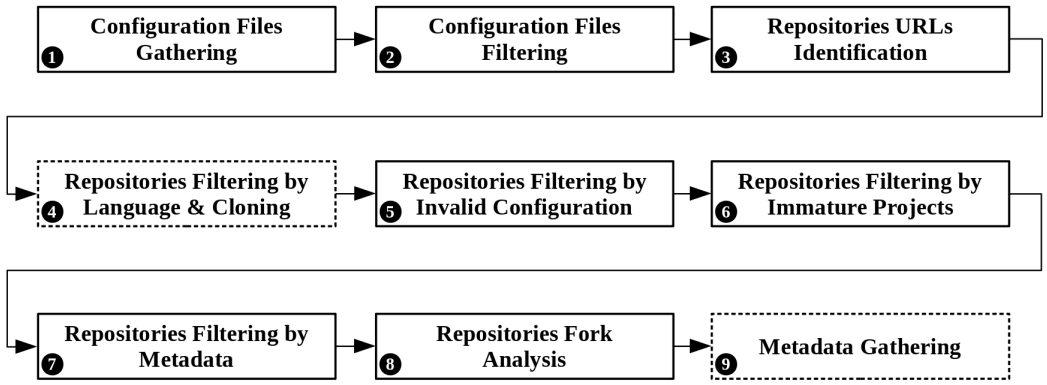

Figure 1: Dataset generation process. The dashed lines indicate the customized steps compared to the Wonderless dataset.

Table 2: Top eight plugins in AWSomePy.

Plugins	Occurrences
`serverless-python-requirements`	95
`serverless-pseudo-parameters`	25
`serverless-domain-manager`	15
`serverless-step-functions`	14
`serverless-offline`	9
`serverless-dotenv-plugin`	8
`serverless-prune-plugin`	8
`serverless-iam-roles-per-function`	7

application architectural parameters (§ 3.1), such as deployment tool plugins used and the number of handlers and events. Second, we analyse the application code (§ 3.2) to identify the most common cloud platform services and APIs.

3.1 Configuration & Architectural Analysis

Plugin analysis. The Serverless Framework supports a large number of plugins that facilitate the integration of complex features. To gain insight into the plugins used when deploying AWSomePy applications, we parse `serverless.yml` files and extract the relevant information from a dedicated tag. For simplicity, when a repository contains more than one YAML configuration file, we consider only one of them, as it is plausible that all have similar features and complexity. The processed file is always the first identified by recursively visiting the repository with the Python standard library function `os.walk`.

A total of 44 plugins were identified, and the top eight most frequently occurring are shown in Table 2. The most frequent plugin by far is `serverless-python-requirements`, which is designed to assist with dependency management. The next plugin, i.e., `serverless-pseudo-parameters`, is also configuration-oriented, as it supports using AWS CloudFormation syntax to specify configuration parameters. Interestingly though, the plugin in question is now deprecated [20], as its functionality is natively supported by the most recent releases of the Serverless Framework. This result is consistent with the approach adopted to generate the dataset (§ 2), which explicitly prioritises mature applications.

Unlike the first and the second, the third and the fourth most frequently used plugins, i.e., `serverless-domain-manager` and `serverless-step-functions` are functionality-oriented. The former enables creating custom domain names by leveraging specialized AWS services, whereas the latter facilitates the deployment of step functions.

As for the remaining four plugins, `serverless-offline`, which supports local testing of serverless applications, is present in only 9 YAML files. Finally, with 7 occurrences, the least frequently detected plugin in Table 2 is `serverless-iam-roles-per-function`. Crucially, this implies that developers are not adopting the best practice of configuring permissions in a granular fashion. Without this plugin, which defines *per-function* IAM roles, every function in the application is deployed with the same *global* IAM role, thus increasing the chances of it being over-privileged.

Complexity analysis. In order to understand architectural characteristics of the applications in our dataset, we assess their complexity by considering the number of lines of code (LOC)[3], events and handlers. The average LOC is 4,468, while the minimum and the maximum are 26 and 132,658, respectively. Furthermore, the cumulative distribution of the LOC in Fig. 2 indicates that 55% of the AWSomePy repositories have less than 1 kLOC, though those with under 100 LOC constitute less than 10% of the dataset.

An assessment based on LOC alone, despite its importance, does not capture key complexity indicators of serverless applications. For this reason, we process each `serverless.yml` file to obtain information about the application's events and handlers. Even though both types of information can be specified in a YAML file in multiple ways, to facilitate the implementation of an automated parser, we limit our analysis to the tag `functions`. Considering the plugins present in AWSomePy (Table 2) and the application code analysis detailed later in § 3.2, we believe this does not significantly affect the accuracy of our results. As elucidated later in § 3.2, the

[3]We compute the LOC with the command-line tool Pygount [19].

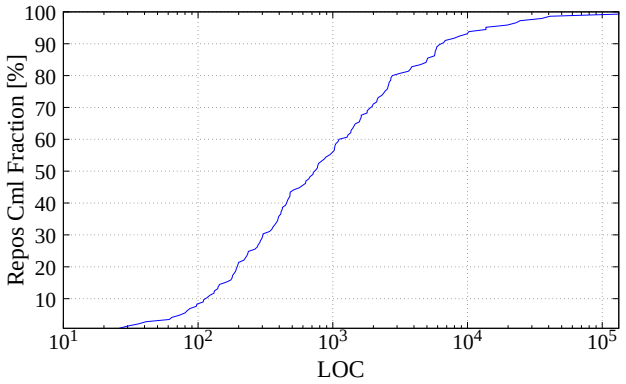

Figure 2: Cumulative distribution of the lines of code in AWSomePy (cumulative fraction of repositories on the y axis).

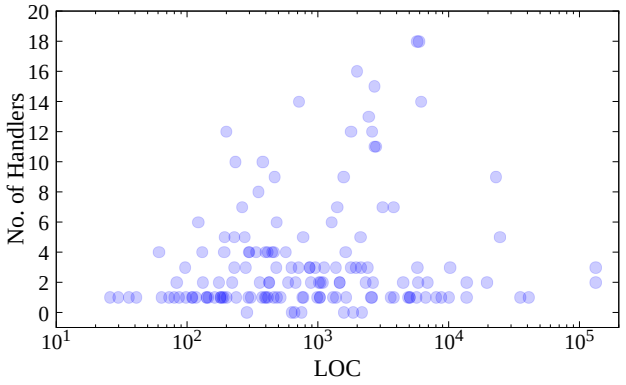

Figure 3: Number of handlers vs lines of code in AWSomePy.

AWS services, e.g., `stepfunctions`, that would require specifying events and handlers in other YAML tags are not widely adopted in AWSomePy.

The correlation between number of handlers and LOC is shown in Fig. 3, where the darkest areas indicate a higher point density. The figure shows that the AWSomePy applications typically include at most four handlers. We note that as a consequence of the aforementioned limitation of our parser a few points in Fig. 3 have no handlers.

Finally, the number of events versus number of handlers diagram in Fig. 4 reveals that the majority of the AWSomePy applications comprise less than five events. However, similarly to Fig. 3, due to our simplified parsing approach, some applications are classified as having no events, which implies that their architecture should be further inspected.

3.2 Cloud Service & API Usage

Cloud services. To identify the provider-managed services most frequently used in our dataset, we look for lines of code that instantiate a client or resource object with the open-source library `boto3` [11]. The latter is an essential component of the analysed

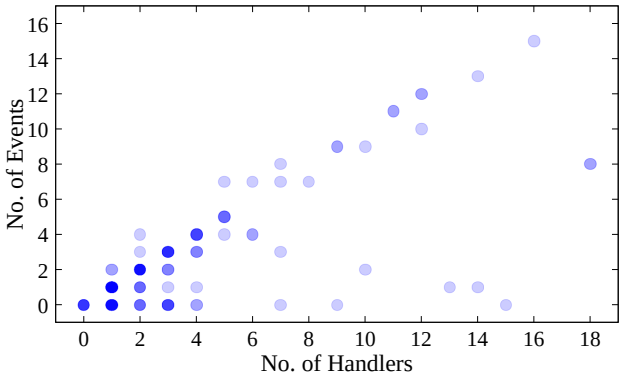

Figure 4: Number of events vs handlers in AWSomePy.

applications as it provides a rich interface to a large number of AWS services. The targeted lines of code, e.g., `boto3.client('s3')` and `boto3.resource('s3')`, can be processed via regular expressions because, despite the differences between client and resource objects[4], the class constructors always require specifying the cloud service as a string.

Our analysis identifies 46 services in total, with those with the highest number of occurrences, computed in terms of `boto3` client and resource objects instantiations, shown in Table 3. Data storage and NoSQL services, i.e., `s3` and `dynamodb`, are significantly more common than the others, with 217 and 201 instantiations detected, respectively. In the case of `s3`, these were found in 59 repositories, whereas they were present in 47 repositories in the case of `dynamodb`.

The third most frequently used service is `lambda`, which allows configuring the serverless platform itself. The fourth is `ssm`, which is used to perform a variety of management tasks. This indicates that despite the wealth of configuration features and the rich plugin ecosystem offered by the Serverless Framework, developers access management-oriented services via their application code.

Cloud APIs. We next provide additional details on how the top five services in our dataset are used by conducting a cloud API-focused analysis. We parse the `boto3` library documentation to extract in a semi-automated manner the APIs exposed by the clients of the AWS services present in our dataset[5]. Our preliminary analysis of the documentation reveals that the sections dedicated to the supported services share the same structure, but there exist some differences. Consequently, implementing a parser capable of dealing with all the services would be time-consuming. We therefore use a simplified documentation parser[6] and subsequently manually validate the obtained results. Unlike an *on-the-fly* processing, i.e., conducted at the same time as the actual analysis of the application code, our

[4]While it is possible to instantiate a client object for all the AWS services supported by the `boto3` library, only some of them can be accessed through a resource object. Additional differences are mentioned in the remainder of this section.
[5]Note that we extract only the APIs supported by the `boto3` client objects. These implement low-level interfaces, which implies that they offer a richer set of APIs in comparison with resource objects [6].
[6]Our `boto3` documentation parser is based on the Beautiful Soup library [21].

Table 3: Top eleven AWS services in AWSomePy. The column Occurrences reports the number of `boto3` client and resource objects instantiations within the relevant repositories.

Services	No. of Repositories	Occurrences
s3	59	217
dynamodb	47	201
lambda	24	47
ssm	14	46
sqs	21	41
sns	11	30
ec2	12	29
sts	9	26
rekognition	8	15
cloudformation	7	14
stepfunctions	9	14

pre-analysis allows storing the parser results, which facilitates their manual validation and management in a version control system.

The API-related information is then used to parse the dataset application code in order to identify relevant lines. For simplicity, we focus our attention on the top five services (Table 3) and, after filtering out the lines of code including the widely adopted API names `close` and `copy`, we manually check the remaining lines to ascertain that they are legitimate `boto3` API calls for one of the services of interest. We recognize that our approach requires manual validation of the API names extracted from the `boto3` documentation and of the lines of code where such APIs are detected, but we believe this constitutes a good compromise between accuracy and ease of implementation. We leave to future work the development of a more automated framework.

Our results are shown in Table 4, which reports the occurrences of the detected `s3`, `dynamodb`, `lambda`, `ssm`, and `sqs` APIs. Interestingly, we observe a similarity between `s3` and `dynamodb`, because their most frequently used APIs, i.e., `put_object` and `put_item`, allow storing information in cloud-based resources. This confirms the importance that these have, given the stateless and ephemeral nature of serverless functions. Moreover, we note that developers rather often create `s3` buckets and `dynamodb` tables *programmatically* via the `create_bucket` and `create_table` APIs. We believe that this trend poses a security challenge, because the configuration of these resources cannot be straightforwardly inspected by analysing the `serverless.yml` file of the application.

As for the `lambda` service, our results highlight that the API with the highest number of occurrences is `invoke`, which supports both asynchronous and synchronous execution of serverless functions. Contrary to our expectations, this implies that some applications do not rely on the facilities provided by the AWS platform for automatic execution of their handlers. While understanding the reasons behind this design choice would require a more in-depth analysis, the manual inspection of ten `invoke` API calls in six applications indicates that, in eight cases, the API is used to facilitate the parameterization of a handler name, which is helpful when there are multiple versions of an application, e.g., production and development. By contrast, the remaining two cases show test scripts that rely on the `invoke` API to artificially trigger the execution of a handler passed as a parameter. Even though a specific permission is required to execute the API in question [10], we emphasize that its usage, unless necessary for testing purposes, affects the application workflows in a way not easily detectable via static analysis, thus potentially leading to security vulnerabilities.

Finally, it is worth mentioning that developers rarely use the `lambda` API `add_permission`, which we detect only seven times. In addition, as far as the `ssm` service is concerned, we observe that three out of the top four APIs[7] allow retrieving information from a provider-managed parameter store. Both these trends are example of security-oriented design patterns present in our dataset.

4 DISCUSSION

The purpose of this section is to further discuss the obtained results and the limitations of our approach.

Application code analysis. We observe that the `boto3` client and resource object instantiations reported in Table 3 are always higher than the respective number of AWSomePy projects. Thus, on average, the analysed applications include multiple instantiations of these objects for the same service. This affects the overall amount of data flows, and we therefore recommend considering the number of such instantiations for the development of serverless-specific complexity metrics.

Security implications. Our architectural analysis shows that the security-focused plugin `serverless-iam-roles-per-function` is used in only 7 applications (Table 2). Considering the functionality that it offers (§ 3.1), we believe that this is one of the most interesting results of this study. Although over-privileged function permissions and roles are among the most critical risks for serverless applications recently identified by PureSec [18] and the Cloud Security Alliance [2], the vast majority of the AWSomePy applications rely on application-wide IAM roles and do not follow the principle of *least privilege*. While this does not imply that all these applications are vulnerable, since the most common services in our dataset are `s3` and `dynamodb`, it could lead, similarly to traditional SQL injection attacks, to loss or unauthorised disclosure of information. In addition, given that serverless applications frequently use such services to store data between different executions, this could be compromised as well.

We emphasize that a comprehensive security analysis of AWSomePy is beyond the scope of this work, but, in the light of our results, it is a possible avenue for future work.

Limitations. As mentioned in § 1, static analysis of serverless applications is challenging due to the variety of events that can trigger the execution of their handlers. It might therefore appear contradictory that the majority of the applications in our dataset comprise less than five events (Fig. 4). To better understand this result, it should be observed that we extract events-related information only from the YAML tag `functions`. This approach has two implications. First, we do not consider that some plugins require specifying events in other parts of the `serverless.yml` file. While

[7] Namely, `get_parameter`, `get_parameters` and `get_parameters_by_path`.

Table 4: Occurrences of the six most widely used APIs for the top five AWS services in AWSomePy. The occurrences of all the other detected APIs are aggregated in the entry *other*. The `ssm` APIs `get_parameters_by_path` and `describe_instance_information` are abbreviated as `get_parameters_by_p` and `describe_instance_i`, respectively.

s3		**dynamodb**		**lambda**		**ssm**		**sqs**	
API	#	API	#	API	#	API	#	API	#
put_object	61	put_item	143	invoke	55	get_parameter	79	send_message	27
get_object	52	scan	64	add_permission	7	put_parameter	18	get_queue_url	16
create_bucket	50	query	62	list_functions	3	get_parameters	7	delete_message	15
upload_file	48	get_item	58	get_policy	3	get_parameters_by_p	3	create_queue	14
download_file	24	update_item	57	get_function	2	list_commands	2	receive_message	13
list_objects_v2	22	create_table	41	list_tags	2	describe_instance_i	1	send_message_batch	2
other	111	*other*	93	*other*	4	*other*	6	*other*	1

one of these plugins, i.e., `serverless-step-functions`, is among the most frequent in AWSomePy, it is used in only 14 out of 145 repositories (Table 2). The second implication is that we do not count events *implicitly* defined by some services, e.g., `dynamodb`, as we focus our attention exclusively on events *explicitly* listed by the developer. However, implicit events do not necessarily trigger the execution of a handler, as this depends on how the latter is configured. Although in the two discussed cases our analysis yields an underestimation of the total number of events, we believe that our results provide a valid starting point for future research.

5 CONCLUSION

In this work[8], we present AWSomePy, a novel dataset of 145 AWS serverless applications developed in Python and compatible with the Serverless Framework. We analyse their architecture by considering serverless-specific parameters, such as plugins, events and handlers, along with the LOC. We also conduct an application code-level analysis that aims to identify the most frequently used cloud services and APIs.

Our results highlight that developers tend to use plugins to facilitate the configuration of their applications and the deployment of complex pieces of functionality. Crucially, the security plugin `serverless-iam-roles-per-function` can be found only in 7 AWSomePy applications, in spite of the fact that 55% of them feature a number of LOC between 26 and 1,000. As for our application code-level analysis, it confirms that data storage and NoSQL services are by far the most commonly used, followed by, interestingly, configuration and management-oriented services. While these provide a high degree of flexibility, they also add workflows that are difficult to inspect prior to deployment. Moreover, the identified APIs indicate that developers make use of programmatic creation of data stores and database-like resources, which has similar security-related implications.

In conclusion, the analysis of the AWSomePy dataset shows that the granular configuration of handler permissions is not widely adopted, and that analysing serverless applications statically to detect misconfigurations and security-sensitive data flows is challenging for real-world serverless applications.

[8]This research was part-funded by EPSRC grant EP/W015927/1.

REFERENCES

[1] Elena N. Akimova, Alexander Yu. Bersenev, Artem A. Deikov, Konstantin S. Kobylkin, Anton V. Konygin, Ilya P. Mezentsev, and Vladimir E. Misilov. 2021. PyTraceBugs: A Large Python Code Dataset for Supervised Machine Learning in Software Defect Prediction. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. 141–151. https://doi.org/10.1109/APSEC53868.2021.00022

[2] Cloud Security Alliance. 2019. The 12 Most Critical Risks for Serverless Applications. Retrieved February 28, 2023 from https://cloudsecurityalliance.org/blog/2019/02/11/critical-risks-serverless-applications/

[3] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure Serverless Computing Using Dynamic Information Flow Control. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 118 (Oct. 2018), 26 pages. https://doi.org/10.1145/3276488

[4] Microsoft Azure. 2023. Power your vision on Azure. Retrieved March 02, 2023 from https://azure.microsoft.com/en-gb/

[5] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. *Serverless Computing: Current Trends and Open Problems*. Springer Singapore, Singapore, 1–20. https://doi.org/10.1007/978-981-10-5026-8_1

[6] Ralu Bolovan. 2018. Python, Boto3, and AWS S3: Demystified. Retrieved March 01, 2023 from https://realpython.com/python-boto3-aws-s3/

[7] Jeremy Daly. 2020. Event Injection: Protecting your Serverless Applications. Retrieved March 02, 2023 from https://www.jeremydaly.com/event-injection-protecting-your-serverless-applications/

[8] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. 2020. Valve: Securing Function Workflows on Serverless Computing Platforms. In *Proceedings of The Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 939–950. https://doi.org/10.1145/3366423.3380173

[9] Pubali Datta, Isaac Polinsky, Muhammad Adil Inam, Adam Bates, and William Enck. 2022. ALASTOR: Reconstructing the Provenance of Serverless Intrusions. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2443–2460. https://www.usenix.org/conference/usenixsecurity22/presentation/datta

[10] Boto3 Documentation. 2022. Lambda client invoke API. Retrieved February 28, 2023 from https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/lambda.html#Lambda.Client.invoke

[11] Boto3 Documentation. 2023. AWS SDK for Python (Boto3) to create, configure, and manage AWS services. Retrieved March 01, 2023 from https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

[12] GitHub REST API Documentation. 2022. Get a Repository API. Retrieved March 01, 2023 from https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28#get-a-repository

[13] Nafise Eskandani. 2021. Wonderless Dataset Github Repository. Retrieved March 01, 2023 from https://github.com/prg-grp/wonderless

[14] Nafise Eskandani and Guido Salvaneschi. 2021. The Wonderless Dataset for Serverless Computing. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 565–569. https://doi.org/10.1109/MSR52588.2021.00075

[15] Julien Lepiller, Ruzica Piskac, Martin Schäf, and Mark Santolucito. 2021. Analyzing Infrastructure as Code to Prevent Intra-update Sniping Vulnerabilities. In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 105–123.

[16] Matthew Obetz, Stacy Patterson, and Ana Milanova. 2019. Static Call Graph Construction in AWS Lambda Serverless Applications. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/obetz

[17] Google Cloud Platform. 2023. Accelerate your transformation with Google Cloud. Retrieved March 02, 2023 from https://cloud.google.com/

[18] PureSec. 2019. The Ten Most Critical Risks for Serverless Applications v1.0. Retrieved March 02, 2023 from https://github.com/puresec/sas-top-10

[19] Pygount. 2023. Pygount Command Line Tool Documentation. Retrieved March 01, 2023 from https://pygount.readthedocs.io/en/latest/index.html

[20] GitHub Repository. 2021. Serverless AWS Pseudo Parameters. Retrieved February 27, 2023 from https://github.com/svdgraaf/serverless-pseudo-parameters

[21] Leonard Richardson. 2023. Beautiful Soup Documentation. Retrieved March 01, 2023 from https://www.crummy.com/software/BeautifulSoup/

[22] Arnav Sankaran, Pubali Datta, and Adam Bates. 2020. Workflow Integration Alleviates Identity and Access Management in Serverless Computing. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 496–509. https://doi.org/10.1145/3427228.3427665

[23] Amazon Web Services. 2023. AWS Solutions. Retrieved March 02, 2023 from https://aws.amazon.com/

[24] Deepak Sirone Jegan, Liang Wang, Siddhant Bhagat, Thomas Ristenpart, and Michael Swift. 2020. Guarding Serverless Applications with SecLambda. *arXiv e-prints* (Nov. 2020). https://doi.org/10.48550/arXiv.2011.05322

[25] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. ACM. https://doi.org/10.1145/3445814.3446714

[26] Datadog Website. 2021. The State of Serverless. Retrieved February 27, 2023 from https://www.datadoghq.com/state-of-serverless-2021/

[27] Datadog Website. 2022. The State of Serverless. Retrieved February 27, 2023 from https://www.datadoghq.com/state-of-serverless/

[28] Serverless Framework Website. 2022. Zero-friction serverless development. Retrieved March 02, 2023 from https://www.serverless.com/

[29] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1556–1560. https://doi.org/10.1145/3368089.3417943