Sound Static Analysis for Microservices: Utopia? A Preliminary Experience with LiSA

Giacomo Zanatta
Ca' Foscari University
Venice, IT

Pietro Ferrara
Ca' Foscari University
Venice, IT

Teodors Lisovenko
Ca' Foscari University
Venice, IT

Luca Negrini
Ca' Foscari University
Venice, IT

Gianluca Caiazza
Ca' Foscari University
Venice, IT

Ruffin White
White Robotics
US

Abstract

Sound static analysis allows one to overapproximate all possible program executions to infer various properties. However, it requires quite some effort to formalize and prove the soundness of program semantics. Most software applications developed nowadays are distributed systems in which different [micro]services communicate through synchronous and asynchronous mechanisms. These applications are composed of programs developed in many programming languages and rely on many technologies. However, sound static analysis might be particularly promising in distributed architectures, where exhaustively (or even partially) testing such systems is often prohibitive. This paper presents our ongoing work on applying LiSA (Library for Static Analysis) to microservices. So far, our effort has focused on one programming language (Python), a few libraries (ROS2, pika, FastAPI, Django), and the architectural reconstruction of distributed applications. However, it already shows some promising results and general patterns that might be followed to develop such analyses.

CCS Concepts

• **Theory of computation → Program analysis**; *Program verification.*

Keywords

Static Analysis, Microservices, Robotics

ACM Reference Format:
Giacomo Zanatta, Pietro Ferrara, Teodors Lisovenko, Luca Negrini, Gianluca Caiazza, and Ruffin White. 2024. Sound Static Analysis for Microservices: Utopia? A Preliminary Experience with LiSA. In *Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '24), September 20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3678721.3686229

1 Introduction

Static analysis is a technique for reasoning about a program's runtime behavior without running it. It relies on a semantic model of the concrete executions of the program and on some forms of abstraction to make the analysis computable w.r.t. some properties of interest. A sound static analysis overapproximates all possible executions. When a sound analyzer infers a property, such property will be satisfied by all its possible executions [3, 4].

Modern software architectures have been based on several independently deployable software units in the last two decades. Those units can communicate (i) synchronously through APIs [12] and (ii) asynchronously through messages [6]. Many distributed architectures were widely adopted, such as layered (e.g., front-end, back-end, and database), service-oriented, event-based, microservices, and serverless [18]. Without entering the details of specific architectures, we focus on the types of communications between different components. In particular, nowadays, most synchronous communications through APIs are based on the REST pattern and exchange data in JSON format. Instead, most asynchronous communications through messages are based on the publish-subscribe pattern supported by different technologies (e.g., Kafka, RabbitMQ).

Existing static analysis techniques primarily focus on applications formed by a single code base, where the source's control flow is modeled by a set of control-flow graphs (CFGs), one for each method. Instead, modern software architectures consist of programs that can be heterogeneous in terms of frameworks, communication patterns, and programming languages. More than traditional static analysis is required to design a static analyzer tool to analyze a whole system. This paper presents our ongoing effort in applying sound static analysis to microservices. Our main goal is to extend and tailor existing static analysis to new software architecture paradigms. All analyses are implemented in LiSA (Library for Static Analysis - https://github.com/lisa-analyzer/lisa) to evaluate the results in practice.

The main challenge we faced during the development of this work was the vast heterogeneity of technologies and programming languages adopted nowadays. Since developing a sound static analyzer is time-consuming, we focused on a single programming language (Python) but supporting different libraries that provide similar semantic patterns. In particular, we focused on asynchronous communications through the publish-subscribe patterns in ROS2 applications and pika microservices and synchronous communications through REST APIs in FastAPI and Django. To generalize standard semantics, we extended SARL [8], a domain-specific language to specify the semantics of libraries, applying it to the abovementioned libraries. Our preliminary experience shows that adopting an intermediate abstraction layer when defining these

FTfJP '24, September 20, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1111-4/24/09
https://doi.org/10.1145/3678721.3686229

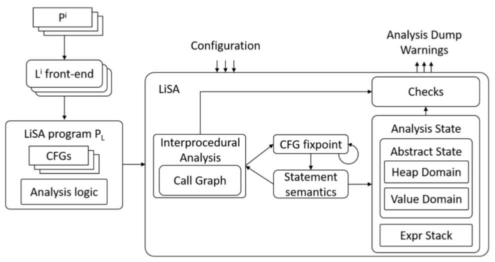

Figure 1: LiSA architecture.

semantics makes tackling systems comprising several different technologies feasible. Our future work plans to extend this experience to other programming languages (in particular, Go [16] and C/C++ through LLVM).

Structure. After discussing the background and related work in Section 2, Section 3 presents LiSA4ROS2[21], a static analyzer tool that reconstructs the architecture of a distributed robotic application, and how we extended it to generic publish-subscribe communication patterns. The core of our ongoing effort on the extension of this analysis to REST API synchronous communications is presented by Section 4. Finally, Section 5 concludes.

2 Background and Related Work

Recent work by Cerny and Taibi [22] investigates the challenges of developing a static analysis tool for microservices, stating that the available tools are currently minimal. They proposed a four-step methodology process for developing microservice-aware static analysis: (i) recognize communication components in code, (ii) establish microservice intermediate representation, (iii) unify across platforms, and (iv) interconnect the intermediate representation of services. Various tools [1, 2] aim to achieve the proposed four-step methodology, such as Prophet[1], a declarative static analysis tool, and MicroDepGraph[2], which analyzes Docker configuration files. However, these tools are unsound by design. In [17], authors discuss microservice smells and propose an extension to the Arcan [10] project that can detect Cyclic Dependencies, Hard-Coded Endpoints, and Shared Persistence. Unfortunately, Arcan is a closed-source application. To our knowledge, a sound, language-agnostic static analysis tool for multi-code-based applications does not exist yet.

2.1 LiSA

In our work, which spans from ROS2 to microservices, we adopted LiSA [9, 13, 14]. LiSA permits building sound, language-agnostic, modular, and extendible static analyzers. LiSA is based on the abstract interpretation framework, which allows for crafting sound abstractions of programs. The underlying motivations that drove us to use LiSA are manifold. In particular, LiSA works on an internal language-agnostic representation of programs. Since our target applications are multi-code-based (and possibly heterogeneous in terms of programming languages) applications, working on a standard, language-agnostic intermediate representation of such programs simplifies our analysis. Instead of having a dedicated analyzer for every language program and combining all the analyses to generate the architectural graph, we can use LiSA to analyze every application component. This allows us to have the same analysis infrastructure and to share the same logic for all the code components of the distributed application. To analyze a source code written in a language L, we need to use a front-end that converts that code into the LiSA's internal representation (aka LiSA Program). LiSA has a dedicated front-end for Python, Rust, and Golang[3], but since it is an open source project, a developer can build a front-end for a not-already official supported language. As a second point, LiSA is extensible, which permits extending the analyzer's architecture to get more precise results by, for example, developing a specific domain for this type of application. Figure 1 shows the architecture of the LiSA library. Starting from a program P written in a language L, we obtain a corresponding LiSA Program by using a front-end for that specific language L. The LiSA Program consists of a set of Control-Flow graphs (CFGs), one for each method. A LiSA CFG models the flow of the instructions of a method or a function. These instructions are defined internally as LiSA Statements and created by the front-end by parsing the source code. Every LiSA Statement has its meaning; that is to say, it represents a specific operation or expression in the source code. The LiSA engine inputs a LiSA Program and evaluates the program's behavior starting from the entry point. This evaluation is performed using an interprocedural analysis that permits the computation of the fixpoint of a CFG through the semantics of LiSA Statements. LiSA's analysis state defines an abstract model of the program's memory and keeps track of the semantics evaluation of the program's statements by reasoning over abstract domains. The internal implementation of LiSA's Abstract State is based on the framework proposed in [7], in which the memory is modeled using an abstract value domain (that tracks values of program variables) and an abstract heap domain (that tracks how the dynamic memory evolves at runtime). A developer can choose to use some already defined abstract domains (such as constant propagation, shape analysis [19], or taint analysis [5, 20]) or can opt to write their specific abstract domains exploiting LiSA's modularity and extendability. To collect valuable information from the analysis state, such as warnings or code smells, one can employ a checker to iterate over the semantics of the program to get meaningful results. For example, as we will see in the following sections, we use a checker to generate the communication graph of a distributed application.

3 Publish-subscribe Communications

In this section, we briefly introduce ROS2 and discuss how we developed a sound static analysis to infer communications through the publish-subscribe pattern in LiSA4ROS2, and the similarities between ROS2 Python libraries and pika, a library to manage AMQP-based communications.

[1]https://github.com/cloudhubs/prophet.
[2]https://github.com/clowee/MicroDepGraph.
[3]All the official front-ends are available here: https://github.com/orgs/lisa-analyzer.

3.1 ROS2

The Robot Operating System 2 (ROS2) [11] is the de facto standard framework for building distributed robotic applications. A ROS2 application is composed of a set of distinct autonomous computational entities, called nodes, that explicitly exchange messages to achieve a common goal with the possibility to choose different communication models: (i) publish/subscribe (using topics), (ii) client/server (using services), and (iii) a preemptable client/server for long tasks (actions). The nodes of a ROS2 system can be heterogeneous in terms of the language used to write client applications. A ROS2 node can be written in Python or C++ using the corresponding client libraries (rclpy for Python and rclcpp for C++). Client libraries contain API calls to the underlying ROS2 architecture and permit developers to write applications quickly without thinking about internal communications and node synchronization.

3.2 LiSA4ROS2

To statically extract the network graph of a ROS2 application, we developed LiSA4ROS2[21][4]. LiSA4ROS2 is a tool that extends the LiSA library by adding capabilities for analyzing ROS2 source code. LiSA supports Python code [15], while the support for a C++ front-end is under development. Therefore, we focused our efforts on ROS2 Python code, while the analysis of C++ is left as future work. LiSA4ROS2 derives `XML`'s minimal access control policies for every node from the extracted network graph. An access control policy enhances the application's security by blocking a node from communicating with channels used for purposes outside its own. LiSA4ROS2 works as follows. First of all, the Python source code of a node is passed to a front-end module to generate a LiSA program, that is, an internal representation of the program understandable by the LiSA library. The obtained LiSA program is then fed to the LiSA analysis engine, which, for every statement, computes an over-approximation of its concrete semantics over an abstract state. For our purpose, we model the abstract state using standard abstract domains already implemented in LiSA. Specifically, we use a field-sensitive point-based heap domain (to keep track of the allocation of ROS2 entities) and a string constant propagation analysis (to track entities' names). Since the internals of the rclpy library contain complex and highly optimized constructs, getting a sound approximation of this library's concrete behaviors is time-consuming and error-prone. For example, to instantiate a publisher, we need to call the `create_publisher` method of a ROS2 node. This method will create a publisher by calling a binding to a C++ function and setting some variables in both nodes and the newly created publisher. For our analysis, some of these variables can be safely ignored. Instead of directly analyzing the source code of rclpy, we adopted a domain-specific language, SARL (Static Analysis Refinement Language) [8], to define a sound approximation of the ROS2 Python library manually. To extract the architectural graph (i.e., the network graph) of a ROS2 application, we use a checker that iterates over the semantics of every statement to find the ones relative to ROS2. For every ROS2 statement that instantiates a ROS2 entity (nodes, subscribers, and publishers), we store the associated entity in a ROS2 network model, keeping track of the source code location and the name. For publishers, subscribers, and other entities related to communication, the checker links such entities with the corresponding owner (aka node). We analyze every node in isolation, extracting its communication links. Then, we glue the output of every analysis to obtain the architectural graph of the application. From the network topology, we derive access control policies.

An example. From ROS2 tutorials, we considered a minimal example where two ROS2 nodes communicate over a topic. In Figure 2, we define a ROS2 node named `minimal_publisher` with a publisher over the topic `topic` by invoking method `Node.create_publisher` (line 8). The node specified in Figure 3 (*minimal_subscriber*) listens on the same topic using a subscription created by invoking method `Node.create_subscription` (line 8). When a message is received in the channel, method `listener_callback` (lines 13-14) is executed. LiSA4ROS2 can extract these entities and generate the graph in Figure 4, where an oval represents a node and a rectangle a communication channel (i.e., topic). An edge from a node to a topic means the node registers itself as a publisher and can send messages over that channel. Vice versa, an edge from a topic to a node means that the node can listen to the messages sent over that channel by subscribing to it. This visual representation effectively illustrates the communication architecture within the system, showing which nodes are interacting through which topics.

```
 1 import rclpy
 2 from rclpy.node import Node
 3 from std_msgs.msg import String
 4
 5 class MinimalPublisher(Node):
 6   def __init__(self):
 7     super().__init__('minimal_publisher')
 8     self.publisher_ = self.create_publisher(String, 'topic', 10)
 9     timer_period = 0.5  # seconds
10     self.timer = self.create_timer(timer_period, self.timer_callback)
11     self.i = 0
12   def timer_callback(self):
13     msg = String()
14     msg.data = 'Hello World: %d' % self.i
15     self.publisher_.publish(msg)
16     self.get_logger().info('Publishing: "%s"' % msg.data)
17     self.i += 1
18
19 def main(args=None):
20   rclpy.init(args=args)
21   minimal_publisher = MinimalPublisher()
22   rclpy.spin(minimal_publisher)
23   minimal_publisher.destroy_node()
24   rclpy.shutdown()
25
26 if __name__ == '__main__':
27   main()
```

Figure 2: Python's ROS2 minimal publisher example

3.3 Extension to Generic Pub-sub Communications

We extended the analyses described above to AMQP-based applications, particularly to the pika Python library, the official Python client library for the RabbitMQ broker. Table 1 highlights similar operations between rclpy and pika. In rclpy, to declare a publisher, we use the method `create_publisher` of a node that will register the node to a specified topic, identified by the String topic taken as an argument. In pika, the same thing is done by the method

[4] https://lisa-analyzer.github.io/lisa4ros2-fe/.

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):
  def __init__(self):
    super().__init__('minimal_subscriber')
    self.subscription = self.create_subscription(
      String,
      'topic',
      self.listener_callback,
      10)
  def listener_callback(self, msg):
    self.get_logger().info('I heard: "%s"' % msg.data)

def main(args=None):
  rclpy.init(args=args)
  minimal_subscriber = MinimalSubscriber()
  rclpy.spin(minimal_subscriber)
  minimal_subscriber.destroy_node()
  rclpy.shutdown()

if __name__ == '__main__':
  main()
```

Figure 3: Python’s ROS2 minimal subscription example

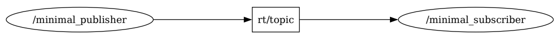

Figure 4: Extracted graph of the minimal example.

`exchange_declare`. However, the main difference is that in the AMQP protocol, we do not have topics; we only have exchanges and queues. In that case, the `exchange_declare` enables the creation of an exchange with a given name (`exc`) and with a specific type (`type`). To publish a message on a topic, we use the `publish` method of a rclpy publisher. Vice versa, in AMQP, we use the `basic_publish` that sends a message (`msg`) with a specific routing key (`key`) to a particular exchange identified by its name (`exc`). The same thing happens with subscriptions (i.e., consumers): the main difference here is that in rclpy, when we create a subscription, we also model the consumptions of the messages; since the operation is atomic, we register a subscription and a callback (`cb`) with a single instruction. The exchange-queue binding of AMQP is omitted from the table since it does not have corresponding ROS2 semantics.

Instead of having a single communication channel between producers and consumers as in ROS2, here we have an active external server (i.e., a broker) that moves messages from exchanges to queues while following some rules. We can model this by exploiting the declaration and exchange-queue binding rules defined in the code. Note that we only sometimes have such information in the code. If the channels are defined as durable, they will survive server’s restarts, and the client could avoid performing a declaration to ensure their existence. Despite this, we can get the corresponding bindings by looking at the code or manually annotating it.

An example. Consider the code in Listings 1 and 2. The first code publishes (lines 6 and 8) something to some exchanges (lines 5 and 7) of an AMQP broker that redirects messages to queues. The second one consumes (lines 16-19) some queues (lines 9-15). We can derive the architecture of this application using the same logic for ROS2 (modeling the pika semantics using SARL and using a dedicated checker to extract communication entities).

```python
1 import pika
2
3 connection = pika.BlockingConnection(pika.ConnectionParameters('localhost')
      )
4 channel = connection.channel()
5 channel.exchange_declare('pika.exchange1', 'direct')
6 channel.basic_publish('pika.exchange1', 'routingKey1', 'Hello World!')
7 channel.exchange_declare('pika.exchange2', 'fanout')
8 channel.basic_publish('pika.exchange2', 'routingKeyFanout', 'Fanout msg')
9 connection.close()
```

Listing 1: pika_tests/amqp1.py.

```python
1 import pika
2 import json
3 import uuid
4
5 def callback(ch, method, properties, body):
6   print(body.decode())
7 connection = pika.BlockingConnection(pika.ConnectionParameters('localhost')
      )
8 channel = connection.channel()
9 channel.queue_declare('queue1')
10 channel.queue_declare('queue2')
11 channel.queue_bind('queue1', 'pika.exchange1', 'routingKey1')
12 channel.queue_bind('queue2', 'pika.exchange1', 'routingKey2')
13 channel.queue_declare('queue3')
14 channel.queue_bind('queue1', 'pika.exchange2', 'routingKey3')
15 channel.queue_bind('queue3', 'pika.exchange2', 'routingKey4')
16 channel.basic_consume('queue1', callback, True)
17 channel.basic_consume('queue2', callback, True)
18 channel.basic_consume('queue3', callback, True)
19 channel.start_consuming()
```

Listing 2: pika_tests/amqp2.py.

4 From LiSA4ROS2 to LiSA4micro(services)

During the development of LiSA4ROS2, we understood that the semantics we defined was quite similar to the one of microservices. Ultimately, they all communicate through HTTP request/reply messages, exchange object states in various formats (such as JSON, XML, and binary), communicate synchronously through REST APIs, and communicate asynchronously through pub-sub communication patterns.

Pitfalls. We discussed the problem of soundly analyzing different languages with one analyzer in Section 2, motivating why LiSA is a good candidate. To say something more, a unique analyzer for heterogenous code-base services permits to (i) have a common logic to perform high-level analyses (i.e., to find problems regarding the architecture, like tight coupling or cyclic dependencies, or to reconstruct the network architecture), and (ii) to discover bugs and issues inside every service. Another critical pitfall is that nodes could communicate in different ways in a microservice architecture. For example, a node could expose a REST API, communicate with other nodes with the RCP protocol, or use an asynchronous message-oriented middleware. Even more, the exposed services can be implemented using different frameworks or libraries: for example, a Python REST service can be implemented using Flask, FastAPI, or Django. Developing an analysis that addresses all the differences between these frameworks becomes unpractical. However, at a high level of abstraction, these frameworks perform the same function: enable communication.

Our approach. We tackled this heterogeneity of programming languages and technologies by extending the aforementioned LiSA SARL. In particular, we introduced new rules to reduce different

Table 1: Comparison of ROS2 and pika

Operation	ROS2	Pika
Declare a publisher	rclpy.Node.create_publisher(topic, ...)	pika.Channel.exchange_declare(exc, type, ...)
Publish a message	rclpy.Publisher.publish(message)	pika.Channel.basic_publish(exc, key, msg, ...)
Declare a subscription	rclpy.Node.create_subscription(topic, cb, ...)	pika.Channel.queue_declare(queue, ...)
Consume messages	rclpy.Node.create_subscription(topic, cb)	pika.Channel.basic_consume(queue, cb, ...)

frameworks to a common abstraction and then used such abstraction to reason about high-level analysis and reconstruct the architectural graph. Intuitively, it does not matter if we use Flask or FastAPI to expose a POST resource. We want to abstract the framework and consider that we are creating a POST resource with a given name and characteristics. This approach reduces the development complexity: there is just a need to write down this reduction using SARL, modeling how a framework creates a communication endpoint. This approach can be extended to other communication paradigms. Consider, for example, the AMQP protocol. Various libraries permit communication with a message-oriented middleware. However, we want to track whether a specific method of a particular library sends a message to an AMQP exchange and whether another permits listening over a queue and receiving messages.

The HTTPService is a synthetic, abstract object that models a concrete communication entity. Instead of analyzing the actual semantics of a communication framework, LiSA converts it to one of its abstractions based on some specified rules. These abstractions are like a lingua franca with their own methods' semantics and attributes. For example, an HTTPService has methods to register new endpoints in the object, and these methods are called by using the rewrite logic discussed above. To model requests to an endpoint (see line 20 of serviceA in Figure 5), we use an HTTPClient. This approach can also be applied to another communication paradigm, like AMQP (reducing it to an AMQPClient), RPC, or MQTT.

Table 2 shows the similarities between FastAPI and Flask. To make it short, they both use annotations over the methods that listen to HTTP requests, but FastAPI has annotators specific for the type of request (GET, POST, ...), while Flask has a unique annotator whose second parameter specifies what type of requests it listens to. Using this methodology, we can exploit properties on the architectural level by working on the extracted graph and also discover local properties by analyzing the semantics of the source. Developers who use this methodology can build their own custom analyses on the graph level and locally by analyzing every service in isolation and implementing a standard LiSA checker. This dual approach allows for comprehensive system-level insights while also providing detailed local information and enabling the capability to extract interesting properties, like data flow and cyclic dependencies.

An example. We developed a prototype implementation showing the analysis' desired outcome and investigating the work's feasibility. As a proof of concept, we target the Python FastAPI library, and we can reconstruct the architecture of a distributed application. Here, we proposed a toy example of a microservice architecture. The application consists of two services, serviceA (Figure 5) and serviceB (Figure 6).

```
1  from fastapi import FastAPI, HTTPException
2  from models import Item
3  import json
4  import requests
5
6  app = FastAPI(debug=True)
7
8  @app.get(path = "/item/{itemID}")
9  def get_item(itemID: int):
10   item = ... #retrieve idemID from the DB
11   return item
12
13 @app.post(path = "/item/")
14 def post_report(item: Item):
15   ... # store item in the DB
16   return item
17
18 @app.get(path = "/report/3")
19 def get_report():
20   response = requests.get("http://serviceB:8000/report/3")
21   return json.loads(response.text)
22
```

Figure 5: Python source for the toy serviceA example.

```
1  from fastapi import FastAPI, HTTPException
2  from models import Report
3
4  app = FastAPI(debug=True)
5
6  @app.get(path="/report/{reportId}")
7  def get_report(reportId: int):
8    report = ... #retrieve reportId from the DB
9    return report
10
```

Figure 6: Python source for the toy serviceB example.

Service A exposes a GET `/item/{itemID}` endpoint (line 8) that retrieves an item from a database and returns it and a POST `/item` endpoint (line 13) that adds a given item to the database. In addition, it exposes a GET `/report/3` (line 18) that performs a GET request to the serviceB. Service B exposes a GET `/report/{reportID}` endpoint (line 6). Our analysis prototype can extract all these endpoints and reconstruct the architectural graph, as shown in Figure 7.

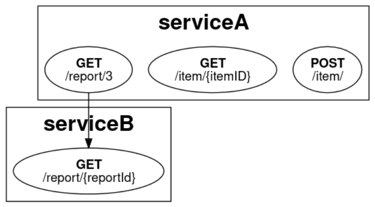

Figure 7: The architectural graph of the toy example.

Table 2: Comparison of FastAPI and Flask

Operation	Fastapi	Flask
Declare a GET endpoint	fastapi.FastAPI.get(endpoint, ...)	flask.Flask.route(endpoint, methods=["GET", ...], ...)
Declare a POST endpoint	fastapi.FastAPI.post(endpoint, ...)	flask.Flask.route(endpoint, methods=["POST", ...], ...)
Declare a PUT endpoint	fastapi.FastAPI.put(endpoint, ...)	flask.Flask.route(endpoint, methods=["PUT", ...], ...)
Declare a PATCH endpoint	fastapi.FastAPI.patch(endpoint, ...)	flask.Flask.route(endpoint, methods=["PATCH", ...], ...)
Declare a DELETE endpoint	fastapi.FastAPI.delete(endpoint, ...)	flask.Flask.route(endpoint, methods=["DELETE", ...], ...)

5 Conclusion

Our preliminary results show that developing a sound static analysis for microservices is not a utopia if we consider only one programming language, only a few existing technologies, and only architecture reconstruction as the property of interest. All this is in a not-yet-production-ready implementation of the analysis.

Future work. First, we are stabilizing our analyses to apply them in an industrial context. Then, as stated in the introduction, we plan to extend our approach to Go and C/C++ applications. However, our final goal is to obtain a valuable tool for developers. These are interested in reports about their code; therefore, we plan to apply the results of our analysis to detect various types of software issues. For example, considering an HTTP service, we can perform a local analysis to determine whether the service has or does not have API versioning and if it has some hardcoded endpoints (i.e., IP and ports). Analyzing the architecture, we can also discover the presence of cyclic dependencies or tightly coupled services, which can hinder the scalability and maintainability of the system.

Acknowledgements

Work partially supported by SERICS (PE00000014 - CUP H73C2200089001) under the NRRP MUR program funded by the EU - NGEU, and by iNEST- Interconnected NordEst Innovation Ecosystem funded by PNRR (Mission 4.2, Investment 49 1.5) NextGeneration EU (ECS_00000043 – CUP H43C22000540006).

References

[1] Alexander Bakhtin, Xiaozhou Li, Jacopo Soldani, Antonio Brogi, Tom Černý, and Davide Taibi. 2023. Tools Reconstructing Microservice Architecture: A Systematic Mapping Study. *Agility with Microservices Programming, co-located with ECSA* (09 2023).

[2] Tomas Cerny, Amr S. Abdelfattah, Vincent Bushong, Abdullah Al Maruf, and Davide Taibi. 2022. Microservice Architecture Reconstruction and Visualization Techniques: A Review. In *Proceedings of the IEEE International Conference on Service-Oriented System Engineering (SOSE 2022)*. 39–48. https://doi.org/10.1109/SOSE55356.2022.00011

[3] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of 4th ACM Symposium on Principles of Programming Languages (POPL 1977)*. ACM Press, 238–252.

[4] P. Cousot and R. Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th Annual ACM Symposium on Principles of Programming Languages (POPL 1979)*. ACM Press, 269–282.

[5] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Ciprian Spiridon, and Fausto Spoto. 2015. Boolean Formulas for the Static Identification of Injection Attacks in Java. In *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2015)*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer-Verlag, 130–145.

[6] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM computing surveys (CSUR)* 35, 2 (2003), 114–131.

[7] Pietro Ferrara. 2016. A generic framework for heap and value analyses of object-oriented programming languages. *Theoretical Computer Science* 631 (2016), 43–72. https://doi.org/10.1016/j.tcs.2016.04.001

[8] Pietro Ferrara and Luca Negrini. 2020. SARL: OO Framework Specification for Static Analysis. In *Proceedings of the 12th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2020) (Lecture Notes in Computer Science, Vol. 12549)*. Springer, 3–20. https://doi.org/10.1007/978-3-030-63618-0_1

[9] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. 2021. Static analysis for dummies: experiencing LiSA. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP 2021)* (Virtual, Canada) *(SOAP 2021)*. ACM Press, 1–6. https://doi.org/10.1145/3460946.3464316

[10] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian Tamburri, Marco Zanoni, and Elisabetta Di Nitto. 2017. Arcan: A Tool for Architectural Smells Detection. In *Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW 2017)*. 282–285. https://doi.org/10.1109/ICSAW.2017.16

[11] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. 2022. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics* 7, 66 (2022), eabm6074.

[12] Mark Masse. 2011. *REST API design rulebook: designing consistent RESTful web service interfaces.* " O'Reilly Media, Inc.".

[13] Luca Negrini, Vincenzo Arceri, Luca Olivieri, Agostino Cortesi, and Pietro Ferrara. 2024. Teaching through Practice: Advanced Static Analysis with LiSA. In *Formal Methods Teaching*.

[14] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. 2023. *LiSA: A Generic Framework for Multilanguage Static Analysis*. Springer Nature, 19–42. https://doi.org/10.1007/978-981-19-9601-6_2

[15] Luca Negrini, Guruprerana Shabadi, and Caterina Urban. 2023. Static Analysis of Data Transformations in Jupyter Notebooks. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP 2023)*, Pietro Ferrara and Liana Hadarean (Eds.). ACM, 8–13. https://doi.org/10.1145/3589250.3596145

[16] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, and Fausto Spoto. 2023. Information Flow Analysis for Detecting Non-Determinism in Blockchain. In *Proceedings of the 37th European Conference on Object-Oriented Programming (ECOOP 2023) (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:25. https://doi.org/10.4230/LIPIcs.ECOOP.2023.23

[17] Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, and Davide Taibi. 2020. Towards Microservice Smells Detection. In *Proceedings of the IEEE/ACM International Conference on Technical Debt (TechDebt 2020)*. 92–97.

[18] Mark Richards and Neal Ford. 2020. *Fundamentals of software architecture: an engineering approach.* O'Reilly Media.

[19] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1999. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999)*. ACM Press, 105–118. https://doi.org/10.1145/292540.292552

[20] Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. 2019. Static Identification of Injection Attacks in Java. *ACM Trans. Program. Lang. Syst.* 41, 3 (2019), 18:1–18:58. https://doi.org/10.1145/3332371

[21] Giacomo Zanatta, Gianluca Caiazza, Pietro Ferrara, Luca Negrini, and Ruffin White. 2024. Automating ROS2 Security Policies Extraction through Static Analysis. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[22] Tom Černý and Davide Taibi. 2024. Microservice-Aware Static Analysis: Opportunities, Gaps, and Advancements. In *Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022)*. https://doi.org/10.4230/OASIcs.Microservices.2020-2022.2