The Journal of Systems & Software 202 (2023) 111722

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Automatic extraction of security-rich dataflow diagrams for microservice applications written in Java☆

Simon Schneider *, Riccardo Scandariato

Institute of Software Security, Technical University Hamburg, Hamburg, Germany

A R T I C L E   I N F O

Article history:
Received 18 October 2022
Received in revised form 24 February 2023
Accepted 18 April 2023
Available online 24 April 2023

Keywords:
Dataflow diagram
Automatic extraction
Security
Microservices
Architecture reconstruction
Feature detection

A B S T R A C T

Dataflow diagrams (DFDs) are a valuable asset for securing applications, as they are the starting point for many security assessment techniques. Their creation, however, is often done manually, which is time-consuming and introduces problems concerning their correctness. Furthermore, as applications are continuously extended and modified in CI/CD pipelines, the DFDs need to be kept in sync, which is also challenging. In this paper, we present a novel, tool-supported technique to automatically extract DFDs from the implementation code of microservices. The technique parses source code and configuration files in search for keywords that are used as evidence for the model extraction. Our approach uses a novel technique that iteratively detects new keywords, thereby *snowballing* through an application's codebase. Coupled with other detection techniques, it produces a fully-fledged DFD enriched with security-relevant annotations. The extracted DFDs further provide full traceability between model items and code snippets. We evaluate our approach and the accompanying prototype for applications written in Java on a manually curated dataset of 17 open-source applications. In our testing set of applications, we observe an overall precision of 93% and recall of 85%. The dataset created for the evaluation is openly released to the research community, as additional contribution of this work.

1. Introduction

The microservice architecture has seen increasing adoption over the last years as it addresses some problems that modern, often cloud-based systems face when following a monolithic approach (Dragoni et al., 2016). Separating an application into *microservices* (or simply *services*) such that each service realizes a single business functionality and is implemented as an independent system with dedicated resources is advantageous for its scalability, deployment, development, and maintenance (Chen et al., 2017; Jamshidi et al., 2018). However, this also introduces additional security challenges, which lie mainly in securing the extended attack surface, in the inter-service communication channels needed to achieve the business functionality, and in establishing trust between services (Hannousse and Yahiouche, 2021; Li et al., 2019a; Pereira-Vale et al., 2021; Yarygina and Bagge, 2018).

Many of these challenges can be tackled by analyzing the architectural model of the microservice applications (in short *microservices*) and by identifying the possible security weaknesses. However, a suitable model needs to be available. In this paper, we focus on dataflow diagrams (DFDs) for representing microservice applications, since they are the starting point for many security assessment techniques, like threat analysis (Sion et al., 2018b; Hernan et al., 2006; Microsoft Corporation, 2016; Torr, 2005) and automated model analysis (Abi-Antoun et al., 2007; Abi-Antoun and Barnes, 2010; Berger et al., 2016; Tuma et al., 2019). In fact, DFDs have been described as *threat model diagrams* because of their frequent use for this purpose (Shostack, 2014). Additionally, since the architecture maps well to DFDs, they are also used specifically for microservices (Chen et al., 2017; Stojanovic et al., 2020; Li et al., 2019b). DFDs depict the system components as nodes and the connections between them as directed edges. However, the DFD notation is rather minimalist and needs to be augmented with annotations that can give further details on components and connections, for example about the deployed security mechanisms or other properties that are useful for analysis.

Although they can have positive impact on application security, DFDs are often seen by developers as being tedious and time-consuming to create. Bernsmed et al. (2021) triangulated four studies to derive benefits and obstacles of using DFDs in companies. The results show that developers, on average, do not like creating them and struggle with details, for example finding the correct granularity or knowing what to include in the diagrams. The microservice architecture brings further obstacles,

☆ Editor: Christoph Treude.
* Corresponding author.
E-mail addresses: simon.schneider@tuhh.de (S. Schneider), riccardo.scandariato@tuhh.de (R. Scandariato).

https://doi.org/10.1016/j.jss.2023.111722
0164-1212/© 2023 Elsevier Inc. All rights reserved.

since (i) applications can be comprised of large numbers of services that are loosely coupled by design, and (ii) their deployment is fluid, with service instances being deployed or taken down regularly. This situation results in an increased cognitive load for the developers or security analysts that are tasked with creating DFDs of microservice systems. Furthermore, the microservice approach is often used in the context of CI/CD pipelines, where the application is modified continuously, hence creating a potential disconnection between (manually created) models and (fast changing) code over time.

Contributions. In this paper, we present a novel approach that extracts security-rich DFDs from microservices' source code written in Java. To the best of our knowledge, this is the first paper focusing on DFDs and microservices for security. As we show in Section 9, some approaches exist that are able to extract the basic architecture of microservices. We decide against extending one of these in favor of developing an approach of our own, since the model items *beyond* the basic structure are the challenging part in our work and, thus, we opt for having a coherent approach and implementation. We choose Java as the target language for our prototype, since a search on GitHub for terms such as 'microservice' retrieves the highest number of results in Java, indicating it to be the most used language for this purpose at the moment. Further, we focus on applications using the Spring framework[1], as it is by far the most used framework for developing microservices in Java according to a recent report which states that 74% of developers that work on microservices use Spring for their work (JRebel, 2022).

Our model extraction technique is based on textual analysis (i.e., code as text) where the source code and the configuration files are statically parsed in search for keywords that, once found, are used as evidence for the creation of corresponding model items. The novel technique that we employ iteratively detects new keywords to connect known commands with developer-chosen identifiers, thereby *snowballing* through an application's codebase. Combined with a direct keyword search and parsing of configuration-, build-, and IaC-files (Infrastructure as Code), the outcome of our approach is a fully-fledged DFD with an extensive set of presented security features. The approach is able to extract the structure of the application, including (i) the services developed to implement the business logic, (ii) the support services that are deployed in the infrastructure (databases, registry, API gateways and so on), as well as (iii) the information flows among these services. Additionally, we locate the security features (e.g., use of encryption, authentication, load balancers, and so on) and other system properties in the codebase[2] and, accordingly, enrich the DFD model with security annotations and other information that might be relevant for an application assessment from a security standpoint. In this respect, the **challenge** that we faced in developing the approach lies in the detection of (i) inter-service communication, which is often implicit and requires the analysis of numerous components across the codebase to grasp all properties of a single connection, and (ii) the identification of many security and availability mechanisms, which are implicitly deployed within other techniques. For example, when detecting information flows via message brokers, the sending and receiving service and the message broker service are all possible places where configurations of queues/exchanges, etc. can exist, which influence the security properties of a connection.

Benefits. The benefits of our technique are twofold. First, it is lightweight and runs fast (on average under five seconds). Therefore, it can be easily integrated in a CI/CD development environment. Second, it provides explainable results. In fact, our technique can produce complete traceability information between the generated model and the codebase. That is, it is possible to view the code snippets that have been used as evidence to create a model item (e.g. an information flow, or a security annotation). This information is meant to assist both security assessors who need to perform a security analysis of the whole application, as well as developers who want to ascertain the architectural (and security) impact of their commits.

Research questions. This work addresses the following (feasibility related) research questions concerning the extraction of dataflow diagrams from microservice applications written in Java with the technique summarized above:

RQ1 What is the precision and recall concerning the extraction of all model items?

RQ2 What is the precision and recall concerning the extraction of just the DFD core model items (services, databases, external entities, and information flows)?

RQ3 What is the precision and recall for the extraction of just the annotations that are relevant for security?

RQ4 What is the execution time for the full extraction?

Validation. In order to answer the above questions, we implemented a prototype (called *Code2DFD* and released to the open domain Schneider and Scandariato (2023)) and used it in a validation experiment on a pool of open-source applications. This was a challenge because no suitable dataset could be found in the public domain. While there is an abundance of suitable open-source projects, no DFDs could be found in literature where the model elements are traced to the code. Therefore, in previous work, we created a dataset from scratch, which could be used as ground truth for our experiments. The evaluation is performed by applying the prototype to the manually curated dataset of 17 microservice applications that have been inspected by four researchers.[3] For each application, the dataset provides the DFD and traceability.

Paper structure. The rest of this paper is organized as follows. In Section 2 we introduce the DFD items covered by our approach. In Section 3 we present our novel approach in detail and Section 4 describes the prototype we built. Section 5 describes the evaluation methodology we followed and Section 6 provides the evaluation results, which are further discussed in Section 7. Section 9 presents the related work and Section 10 concludes this paper.

2. Mapping microservice code to a DFD

Since DFDs do not have a precise definition as to what model items can be included in them, some variations are used throughout the related literature. What all DFDs have in common are four core item groups, *external entities*, *data flows*, *processes*, and *data stores*, as already described early in the emergence of this model type (DeMarco, 1979; Bruza and van der Weide, 1989; Larsen et al., 1998). Additionally, many approaches that work with DFDs add further information to them. As Sion et al. described (Sion et al., 2020), DFDs comprising only the four basic groups of model items lack expressivity in terms of security concepts and should thus be extended. As possible solution, multiple approaches add trust boundaries or different kinds of annotations to DFDs (Berger et al., 2016; Tuma et al., 2019; Sion et al., 2018a;

[1] https://spring.io – In this paper, we refer to the different Spring projects (e.g. Boot, MVC, Cloud, Security) collectively as 'Spring framework'.

[2] In this paper, we jointly refer to both Java and configuration files as 'code'.

[3] The dataset is available under https://tuhh-softsec.github.io/microSecEnD/ or via its DOI (Schneider et al., 2023).

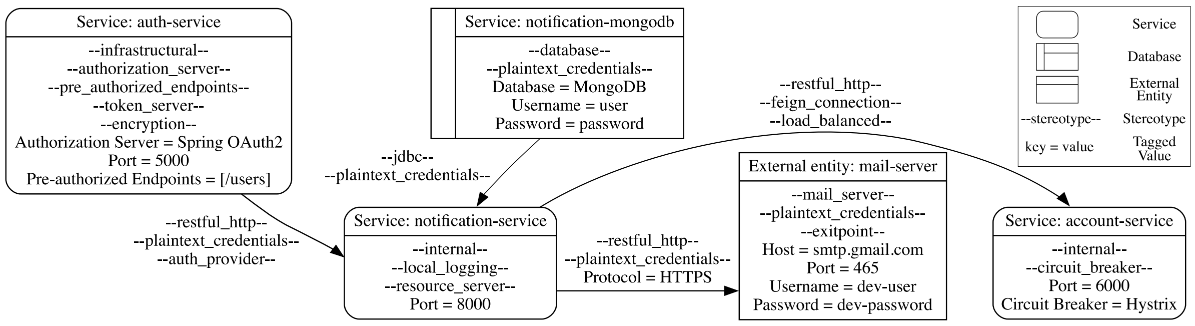

Fig. 1. Example of extracted model items. Excerpt of the DFD for Piggy Metrics; not all annotations shown for better readability.

Tuma et al., 2018) to enhance their usability for security analysis. In our approach, we add annotations capturing information about implemented security and other features, thereby directly addressing some of the concerns raised by Sion et al. (2020).

In this paper, DFDs consist of the four base item groups and the additional group of annotations (which are described later in Tables 1 and 2). Fig. 1 shows an excerpt of the DFD extracted from the application PiggyMetrics.[4] A DFD is a directed graph that contains processes (e.g., microservices), external entities (e.g., clients or third-party systems), data stores (e.g., databases) and information flows among them. Later on, we collectively refer to services, external entities and data stores as *nodes* in the graph. In the case of microservice applications, processes (at their finest granularity) correspond to services. Data stores are also services and, hence, are represented as processes that carry the 'database' annotation and have a dedicated visualization. Finally, note that each element in the diagram can be annotated with additional (security) information.

There are no definitive rules for mapping code elements to a dataflow diagram and in the following paragraphs we describe where the items of each group (services, external entities, information flows and annotations) are discovered in the code.

Services. Microservices are directly mapped to processes (in DFD terminology) and are represented in the figure as rectangles with rounded corners. In the code, we find services in build files (like Maven) or in Infrastructure as Code files (like Docker and Docker Compose). To differentiate the business logic implemented by the application from supporting, off-the-shelf components, we further distinguish between *internal microservices* (processes), *infrastructural microservices* (processes), and *databases* (data stores). Internal services (e.g., 'notification-service' and 'account-service' in Fig. 1) are developed in the project repository and contain the application's functional code. Infrastructural microservices (e.g., 'auth-service' in Fig. 1) mainly use existing code libraries and make only marginal adjustments in order to fit to the application's needs. Examples include, but are not limited to, message brokers, service discovery solutions, API-gateways, web servers, or configuration servers. Databases are services with the additional specification of being used as data stores. In the DFD, they appear as rectangles with an additional vertical line on the left (see 'notification-mongodb' in Fig. 1).

External Entities. These nodes are either clients (i.e., users) or software components that are not deployed with the application, like the Google mail server, a web site, or a GitHub repository used to fetch configuration data. As shown in Fig. 1 ('mail-server'), external entities are represented as rectangles. A client is added to the DFD implicitly to represent the human user when either a gateway or a web application is identified. External software components are called from the application code but there is no deployment descriptor specifying them (e.g., a Dockerfile). This also includes the case of references to (not co-deployed) databases and results in the creation of an external entity (with the stereotype 'database') rather than a data store.

Information Flows. These represent connections between two instances of services and/or external entities over which information is shared. They are depicted as directed edges in the DFD. From the perspective of mapping the code to the model, we consider API calls, connections to databases, and message exchanges via message brokers. Flows are also induced by identifying infrastructure services. For instance, when a configuration service is found, a flow is created to services that specify it as their source of configuration data. Information flows have a direction indicated by an arrow tip. We consider the direction to be defined by which service or external entity provides data that is used by the other.

Annotations. The DFD elements described above are enriched with additional annotations. Annotations are *stereotypes* and *tagged values* (key/value-pairs). In Fig. 1, the annotations are shown as text labels: stereotypes as –*stereotype*–, tagged values as *key* = *value*. Stereotypes represent the semantic purpose of a model element (e.g., a process is an 'authorization_server') or provide some non-functional property of the item (e.g., an information flow carries 'plaintext_credentials'). A list of available stereotypes is presented later in Tables 1 and 2. Tagged values are used to represent additional attributes, such as ports, API-endpoints, or specific technology solutions. There are no formal constraints to their content and they are used to provide additional information for a stereotype. For instance, in the external entity of Fig. 1, the tags are used to track the username and password used in a plaintext credential (the latter being a stereotype). The detection of stereotypes and tagged values is done in all types of files throughout the code.

Traceability information. With our textual analysis approach, it is also possible to easily extract traceability information for all model items described above. The detection of keywords inherently produces this information as a side-effect and only has to be collected in a structured and coherent way. Later in Section 5.2, we provide an example of how the traceability looks like in our dataset and in the models extracted by our technique. In summary, the traceability information contains the location of the evidence related to a model item, and is given as the containing file's path and the lines of code.

With these descriptions of the different items in place, we can model DFDs. Similarly to the standard definition of graphs as a set of vertices and a set of edges, we define a dataflow diagram as a tuple $d = (N, I)$ where N is a set of nodes and I a set of information flows. A node $n \in N$ consists of a name, type $type \in \{service, database, external\ entity\}$, set of stereotypes S, and set of tagged values T. An information flow $i \in I$ consists

[4] https://github.com/sqshq/piggymetrics.

of a sender node $sender \in N$, receiver node $receiver \in N$, set of stereotypes S, and set of tagged values T.

$$n = (name,\ type,\ S,\ T) \in N$$
$$i = (sender,\ receiver,\ S,\ T) \in I$$

The set of all stereotypes as defined by our implementation is presented in Tables 1 and 2.

3. Approach in detail

As DFDs are representations of software systems, their manual creation from code usually consists of sifting through the codebase for artifacts that indicate the existence of an item in the model. Commands (e.g., program instructions, script commands) carry a meaning for the creation of DFDs because of the functionality they invoke in the system. We leverage this circumstance to define a technique that extracts DFDs automatically and via a static analysis of the source code. As will be shown in the following, some items can be detected by looking for single keywords in the code. For other items, however, the evidence is scattered across the codebase. For example, detecting a RESTful API connection between microservices requires detecting the creation of the endpoint at the receiving service, the detection of the request at the sending service, and possibly the detection of further properties that define the connection, such as the use of a load balancer, a circuit breaker, or encryption. Each of these consists of multiple commands in possibly different places. The challenge thus lies in *connecting the dots*, i.e. deriving coherent statements about a system property (or feature) from a complex set of commands in the code.

In our approach, we employ three different techniques to detect evidence for DFD items: *parsing*, *direct keyword search*, and *iterative keyword search*.

3.1. Parsing

Parsing is the simplest of the three techniques. It is used on structured documents to retrieve information stored in a defined format. Applied to Infrastructure as Code- and build-files, it reveals the list of microservices in an application and thus provides the initial "building blocks" of the DFD. When Docker or Docker Compose is used in the analyzed application, the images used to build a service can further indicate their functionality. We make use of this by checking whether the images of all services are on a list of well-known and widely-used ones that imply their functionality, such as Apache httpd, Kafka, or RabbitMQ.

A second important source of information where parsing is applied are configuration files. They hold many properties such as the port or name of a service as well as connections between services. The following excerpt shows part of a `bootstrap.yml` file. Parsing it yields the service's name ('notification-service', as seen in Fig. 1) as well as an information flow from another service called 'config'.

```
spring:
  application:
    name: notification-service
  cloud:
    config:
      uri: http://config:8888
```

3.2. Direct keyword search

Other items are detected via direct keyword search, as single commands can often be sufficient evidence for items in the DFD. For instance, Java annotations are a powerful tool to implement complex functionality with a single command in Java applications. They can thus be leveraged to detect this functionality by merely detecting the keyword of the annotation itself. An example for such an annotation is

```
@EnableAuthorizationServer
```

which will start up a Spring OAuth2 authorization server.[5] Detecting this keyword in the code is evidence for the existence of the service 'auth-service', as seen in Fig. 1.

For other model items, the direct keyword search is used as a primer to identify places in the code base where further extraction functions can be applied. The following example illustrates such an occasion for the information flow from 'notification-service' to 'account-service' in Fig. 1. The following line of code associated with the 'notification-service' creates a Feign client[6] connection to the 'account-service', which is given as parameter:

```
@FeignClient(name = "account-service")
```

A search for '@FeignClient' is used to find lines of code that implement a connection between services in this way. The target service can then be extracted based on the knowledge of the command's syntax and the corresponding documentation, thus, has to be consulted when developing extractors. The terms used for the final extraction, such as 'name' in the example above, are found in numerous other places in the code with differing meaning because they are quite generic. A search for the unambiguous keyword '@FeignClient' is thus needed to localize the lines of code where the extraction function can be applied without errors.

These two ways in which a direct keyword search is applied can detect many of the DFD items and are easily adaptable in new extraction implementations by providing the keywords used for searching and possibly the functions that extract the desired parameters from the findings, which often could be a simple regular expression.

3.3. Iterative keyword search

The direct keyword detection ceases to work reliably when developer-chosen identifiers are introduced, because the inherent connection between keyword and functionality is broken. Although adhering to naming conventions and choosing descriptive identifiers can retain this connection while improving the code's quality (Butler et al., 2009), automatic techniques have to follow the assumption that identifiers are not descriptive enough to base analyses on them. Neglecting the variability introduced by this and detecting model items by matching keywords against identifiers would be unreliable.

As a solution for this problem, we adopt an iterative keyword search technique, which bridges the gap between known keywords and developer-specific identifiers. This simple, yet effective technique works in three steps:

1. search for known keywords used as seeds indicating the use of a security feature (e.g., class names from a known security library)
2. extract new keywords from found instances (identifiers of instantiated classes)
3. search for new keywords (extracted identifiers) in combination with further known keywords (methods of the classes)

[5] https://docs.spring.io/spring-security-oauth2-boot/docs/current/reference/html5.

[6] https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-feign.html.

In the example in Fig. 1, 'auth-service' uses Spring framework's `BCryptPasswordEncoder` to encode a password. After importing the module, an instance of the class is created and named 'encoder':

```
private static final BCryptPasswordEncoder encoder
    ↪ = new BCryptPasswordEncoder();
```

The iterative search finds this line via the initial keyword 'BCryptPasswordEncoder' and extracts 'encoder' to be used as keyword for the next iteration. The second iteration leads to the discovery of the line where the encoding of the user's password takes place further down in the code:

```
String hash = encoder.encode(user.getPassword())
```

This is the place where the feature is found. However, the approach provides a more complete traceability of code chain related to this feature.

The search is not restricted to a single file, but can also 'jump' to other files. For this, when the extracted new keyword contains dot notation, the iterative search checks if there is a file in the same namespace that matches the part before the dot. In that case, it looks in this file for the keyword specified after the dot. Another context switch that is checked for, is the use of environment variables. If the extracted identifier begins with a $ and is enclosed with curly braces, the search mechanism checks for the existence of a `.env`-file to look up the variable.

In our experience, in the context of Java, this three-step process covered all occurrences of developer-introduced identifiers. More iterations might be necessary for other languages or artifacts, but the concept remains similar. We note that the presented technique is an effective solution for code using known libraries, but does not work automatically for functions implemented entirely in the analyzed system, since the known keywords (general purpose functions) do not reveal the overall functionality. However, security functions should not be implemented in such a way.

3.4. Combining the techniques to extract DFDs

The three presented techniques are all able to detect (pieces of) evidence for DFD model items in source code. However, the challenge often lies in piecing the pieces together. To achieve this, we suggest the use of extractors such that the results from multiple independent detection modules are merged and integrated into a single DFD. These extractors are designed such that they can be executed automatically one after another.

We introduce the concept of a *technology-specific extractor* (or simply *extractor*) as the detection functionality for a single technology solution. A technology-specific extractor can detect the existence of an item or a small number of items to be added to the DFD by finding the according evidence in the code. Each extractor takes as input the latest state of the extracted DFD and accesses the source code via a search functionality that encompasses the three techniques described above. It returns the input DFD with possibly added items (nodes, information flows, or annotations to already existing nodes and flows). Additionally to extracting DFD items, technology-specific extractors can also produce *intermediate features*, which are part of but not complete evidence for items. These get passed along with the DFD and if later model extractors detect the missing parts of the evidence, the corresponding item is added. For example, parsers for configuration files are executed first in the extraction process to gather intermediate features that give context to later extracted DFD items. An example for such a feature is indicated by the entry

```
server.ssl.enabled = True
```

which will have the effect that all information flows that are detected connected to the corresponding service will be marked as using HTTPS.

Finally, we remark that there are multiple possibilities to implement any given item in the model (e.g., communication encryption). Therefore, there can be multiple technology-specific extractors for the same model item, each specific to a different technology. Both Mayer and Weinreich (2018) and Mosser et al. (2020) recognized this need to have a multitude of extraction mechanisms for different technologies in an environment as diverse and fast changing as microservices. For instance, Mosser et al.'s approach consists of a set of *probes*, which are manually selected by the user to be run on a certain part of the codebase and enhance a *map* of the software system. With our technology-specific extractors, we follow a similar approach, however, they do not need to be selected and executed manually but are called automatically. This requires some consideration of their execution order, because extractors might use intermediate features from other extractors in order to unfold their full functionality.

3.5. Generalization beyond java

This work focuses on Java applications using the Spring framework . The challenges that static code analysis faces in a field as heterogeneous as that of microservices have been identified by others already (Mosser et al., 2020; Černý and Taibi, 2022; Schiewe et al., 2022). We foresee the interest and necessity to extend our analysis also to other programming languages or frameworks in order to make it useful for a larger part of microservice applications. To evaluate the feasibility of this task, here we identify the modules that are currently Java- and Spring-specific and reason about their generalizability.

Some of the extractors are already independent of the programming language and framework, because they parse information in configuration files and other generic technologies (e.g. Docker). For other technology-specific extractors that operate at code level and use the keyword search techniques (direct or iterative), we foresee the need to change the set of keywords they use, e.g., to reflect the identifiers used in the third-party security libraries in other languages and frameworks. For implementations covering other Java frameworks, this constitutes the provisioning of new domain knowledge to Code2DFD, a task that can be done fairly easily by consulting the corresponding documentations. Allowing the analysis of other languages than Java requires additional changes, since some search techniques depend on the language syntax of (currently) Java. For instance, to switch context across files, the search technique assumes the use of the dot notation. However, as we mostly rely on the analysis of the programming code as text, we speculate that dependencies on language-specific elements are not too difficult to adapt. To this regard, we emphasize that the design of our tool with its use of technology-specific extractors, the three described detection techniques, and the proposal of what items to consider and how to present them is a solid and valuable foundation for similar tools focusing on other languages or frameworks.

In summary, we believe that porting the existing Java- and Spring-based extractors to other frameworks or languages, like Python, Node, or Go, should be a limited effort, given that our reference implementation for Java Spring is now publicly available and can be used for guidance. However, this is an interesting aspect that we plan on investigating soon.

Table 1
List of stereotypes for services in DFDs.

Service (process or data store)		
Stereotype	*Semantic*	
administration_server	Central administration server for all instances.	
configuration_server	Server for managing configurations.	
database	Service is used as datastore.	
gateway	Dedicated entry point for requests into the system.	
infrastructural	Service that is part of the supporting infrastructure of the system.	
internal	Service implementing the business logic.	
in_memory_authentication	Service performs in-memory authentication.	
in_memory_datastore	Service uses an in-memory datastore.	
message_broker	Message broker for service-to-service communication.	
search_engine	Service used in the presentation of logging data.	
service_discovery	Service registry keeping track of running instances.	
web_application	Web application as front-end for the system.	
web_server	Service is a web server.	
Security stereotype	*Semantic*	*Security objective*
authentication_scope_all	Service enforces authentication for all incoming requests.	Authentication
authorization_server	Central server for managing authorization decisions.	Authorization
basic_authentication	Basic HTTP authentication enabled for this service.	Authentication
csrf_disabled	Use of CSRF tokens to protect against these attacks is disabled.	Authentication
circuit_breaker	Mechanism to prevent cascading failures.	Availability
encryption	Encryption mechanisms are employed.	Confidentiality
load_balancer	Distributes requests to a service between all instances of it.	Availability
local_logging	Logging locally at a service.	Availability, non-repudiation
logging_server	Central storage server for log messages.	Availability, non-repudiation
metrics_server	Central storage server for metrics.	Availability
monitoring_dashboard	Central visualization of monitoring information of services.	Availability, confidentiality
monitoring_server	Central service collecting monitoring information from services.	Availability, confidentiality
plaintext_credentials	Plaintext password and/or username found in code.	Confidentiality
pre_authorized_endpoints	Endpoints for which authorization is checked before execution.	Authorization
resource_server	Resource server in the OAuth protocol.	Authorization
ssl_enabled	Process enforces SSL on connections to it.	Confidentiality, integrity
token_server	Service managing access tokens.	Authorization
tracing_server	Central service collecting tracing information from services.	Availability

4. The Code2DFD prototype

To evaluate our approach, we implemented a prototype in Python that is able to analyze microservice applications written in Java. The prototype is called *Code2DFD* and is publicly available at Schneider and Scandariato (2023). Following the concept of technology-specific extractors, Code2DFD is structured as a framework: a set of general functionality (the three search techniques presented in Section 3, visualization, model management, and similar) builds the foundation on top of which a series of extractors can be hooked. Our prototype is able to detect the model items presented in Section 2 for multiple technology solutions. We have currently implemented 43 extractors (see the repository for the list and code), covering a large portion of the most prominent technology solutions used for developing microservice applications in Java. Tables 1 and 2 give the complete list of stereotypes extracted by the tool, each with a short description. At the top layer of the prototype, there is a module that orchestrates the execution of the extractors and combines all detected items into the final DFD.

The tool is implemented as a standalone service with a REST API but can also be run in the terminal for more verbose status information during the analysis. As input, it expects a handle to the git repository or path to the local directory containing the microservice application. In the former case, the repository will be cloned locally first, where the search is performed based on `grep`. A keyword search via GitHub's Search API has shown to be much slower. As output, the DFDs are stored textually using the CodeableModels[7] format and graphically in PNG format.

[7] https://github.com/uzdun/CodeableModels.

5. Evaluation methodology

To evaluate our approach, we created a dataset of DFDs in previous work (Schneider et al., 2023) by manually inspecting 17 open-source microservice applications. We report on the creation here as well for completeness and since it was done specifically for the purpose of evaluating our approach. For the evaluation, the dataset has been partitioned into a reference set (7 apps) and validation set (10 apps), each used to develop and test our approach, respectively.

5.1. Dataset creation

No dataset of (microservice) application code paired to DFDs could be obtained from open-source resources, related literature, or other sources. Most publicly available DFDs are of exemplary nature and do not correspond to actual implementations. For the ones that depict real systems, the underlying code is usually not accessible. We thus curated a dataset ourselves.

The applications are selected from three sources:

1. Lists published by Alshuqayran et al. (2018), Márquez and Astudillo (2018), and Rahman et al. (2019)
2. Search on GitHub for *microservice spring*
3. Referrals by forums, Google search, etc. during implementation of certain technology-specific extractors

While the third source did not follow a systematic methodology, applications from this category were only used for reference during development to investigate different implementations of the same concept and make the implementation more generic. The validation set consist only of applications from the lists found in literature and the most prominent ones on GitHub based on their relevance.

Table 2
List of stereotypes for information flows and external entities in DFDs.

Information flow		
Stereotype	*Semantic*	
feign_connection	Connection realized using Spring's FeignClient.	
jdbc	Database connection using JDBC protocol.	
message_producer_kafka	Sending to a Kafka Queue.	
message_producer_rabbitmq	Sending to a RabbitMQ exchange.	
message_consumer_kafka	Listening to a Kafka Queue.	
message_consumer_rabbitmq	Listening to a RabbitMQ exchange.	
restful_http	Connection uses restful HTTP.	
Security stereotype	*Semantic*	*Security objective*
auth_provider	Authorization and/or authentication information sent on flow.	Authorization
authenticated_request	Flow between internal services on which requests are authenticated.	Authentication
circuit_breaker_link	Flow guarded by a circuit breaker.	Availability
load_balanced_link	Load balanced flow.	Availability
plaintext_authentication	Plaintext password and/or username found in code.	Authentication
plaintext_credentials_link	Flow over which plaintext credentials are passed.	Confidentiality
External entity		
Stereotype	*Semantic*	
external_database	Database for which the source code is not accessible.	
external_website	External website.	
github_repository	GitHub Repository holding configuration information.	
mail_server	External mail server.	
user	Represents the human user of the application.	
Security stereotype	*Semantic*	*Security objective*
entrypoint	External entity from which information enters the system.	Authorization
exitpoint	External entity to which information is sent.	Confidentiality
logging_server	Central storage for log messages.	Availability, non-repudiation
plaintext_credentials	Plaintext password and/or username found in code.	Confidentiality
tokenstore	Datastore for access tokens.	Authorization

Table 3 lists the resulting set of applications. In selecting the apps, we applied the following exclusion criteria:

EC1 Programming language is not Java
EC2 Natural language of documentation is not English
EC3 System is a framework instead of a standalone application
EC4 Application uses a technology that is not used by any other application
EC5 Application is too big or complex to be manually inspected with the resources at hand in the context of this paper

We note the limitations these criteria introduce. The last two criteria are clear cases of convenience sampling and represent a threat to validity. We introduce them nonetheless, because of the human effort needed to manually create DFDs for the applications to serve as groundtruth in our evaluation. Such effort would be infeasible for larger applications in this context. However, we note that the exclusion criterion related to the technology (EC4) only applied to two applications (one focusing on deployment with Travis, the other on integration of a Twitter API). An example for an application excluded because of EC5 is *Train Ticket*,[8] an application that consists of more than 40 microservices plus databases and over 300.000 lines of code. The time needed to analyze this application with the level of detail we include in our DFDs is vastly disproportional to the added value of having it in the dataset. Nonetheless, we hope to include a small number of such big applications in the dataset in the future to observe the prototype's performance. We do not expect an observable difference in the number of correctly and incorrectly extracted model items that is related to the size of the application. Thus, the criteria are seen as acceptable bias.

The DFDs were created manually by reading through the code and documentation. Prior to the analysis, we created a list of conceptual mappings between source code artifacts and an initial set of DFD items. These initial mappings were inspired by the architectural rules proposed by Bambhore Tukaram et al. (2022). Each code snippet indicating an item to be added to the DFD was turned into its corresponding model item. During code inspection, we extended the stereotypes and mappings as we encountered properties that we judged to be of interest for assessing the applications (from a security perspective) until we did not find any new ones anymore. By re-iterating the inspection of already created DFDs, we achieve consistency across the dataset. The inspection was performed by the first author, resulting in the extraction of 1989 model items.[9] We selected a subset of 200 model items following proportional stratified random sampling, corresponding to about 10% of the whole dataset. Each element in the sampled subset has been checked by three further researchers independently.

5.2. Dataset

The dataset consists of 17 DFDs of open-source applications on GitHub. Each DFD model includes the application's structure enriched with extensive (security and other) annotations and provides full traceability from model to code, i.e. each item's evidence in the code is tracked by means of its file and line number. The models are provided in JSON, PNG, and CodeableModels format. The traceability for each model item is presented in a JSON file for each model.

Listing 1: Example for node in JSON representation.

```
{
    "name": "notification_service",
    "stereotypes": [
```

[8] https://github.com/FudanSELab/train-ticket.

[9] We remind the reader that model items include (i) nodes (identified by their name), (ii) information flows (identified by the sender/receiver-pair), (iii) stereotypes, and (iv) tagged values (identified by key/value-pair).

Table 3
Applications used for validation with information on size and popularity.
(S = Services, E = External entities, I = Information flows, A = Annotations, Tot. = Total items).

ID	Application (GitHub handle)	S	E	I	A	Tot.	LoC	Technologies	Stars	Forks
Reference set										
1	apssouza22/java-microservice	13	2	34	100	149	5283	Docker, Docker Compose, Eureka, Hystrix, Kafka, Logstash, Maven, Nginx, Spring Admin, Spring Config, Spring OAuth, ZooKeeper	299	220
2	Callistaenterprise/blog-microservices	15	2	32	126	175	2786	Docker Compose, Elasticsearch, Eureka, Gradle, Hystrix, Kibana, Logstash, RabbitMQ, Ribbon, Spring Config, Spring OAuth, Turbine, Zipkin, Zuul	399	308
3	fernandoabcampos/spring-netflix-oss-microservices/	9	2	24	75	110	1644	Docker, Docker Compose, Eureka, Hystrix, Maven, RabbitMQ, Spring Config, Turbine, Zuul	10	11
4	georgwittberger/apache-spring-boot-microservice-example	4	1	6	18	29	604	Apache httpd, Maven	7	9
5	mudigal-technologies/microservices-sample	14	1	34	115	164	14527	Consul, Docker, Docker Compose, Elasticsearch, Kibana, Logstash, Nginx, RabbitMQ, Weave Scope, Zuul	294	311
6	spring-petclinic/spring-petclinic-microservices	10	2	28	89	129	3990	Docker, Docker Compose, Eureka, Grafana, Hystrix, Maven, Prometheus, Ribbon, Spring Admin, Spring Config, Spring Gateway, Zipkin,	1195	1543
7	sqshq/piggymetrics	14	3	37	192	246	9977	Docker, Docker Compose, Eureka, Hystrix, Maven, RabbitMQ, Ribbon, Spring Config, Spring OAuth, Turbine, Zuul	11862	5697
	Average reference set	11	2	28	102	143	5544		1995	1157
Validation set										
ID	*Application (GitHub handle)*	*S*	*E*	*I*	*A*	*Tot.*	*LoC*	*Technologies*	*Stars*	*Forks*
8	anilallewar/microservices-basics-spring-boot	10	2	29	108	149	4245	Docker, Docker Compose, Eureka, Gradle, Hystrix, Ribbon, Spring Config, Spring OAuth, Turbine, Zipkin, Zuul	645	410
9	ewolff/microservice	6	1	13	46	66	3117	Eureka, Hystrix, Maven, Ribbon, Turbine, Zuul	651	328
10	ewolff/microservice-kafka	7	1	12	56	76	2979	Apache httpd, Docker, Docker Compose, Kafka, ZooKeeper	510	272
11	jferrater/tap-and-eat-microservices	8	1	16	63	88	1484	Docker, Docker Compose, Eureka, Hystrix, Maven, Spring Config	6	4
12	koushikkothagal/spring-boot-microservices-workshop	4	1	6	26	37	638	Eureka, Maven	628	1012
13	mdeket/spring-cloud-movie-recommendation	6	5	18	93	122	1800	Eureka, Hystrix, Maven, Ribbon, Spring Config, Zipkin, Zuul	13	11
14	piomin/sample-spring-oauth2-microservices	5	3	13	78	99	1028	Eureka, Ribbon, Spring OAuth, Zuul	119	139
15	shabbirdwd53/springboot-microservice	7	2	18	65	92	879	Eureka, Hystrix, Maven, Spring Config, Spring Gateway, Zipkin	243	521
16	rohitghatol/spring-boot-microservices	8	3	26	100	137	2328	Docker Compose, Eureka, Gradle, Hystrix, Ribbon, Spring Config, Spring OAuth, Zuul	1667	906
17	yidongnan/spring-cloud-netflix-example	9	1	30	81	121	1182	Docker, Docker Compose, Eureka, Gradle, Hystrix, RabbitMQ, Ribbon, Spring Admin, Spring Config, Turbine, Zipkin, Zuul	782	371
	Average validation set	7	2	18	72	99	1968		526	397

```
        "internal",
        "local_logging",
        "resource_server"
    ],
    "tagged_values": {
        "Port": 8000
    }
}
```

Listing 1 presents an example excerpt of the JSON representation. The excerpt shows a node in the DFD (corresponding to a node in Fig. 1).

Listing 2: Example for node traceability. URL omitted for better readability.

```
"notification-service": {
        "file": "*URL*",
        "line": 3,
        "span": "(10:30)",
        "sub_items": {
            "Port": {
                "file": "*URL*",
                "line": 13,
                "span": "(8:12)"
            },
```

Listing 2 shows an example of the traceability information. The excerpt points to the place in the source code that proves the existence of the model item shown in Listing 1.

Table 3 lists the information about the applications in the dataset. The lines of code (LoC) show a good coverage of small- to medium-sized applications. The number of services, external entities, information flows, and annotations indicate the size of the corresponding DFD and have a good variation in size and density of annotations. The list of technologies gives an indication of the extraction complexity, i.e. how many technology-specific extractors are needed for the analysis. Finally, the numbers of stars and forks on GitHub are given to indicate the applications' popularity. Although these numbers do not give information about the code's quality, they do suggest that the majority of the applications in the dataset have a decent up to very good acceptance in the community. The parameters suggest that our dataset is a good

Table 4
Raw evaluation results. S = Services, E = External entities, I = Information flows, A = Annotations.

	S			E			I			A			Overall		
App	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
1	13			1	1	1	22	17	12	71	27	29	107	45	42
2	15	1		1		1	19		13	89	4	37	124	5	51
3	9			2			21	2	3	61	2	14	93	4	17
4	4			1			5		1	17		1	27		2
5	14			1			34			107	6	8	156	6	8
6	10			2			27	1	1	80	3	9	119	4	10
7	14			3			34		3	173		19	224		22
Sum 1–7	79	1	0	11	1	2	162	20	33	598	42	117	850	64	152
8	10	1		2	1		18	1	11	67	20	41	97	23	52
9	6			1			12	1	1	43	1	3	62	2	4
10	7			1			12	3		52	4	4	72	7	4
11	8			1			16			59	3	4	84	3	4
12	4					1	5		1	23		3	32		5
13	6			4		1	17		1	84	6	9	111	6	11
14	5			3			12	1	1	62	5	16	82	6	17
15	7			2			16		2	57		8	82		10
16	8			3	1		18	1	8	75	12	25	104	14	33
17	9			1			26		4	74	2	7	110	2	11
Sum 8–17	70	1	0	18	2	2	152	7	29	596	53	120	836	63	151
Sum all	149	2	0	29	3	4	314	27	62	1194	95	237	1686	127	303

and diverse representation of popular small- to medium-sized microservice applications as found publicly on GitHub.

5.3. Experimental validation procedure

The technique has been evaluated by executing the final prototype on each application in the dataset and analyzing the results. We remark that the authors had access to seven applications in the dataset while developing and tuning the Code2DFD prototype. In Table 3, these applications are listed under the *Reference Set* section (applications 1–7). The other 10 applications listed under the *Validation Set* section (applications 8–17) have not been analyzed until after the completion of the prototype. For completeness of information, we report the results for all 17 applications but only consider the validation set when answering the research questions.

We compare the automatically extracted DFDs to the manually curated ones and quantify the approach's performance by counting the items that are correctly and incorrectly identified by the tool (true positives TP and false positives FP), as well as those that go unnoticed (false negatives FN). Accordingly, we report the precision and the recall achieved by the tool for each individual application, as well as an aggregate over the reference set, the validation sets, and the whole dataset. We also report precision and recall in two sub-categories: (i) for what concerns only the structural core elements of a DFD and (ii) the security annotations (see TablesTables 1 and 2). Although a key feature of our approach is the extraction of the security-enriching annotations, the technique could also be used outside of security, e.g., by only extracting the core DFD (i.e., the structure of a system consisting of all nodes and connections between them) for program comprehension or security assessment techniques. Incidentally, this is the reason why we have formulated RQ2. The category of core items without annotations thus comprises services, external entities, and information flows. The second category (security annotations) is of particular interest for this paper as this contribution is completely novel (and challenging) with respect to the related work. For this, we formulated RQ3.

6. Experimental results

6.1. Quality of the extracted DFD models

We first look at the approach's performance in terms of correctly extracting dataflow diagram items. Table 4 shows the raw results produced by the prototype for the 17 applications. As a first observation, the detection seems to have worked well in all cases. All applications show high TP numbers compared to the FN for all groups, meaning that no grave mistakes seem to have occurred. For the false positives (FP), most cells in the table even have no occurrences at all, and only for the annotations can we see some higher numbers (27 for application 1, 20 for application 8). For some applications (4, 7, 12, and 15), no FP are detected at all. We see that only five incorrect nodes (services + external entities) are detected, while the FP for information flows and annotations are higher. This was expected, given the challenging nature of these model items.

Table 5 shows the computed precision and recall, allowing us to put the raw numbers into perspective. A first observation is a very good result with an overall average precision of 93% and recall of 85%. As mentioned before, we observe a high precision and recall for the nodes but slightly lower numbers for the information flows and, particularly, the annotations. This is explicitly evident in the values for the extraction of core items without annotations, with a precision of 94% and recall of 88%. With 87%, the security annotations show a lower recall than the other groups, where 71% is the worst observed recall (application 1). Looking at the results for individual applications, we notice that some achieve a precision or recall of only around 70%. The lowest overall value is a 65% recall for application 8.

Table 5 (Applications 8 – 17) allows us to provide the following answers to RQ1, RQ2, and RQ3:

RQ1 We observe a precision of 0.93 and recall of 0.85 for the extraction of all DFD items.

RQ2 We observe a precision of 0.96 and recall of 0.89 for the extraction of DFD core items.

RQ3 We observe a precision of 0.88 and recall of 0.75 for the extraction of the security-relevant annotations.

In summary, the results show that our approach is a promising solution for reliably extracting dataflow diagrams. In Section 7, we elaborate further on how to interpret the results.

Table 5
Precision (P) and recall (R) for each item group. S = Services, E = External entities, I = Information flows, A = Annotations, Core items = S + E + I.

	S		**E**		**I**		**A**		**Overall**		**Core items**		**Security annot.**	
App	P	R	P	R	P	R	P	R	P	R	P	R	P	R
1	1	1	0.5	0.5	0.56	0.65	0.72	0.71	0.7	0.72	0.67	0.73	0.78	0.47
2	0.94	1	1	0.5	1	0.59	0.96	0.71	0.96	0.71	0.97	0.71	1	0.73
3	1	1	1	1	0.91	0.88	0.97	0.81	0.96	0.85	0.94	0.91	1	0.67
4	1	1	1	1	1	0.83	1	0.94	1	0.93	1	0.91	1	1
5	1	1	1	1	1	1	0.95	0.93	0.96	0.95	1	1	1	1
6	1	1	1	1	0.96	0.96	0.96	0.9	0.97	0.92	0.98	0.98	0.88	0.93
7	1	1	1	1	1	0.92	1	0.9	1	0.91	1	0.94	1	0.92
Avg. 1–7	0.99	1	0.92	0.85	0.89	0.83	0.93	0.84	**0.93**	**0.85**	0.92	0.88	0.97	0.83
8	0.91	1	0.67	1	0.95	0.62	0.77	0.62	0.81	0.65	0.91	0.73	1	0.88
9	1	1	1	1	0.92	0.92	0.98	0.93	0.97	0.94	0.95	0.95	1	0.89
10	1	1	1	1	0.8	1	0.93	0.93	0.91	0.95	0.87	1	1	0.89
11	1	1	1	1	1	1	0.95	0.94	0.97	0.95	1	1	1	1
12	1	1			1	0.83	1	0.88	1	0.86	1	0.82	0.91	0.87
13	1	1	1	0.8	1	0.94	0.93	0.9	0.95	0.91	1	0.93	0.84	0.72
14	1	1	1	1	0.92	0.92	0.93	0.79	0.93	0.83	0.95	0.95	1	0.73
15	1	1	1	1	1	0.89	1	0.88	1	0.89	1	0.93	0.83	0.61
16	1	1	0.75	1	0.95	0.69	0.86	0.75	0.88	0.76	0.94	0.78	0.93	1
17	1	1	1	1	1	0.87	0.97	0.91	0.98	0.91	1	0.9	0.93	0.79
Avg. 8–17	0.99	1	0.9	0.9	0.96	0.84	0.92	0.83	**0.93**	**0.85**	0.96	0.89	0.88	0.75
Avg. all	0.99	1	0.91	0.88	0.92	0.84	0.93	0.83	**0.93**	**0.85**	0.94	0.88	0.93	0.79

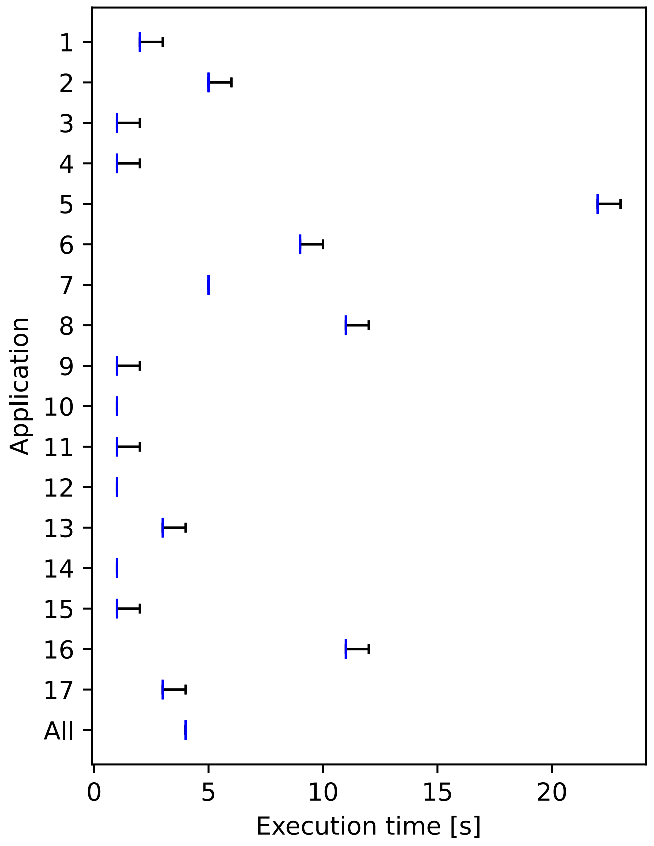

Fig. 2. Execution times on all applications in seconds, average over five execution runs. Lines are min to max, blue markers are average over the five runs. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

6.2. Time performance of model extraction

Since a possible use-case of this approach is the adoption in CI/CD pipelines, delta security evaluation and evaluation of apps, and other time-sensitive scenarios, the prototype's performance is of interest as well. We performed the DFD extraction five times on each application and took the average of the complete execution time, excluding the cloning of the repositories from GitHub. The experiments were performed on a MacBookPro with a 2 GHz processor and 16 GB RAM.

The results are shown in Fig. 2. Per application, the extraction times vary between 1 and 22 s, with an overall average of 4 s. Most apps lie under the 5 s mark. The standard deviations for all applications is under 1 s, indicating a very stable extraction time. When comparing Fig. 2 and Table 3, the numbers show no obvious correlation between the execution time and size of the corresponding application or model, number of used technologies, or other parameter. An analysis of what factors influence the execution time is left open for future work.

Fig. 2 allows us to provide the following answer to RQ4:

RQ4 We observe a maximum model extraction time of 22 s, with an overall average of only 4 s.

Overall, the execution times illustrate that our (fully automated) approach is applicable in fast-paced development pipelines and other time-sensitive scenarios.

6.3. Threats to validity

Some limitations in our evaluation stem from the lack of available data and the resulting creation of a manually curated dataset. Extracting the DFDs and their model elements from the code of the applications has been done manually by one researcher and checked by three others. Although we are quite confident that the models and the traceability information are correct, bias and material errors are always a possibility.

We are also aware that the dataset is of limited size (17 applications), the applications have been selected by means of convenience sampling, and that a few applications are not particularly complex. Therefore, the results we report might not entirely transfer to other real-world applications. In this respect, the lack of data for validation is a well-known issue in the model-based security community. While some papers do present DFD models extracted from medium-to-large size application, they are not useful in our case as (i) they are not related to microservices and (ii) they do not provide the traceability to code information.

Finally, our validation results might be specific to the technologies that we support in our extraction framework. While the stereotypes we consider (as listed in Tables 1 and 2) are meaningful and comprehensive for security, they evidently do not cover all technological and security features one could encounter in microservices written in Java.

7. Discussion of results

In this section, we further elaborate on the causes for the false positives and false negatives encountered in the evaluation as shown in Table 4 and talk about the corresponding precision and recall presented in Table 5. We also elaborate on which items these errors apply to, in order to better assess what results can be expected when employing our approach. We further put the results into context regarding the usability of our approach for security analysis use-cases.

Nodes. The correct detection of all but two nodes is assuring but not surprising, since they are mostly detected by the simplest search technique, parsing. Alternatively, the nodes are found by references to an external entity. The group of services does not have any false negatives, but there are four applications for which an external entity is not detected. Looking at the causes, these failing detections are acceptable for the state of the prototype. For example, one is an undetected database, for which only an ambiguous line of code and a comment in the `README` file are evidence. A failing detection of this item can be regarded as acceptable error or even as out-of-scope of a static analysis approach, definitely out-of-scope of our approach. Similar findings hold for the false positives. In two cases, the tool creates two DFD items for the same database because two different names are used for it in the application, in another case a false positive node is a GitHub repository which is detected as an external entity because it is referenced in a configuration file as external configuration repository, but not present in the ground truth because it is the same repository where the complete code lies. It is debatable if this should even count as a false positive, but to keep the results conservative, we count it as such.

Information flows. A precision of 92% and a recall of 84% for this group shows that the techniques are applicable for complex detections, such as that of message broker connections, where outgoing and incoming endpoints have to be detected individually and matched while taking the syntactical rules of queues, exchanges, and routing keys into account. We see this as affirming result for our approach.

The reasons for some false positives and false negatives lie in the direction of the information flows. For the ground truth, we decided on the direction based on the source of the data asset that is exchanged. In some cases, the prototype detects the flow but with the wrong direction, which leads to both a false positive and a false negative. This highlights the need to improve the corresponding extractors. Other false positives happen because the prototype detects connections of services to themselves. For example, a monitoring server can have the scope 'all services'. A flow from that service to itself will be detected in this case. However, there are plain errors as well, where the detection failed due to inaccurate extractors. The only case for which the number of false positive information flows is concerningly high is for application 1. Here, the above mentioned problem of flows being detected with the wrong direction happened multiple times in addition to a faulty implementation of the technology-specific extractor for Apache ZooKeeper. An assessment of the false positives and false negatives of all applications (especially applications 2, 8, and 16 with their comparatively low recall) yielded, that the majority of the errors should be addressable with minor adjustments to the existing extractors or by adding some. We believe that achieving better results while not overfitting to a dataset and keeping generality is possible with a more sophisticated implementation of our approach.

Annotations. The annotations are an important group of items to evaluate our approach since they are a distinguishing feature compared to related approaches and many of them carry the security-specific information. Results of 93% precision and 83% recall are thus seen as a very good and promising outcome. False positives often stem from the reduced scope of implemented mechanisms, which is often hard to grasp with the technology-specific extractors. For example, the extractor for circuit breakers marks all outgoing flows from a service it detects as being guarded by the circuit breaker, which is not always correct. Other false positives are due to legacy code found in some applications, which are detected by the extractors but do not have any effect in the application. As for the information flows, the majority of undetected annotations is due to the implementation of our prototype, with flaws in the extractors or simply missing extractors for some technology solutions. We did not encounter any annotations for which we believe a detection with our approach to be impossible.

Security annotations. To more specifically evaluate the performance concerning the security analysis, we look at the precision and recall for the subset of security-relevant annotations. Compared to the other cases, the recall is slightly lower (79%). The reason lies in the detection complexity of many of the security annotations. Especially authentication and authorization are concerns that require detailed configurations at numerous places in the code, which makes the detection of the corresponding stereotypes particularly challenging. However, the numbers also show that these more complex annotations can still be detected with our approach, although more work in this area could be needed in the future.

Validation vs. reference set. As described in 5.3, we used the reference set for testing and tuning during the development of Code2DFD. Accordingly, we expected a degree of 'over-fitting' in the reference set accompanied by lower performance results in the validation set. However, the drop in performance did not happen. One explanation lies in the relatively small dataset, for which the overall results can be swayed by the outcome of single applications. Also, a larger dataset might cover applications where the code displays more variations in the way the technologies are used (hence causing more false negatives). For now, we report the good results even for unseen applications and plan a more extensive evaluation for future work.

8. Comparison to other approaches

To provide context for our prototype's performance, we compare its results with those of other related approaches. We identified available and executable tools of the approaches for architecture reconstruction that we present later in Section 9 (listed in Table 7). Two of these provide implementations of their approach that we were able to obtain and execute. The remaining approaches in Table 7 either do not present and make available an implementation, or the approach is dynamic and thus not applicable for this comparison. Moreover, in those publications where a comparable implementation is absent, there are no evaluation results reported that could be presented here as a comparison. We analyzed the two identified tools on all 17 microservice applications in our dataset of microservice applications and evaluate the outcomes. The results reported in this subsection are thus verified by us on the same dataset of applications that we evaluated our own tool on.

The two identified tools do not consider the extraction of (security) annotations. Therefore, we only compare the architecture reconstruction. We were unable to find any relevant work that focuses on detection of security features or other comparable system properties.

Table 6
Precision and recall in architecture reconstruction for the related approaches. S = Services, E = External entities, I = Information flows.

Approach	Analyzed applications	S		E		I		Overall	
		P	R	P	R	P	R	P	R
Prophet (Bushong et al., 2021, 2022)	5/17[a]	0.59	0.22	0	0	0.75[a]	0.07[a]	**0.61**	**0.15**
MicroDepGraph (Rahman et al., 2019)	10	1	0.86	1	0.13	0.98	0.51	**0.99**	**0.58**
Code2DFD	17	0.99	1	0.91	0.88	0.92	0.84	**0.94**	**0.88**

[a]The tool produced a list of microservices for 17 apps, but communication diagrams only for five of them. Results for **I** are only for these five.

Table 7
Extraction techniques employed in the related work for architecture reconstruction.

ID	Approach	Services	Information flows
1	Alshuqayran et al. (2018)	Static parser	Inject Zipkin tracing
2	Bushong et al. (2021, 2022)	Static	Static detection of API endpoints
3	Granchelli et al. (2017b,a)	Static parser	Injected TcpDump, existing service discovery
4	Kleehaus et al. (2018)	Existing Consul or Eureka	Google Dapper
5	Ma et al. (2019)	Existing Eureka	Java annotations
6	Mayer and Weinreich (2018)	Static parser, Swagger	Own interceptor modules
7	Rademacher et al. (2020)	Static parser	Java annotations
8	Rahman et al. (2019)	Static parser	Spring annotations
9	Soldani et al. (2021)	Static parser	Kubernetes network traffic
10	Walker et al. (2021)	Static (source and byte code)	Static (source and byte code)

8.1. Compared tools

The first tool, *Prophet* (Bushong et al., 2021, 2022), extracts microservices and communication links between them by leveraging enterprise standards, i.e. knowledge about standard components and constructs created by popular development frameworks. According to the authors, the semantic purpose of relevant components in code can be identified via naming conventions and other metadata, which leads to the detection of microservices. Communication links between the services are detected by identifying API endpoints and API calls to other services in the same way. The tool produced complete results for five of the 17 applications. For all others, it only extracted the list of services it detected in the applications. In the calculation of precision and recall for the information flows, we only consider these five successful extractions to provide a fair comparison.

The second tool, *MicroDepGraph* (Rahman et al., 2019), parses Docker Compose and Java files to detect microservices and communication links between them. The tool parses a Docker Compose file (if one exists) to extract the list of microservices and communication links (from specified service dependencies). Additionally, direct API calls between services are extracted from Java source code by checking for Spring Boot annotations that define API endpoints and for requests to these endpoints from other services. The reliance on the use of Docker Compose led to a limitation of the analysis to 10 of our 17 applications. We use these 10 to calculate the results. In addition, minor changes to the code of the analyzed applications were needed, because the tool does not consider all legal syntaxes of Docker Compose and some applications had to be adjusted to be correctly parsed by the tool.

8.2. Comparison results

Table 6 shows the results of Code2DFD and the two compared tools for the extraction of the microservices' architecture (i.e., services, external entities, and information flows; annotations omitted since the related approaches do not consider them).

The first tool achieves an overall precision of 0.61 and recall of 0.15. The tool produced many false positive services from which the names correspond to directories in the applications' repositories. The precision for just the information flows (calculated on five applications for which a communication diagram was successfully created by the tool) is slightly higher, but the recall is 0.07. Overall, this approach does not offer a reliable architecture extraction for the applications in our dataset.

The second tool produces an overall precision of 0.99 and recall of 0.58. The tool extracts most information via simple parsing of Docker Compose files. These files are usually a reliable source for architecture reconstruction, but contain limited information. This leads to a high precision because false positives rarely happen in this process, but also to a rather low recall.

Compared to our Code2DFD, the second approach achieves a slightly higher precision (0.99 against 0.94). However, while we observe a recall of 0.88 in Code2DFD, the second approach achieves a lower recall of 0.58. The first approach produces significantly lower results than Code2DFD on our dataset. In conclusion, our approach proves to be competitive with the two compared approaches regarding the architecture reconstruction. It shows a slightly lower precision than approach 1 and outperforms both approaches significantly concerning the recall.

9. Related work

With its extraction of extensively annotated DFDs that can be used to analyze software systems' security, our approach combines different scientific research fields. The extraction of the core DFDs (i.e. their nodes and edges) falls under the field of architecture reconstruction, the extraction of annotations relates to feature identification, and the enabling of a security analysis with the DFDs has related work in model-based security. In the following, a comparison to the literature is presented separately for these three fields.

Architecture reconstruction. Multiple approaches have been presented in literature that employ different combinations of static and dynamic analysis to recover the basic architecture of microservice applications. Table 7 lists those related to our work and shows the techniques used to extract the services and information flows, thus providing an overview of the differences to our approach. Not all of the presented approaches are automated, but they at least claim to be automatable or mention it as future work. A detailed description is given in the following (mentioned approach IDs correspond to those in the table).

Most approaches retrieve the list of services in the same way we do in our approach, by parsing Docker, Docker Compose, Maven, or Kubernetes manifest files (approaches 1 Alshuqayran et al., 2018, 2 Bushong et al., 2021, 2022, 3 Granchelli et al., 2017a,b, 6 Mayer and Weinreich, 2018, 7 Rademacher et al., 2020, 8 Rahman et al., 2019, and 9 Soldani et al., 2021). Approach 10 (Walker et al., 2021) works statically as well, but analyses control-flow graphs and program dependency graphs of the applications. Approaches 4 (Kleehaus et al., 2018) and 5 (Ma et al., 2019) detect microservices via service discovery technologies at run-time. For this, they query service discovery services to retrieve the list of currently deployed instances of microservices. The two approaches are limited to implementations using *Consul* (only approach 4) or *Eureka* (both) that already exist in the analyzed applications, thus reducing the scope to such applications where this holds true.

In the related work, information flows are either detected statically or dynamically with different tracing implementations. Approaches 2, 5, 7, and 8 leverage Java annotations to statically detect the flows, following one of the detection techniques we use in Code2DFD. We note the difference in scope between these approaches and ours. Apart from further detecting information flows via methods not connected to annotations (i.e. direct API calls with e.g. *RestTemplate*), we also consider more annotations in the detection. From the approaches using dynamic analysis to retrieve information flows, approach 3 uses existing service discovery services and an injected monitoring tool. Approach 1 injects a *Zipkin* tracing server, a technology that is used by some of the applications in our dataset as well, while approach 4 injects *Dapper*, another tracing technology. By letting all existing services send tracing data to the injected service, the information flows can be observed. Approach 6 also injects components to trace information, but their interceptor modules are developed specifically for this purpose. Finally, approach 9 specializes on deployment via Kubernetes and observes the network traffic to obtain communication links between services. All these dynamic approaches have the drawback that they require changes to be made to the existing application that is to be analyzed. Further, they can only detect information flows that get executed during the analysis, as all dynamic approaches do.

It is evident, that there are similarities between the related work and our approach considering only the extraction of the structure of microservice applications. Approaches 2, 7 and 8 all work fully statically and follow the same base idea for retrieving services and information flows. However, apart from the larger scope and lightweight static analysis, another clear distinguishing feature of our work is the extraction of application properties beyond the basic structure (and with focus on security), in order to give a more detailed view of the software system. Approach 1 is the only other approach that is also detecting load balancers, gateways, service discovery, circuit breakers, and access tokens. These features, however, are a small subset of what we extract, especially from a security standpoint (compare Tables 1 and 2). Finally, and most notably, all approaches in Table 7 require human input and, most of them, a heavyweight dynamic analysis orchestration, both of which we avoid.

Concerning the structure of the proposed technique, Mosser et al. (2020) present an approach that follows a similar architecture as Code2DFD. Their *Animaxander* consists of probes that extract a map of the microservice application, showing parallels to our technology-specific extractors. However, it relies on human input to a large extent and does not focus on security.

Feature identification. No related work addressing automated (security) feature identification specifically for microservice applications could be found in the literature. In fact, Assunção et al. identified this as a challenge for future work on microservices (Assunção et al., 2020). However, there are approaches that consider feature identification in other contexts without a focus on microservices. Many approaches for the identification and traceability of features and their location in source code have been made in the fields of program comprehension and software maintenance and evolution (Dit et al., 2013; Rubin and Chechik, 2013).

Many static approaches for feature identification rely on the existence of some source of information that is created by the developers beyond the functional code. For example, many approaches leverage annotations in the source code that are added specifically for the purpose of tracing and visualizing features (Abukwaik et al., 2018; Andam et al., 2017; Bergel et al., 2021; Entekhabi et al., 2019; Martinson et al., 2021; Seiler and Paech, 2017). Since the code has to be annotated first (either by the developers during implementation or retroactively), these approaches do not solve the problem of automatically detecting the features on unmodified code or code that was not developed with subsequent feature identification in mind.

Other approaches omit this dependence on dedicated annotations by leveraging already existing information in code. They use natural language processing, machine learning, pattern matching, or other techniques to identify features. For this purpose, identifiers, comments in the code, or documentations are commonly used as input (Burger and Grüner, 2018; Eaddy et al., 2008; Marcus et al., 2004; Savage et al., 2010; Zhao et al., 2004). These approaches suffer from a reliance on the developers choosing meaningful identifiers and writing descriptive comments and documentation, otherwise they cease to work. We avoid this in our approach with the help of the iterative keyword search and codification of domain-knowledge about technologies via the technology-specific extractors.

We conclude, that our approach is novel in regard to identifying (security) features in microservice applications and that with this it addresses one of the challenges for microservices posed by Assunção et al. (2020). Further, the presented related approaches rely on information that is chosen by developers and as such not completely reliable, which we avoid.

Model-based security. Some approaches exist that use DFDs to perform security analysis at model level. Abi-Antoun et al. (2007) compare a security-annotated DFD extracted from the implementation code of a system against one that represents the design specifications. By checking conformance between the two, architectural drift can be detected. The approach further enables a security analysis by formulating security properties based on the annotations in the DFDs that can be checked.

Berger et al. (2016) introduce eDFDs (short for extended DFDs), an extension of DFDs that also captures properties like trust boundaries and better tracking of data flows. The authors collect threats in a knowledge base and use the Microsoft Threat Modeling technique to find architectural weaknesses. Tuma et al. (2019) introduce SecDFDs (short for Security DFDs), an extension to DFDs that includes further information like security policies, assets' importance, and security labels (corresponding to private or public information). Based on security contracts defining how security labels change for specific operations, the assets are tracked through the SecDFD and violations of security policies can be detected. In both approaches, the user needs to understand the analyzed application well, as eDFDs and SecDFDs have to be created manually.

Sion et al. (2018b) present an approach in which architectural patterns can be formulated to capture implemented security or privacy countermeasures. The patterns are expressed by referring to the elements of corresponding DFDs. When performing a security analysis such as STRIDE with the DFDs as input, the patterns support the analysis by removing threats from consideration that are already mitigated by a security solution. Our extracted DFDs might be a good fit for such architectural patterns, as their extensive annotations should allow the formulation of rich and expressive patterns.

Instead of extending DFDs with additional information of some kind as the approaches above do, Faily et al. (2020) propose to put them into context of other usability and requirement models. They argue, that the simplicity of basic DFDs should be preserved and that alignment with other models should be strengthened. The authors illustrate this idea on the example of taint analysis.

Berger et al. (2013) proposed ArchSec, a solution for automatically extracting the security architecture of Java applications. ArchSec only works for J2EE and Android applications, and not for microservices. Furthermore, the tool expects that the system's decomposition into the top level components (and the segmentation of the codebase into the corresponding components) is provided as input by a human.

In conclusion, to the best of our knowledge no approach has been proposed in the literature that allows a fully automated extraction of dataflow diagrams or similar model representations of microservices which are enriched with additional, extensive (security) annotations and thus allow an assessment of the corresponding application's security. Finally, we further emphasize the extent of our evaluation with the comparably large dataset, which is unrivaled by the mentioned related approaches.

10. Conclusion

This paper presents an approach for extracting dataflow diagrams enriched with extensive annotations from microservices' source code. The approach analyzes applications statically and interprets the code (source code, configuration files, scripts, and so on) as text. The implementation is focused on Java applications and can be applied to systems that use the technologies for which an extraction implementation is provided. However, we discussed how the approach can be extended to include other languages and technologies. In our validation results, we observed a very satisfying performance, with a precision above 90% and recall of 85%, which outperforms two related approach in architecture reconstruction that we executed on the same dataset.

In future work, we plan on extending our prototype to support polyglot applications. Further, we want to increase the size of our dataset with additional applications, as the lack of data is an issue in the model-based security community.

CRediT authorship contribution statement

Simon Schneider: Conceptualization, Methodology, Software, Validation, Investigation, Data curation, Writing – original draft, Writing – review & editing. **Riccardo Scandariato:** Conceptualization, Methodology, Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A dataset of 17 dataflow diagrams (DFDs) and a prototype for the automatic extraction of DFDs is made available alongside this publication. The repositories are referenced in the paper

Acknowledgments

The authors thank Catherine Tony, Quang Cuong Bui, and Torge Hinrichs for their support in validating the dataset.

This work was partly funded by the European Union's Horizon 2020 programme under grant agreement No. 952647 (AssureMOSS).

References

Abi-Antoun, M., Barnes, J.M., 2010. Analyzing security architectures. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10. Association for Computing Machinery, New York, NY, USA, pp. 3–12. http://dx.doi.org/10.1145/1858996.1859001.

Abi-Antoun, M., Wang, D., Torr, P., 2007. Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security, ASE '07. Association for Computing Machinery, New York, NY, USA, pp. 393–396. http://dx.doi.org/10.1145/1321631.1321692.

Abukwaik, H., Burger, A., Andam, B.K., Berger, T., 2018. Semi-automated feature traceability with embedded annotations. In: 2018 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 529–533. http://dx.doi.org/10.1109/ICSME.2018.00049.

Alshuqayran, N., Ali, N., Evans, R., 2018. Towards micro service architecture recovery: An empirical study. In: 2018 IEEE International Conference on Software Architecture. ICSA, pp. 47–4709. http://dx.doi.org/10.1109/ICSA.2018.00014.

Andam, B., Burger, A., Berger, T., Chaudron, M.R.V., 2017. Florida: Feature location dashboard for extracting and visualizing feature traces. In: Proceedings of the 11th International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '17. Association for Computing Machinery, New York, NY, USA, pp. 100–107. http://dx.doi.org/10.1145/3023956.3023967.

Assunção, W.K.G., Krüger, J., Mendonça, W.D.F., 2020. Variability management meets microservices: Six challenges of re-engineering microservice-based webshops. In: Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume a - Volume a, SPLC '20. Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3382025.3414942.

Bambhore Tukaram, A., Schneider, S., Díaz Ferreyra, N.E., Simhandl, G., Zdun, U., Scandariato, R., 2022. Towards a security benchmark for the architectural design of microservice applications. In: Proceedings of the 17th International Conference on Availability, Reliability and Security, ARES '22. Association for Computing Machinery, New York, NY, USA, pp. 1–7. http://dx.doi.org/10.1145/3538969.3543807.

Bergel, A., Ghzouli, R., Berger, T., Chaudron, M.R.V., 2021. Featurevista: Interactive feature visualization. In: Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume a, SPLC '21. Association for Computing Machinery, New York, NY, USA, pp. 196–201. http://dx.doi.org/10.1145/3461001.3471154.

Berger, B.J., Sohr, K., Koschke, R., 2013. Extracting and analyzing the implemented security architecture of business applications. In: 2013 17th European Conference on Software Maintenance and Reengineering. pp. 285–294. http://dx.doi.org/10.1109/CSMR.2013.37.

Berger, B., Sohr, K., Koschke, R., 2016. Automatically extracting threats from extended data flow diagrams. In: Engineering Secure Software and Systems, Vol. 9639. pp. 56–71. http://dx.doi.org/10.1007/978-3-319-30806-7_4.

Bernsmed, K., Cruzes, D., Jaatun, M., Iovan, M., 2021. Adopting threat modelling in agile software development projects. J. Syst. Softw. 183, 111090. http://dx.doi.org/10.1016/j.jss.2021.111090.

Bruza, P., van der Weide, T., 1989. The Semantics of Data Flow Diagrams. University of Nijmegen, Department of Informatics, Faculty of Mathematics and Informatics, Nijmegen TR.

Burger, A., Grüner, S., 2018. Finalist2: Feature identification, localization, and tracing tool. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 532–537. http://dx.doi.org/10.1109/SANER.2018.8330254.

Bushong, V., Das, D., Al Maruf, A., Cerny, T., 2021. Using static analysis to address microservice architecture reconstruction. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 1199–1201. http://dx.doi.org/10.1109/ASE51524.2021.9678749.

Bushong, V., Das, D., Černý, T., 2022. Reconstructing the holistic architecture of microservice systems using static analysis, 149–157. http://dx.doi.org/10.5220/0011032100003200.

Butler, S., Wermelinger, M., Yu, Y., Sharp, H., 2009. Relating identifier naming flaws and code quality: An empirical study. In: 2009 16th Working Conference on Reverse Engineering. pp. 31–35. http://dx.doi.org/10.1109/WCRE.2009.50.

Černý, T., Taibi, D., 2022. Static analysis tools in the era of cloud-native systems.

Chen, R., Li, S., Li, Z.E., 2017. From monolith to microservices: A dataflow-driven approach. In: 2017 24th Asia-Pacific Software Engineering Conference. APSEC, pp. 466–475. http://dx.doi.org/10.1109/APSEC.2017.53.

DeMarco, T., 1979. Structure Analysis and System Specification. Springer Berlin Heidelberg, http://dx.doi.org/10.1007/978-3-642-48354-7_9.

Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D., 2013. Feature location in source code: a taxonomy and survey. J. Softw.: Evol. Process 25 (1), 53–95. http://dx.doi.org/10.1002/smr.567, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.567.

Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., 2016. Microservices: Yesterday, Today, and Tomorrow. Springer International Publishing, pp. 195–216. http://dx.doi.org/10.1007/978-3-319-67425-4_12, Ch. 12.

Eaddy, M., Aho, A.V., Antoniol, G., Guéhéneuc, Y.-G., 2008. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In: 2008 16th IEEE International Conference on Program Comprehension. pp. 53–62. http://dx.doi.org/10.1109/ICPC.2008.39.

Entekhabi, S., Solback, A., Steghöfer, J.-P., Berger, T., 2019. Visualization of feature locations with the tool featuredashboard. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B, SPLC '19. Association for Computing Machinery, New York, NY, USA, pp. 1–4. http://dx.doi.org/10.1145/3307630.3342392.

Faily, S., Scandariato, R., Shostack, A., Sion, L., Ki-Aries, D., 2020. Contextualisation of data flow diagrams for security analysis. In: Eades, H., Gadyatskaya, O. (Eds.), Graphical Models for Security. Springer International Publishing, Cham, pp. 186–197.

Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L., Di Salle, A., 2017a. Towards recovering the software architecture of microservice-based systems. In: 2017 IEEE International Conference on Software Architecture Workshops. ICSAW, pp. 46–53. http://dx.doi.org/10.1109/ICSAW.2017.48.

Granchelli, G., Cardarelli, M., Francesco, P., Malavolta, I., Iovino, L., Di Salle, A., 2017b. Microart: A software architecture recovery tool for maintaining microservice-based systems. In: 2017 IEEE International Conference on Software Architecture Workshops. ICSAW, pp. 298–302. http://dx.doi.org/10.1109/ICSAW.2017.9.

Hannousse, A., Yahiouche, S., 2021. Securing microservices and microservice architectures: A systematic mapping study. Comp. Sci. Rev. 41, 100415. http://dx.doi.org/10.1016/j.cosrev.2021.100415.

Hernan, S., Lambert, S., Ostwald, T., Shostack, A., 2006. Threat modeling-uncover security design flaws using the stride approach. MSDN Mag. 68–75.

Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S., 2018. Microservices: The journey so far and challenges ahead. IEEE Softw. 35 (3), 24–35. http://dx.doi.org/10.1109/MS.2018.2141039.

JRebel, 2022. 2022 Java Developer Productivity Report. Tech. rep., JRebel, URL https://www.jrebel.com/resources/java-developer-productivity-report-2022.

Kleehaus, M., Uludag, Ö., Schäfer, P., Matthes, F., 2018. Microlyze: A framework for recovering the software architecture in microservice-based environments. In: Information Systems in the Big Data Era. Springer International Publishing, pp. 148–162. http://dx.doi.org/10.1007/978-3-319-92901-9_14.

Larsen, P., Plat, N., Toetenel, H., 1998. A formal semantics of data flow diagrams. Form. Asp. Comput. 6, http://dx.doi.org/10.1007/BF03259387.

Li, X., Chen, Y., Lin, Z., 2019a. Towards automated inter-service authorization for microservice applications. In: Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos, SIGCOMM Posters and Demos '19. Association for Computing Machinery, New York, NY, USA, pp. 3–5. http://dx.doi.org/10.1145/3342280.3342288.

Li, S., Zhang, H., Jia, Z., Li, Z., Zhang, C., Li, J., Gao, Q., Ge, J., Shan, Z., 2019b. A dataflow-driven approach to identifying microservices from monolithic applications. J. Syst. Softw. 157, 110380. http://dx.doi.org/10.1016/j.jss.2019.07.008.

Ma, S.-P., Liu, I.-H., Chen, C.-Y., Lin, J.-T., Hsueh, N.-L., 2019. Version-based microservice analysis, monitoring, and visualization. In: 2019 26th Asia-Pacific Software Engineering Conference. APSEC, pp. 165–172. http://dx.doi.org/10.1109/APSEC48747.2019.00031.

Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J., 2004. An information retrieval approach to concept location in source code. In: 11th Working Conference on Reverse Engineering. pp. 214–223. http://dx.doi.org/10.1109/WCRE.2004.10.

Márquez, G., Astudillo, H., 2018. Actual use of architectural patterns in microservices-based open source projects. In: 2018 25th Asia-Pacific Software Engineering Conference. APSEC, pp. 31–40. http://dx.doi.org/10.1109/APSEC.2018.00017.

Martinson, J., Jansson, H., Mukelabai, M., Berger, T., Bergel, A., Ho-Quang, T., 2021. Hans: Ide-based editing support for embedded feature annotations. In: Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B, SPLC '21. Association for Computing Machinery, New York, NY, USA, pp. 28–31. http://dx.doi.org/10.1145/3461002.3473072.

Mayer, B., Weinreich, R., 2018. An approach to extract the architecture of microservice-based software systems. In: 2018 IEEE Symposium on Service-Oriented System Engineering. SOSE, pp. 21–30. http://dx.doi.org/10.1109/SOSE.2018.00012.

Microsoft Corporation, 2016. Microsoft threat modeling tool 2016. URL https://www.microsoft.com/en-us/download/details.aspx?id=49168.

Mosser, S., Caissy, J.-P., Juroszek, F., Vouters, F., Moha, N., 2020. Charting microservices to support services' developers: The anaximander approach. In: Service-Oriented Computing. Springer International Publishing, pp. 36–44. http://dx.doi.org/10.1007/978-3-030-65310-1_3.

Pereira-Vale, A., Fernández, E., Monge, R., Astudillo, H., Márquez, G., 2021. Security in microservice-based systems: A multivocal literature review. Comput. Secur. 103, 25. http://dx.doi.org/10.1016/j.cose.2021.102200.

Rademacher, F., Sachweh, S., Zündorf, A., 2020. A modeling method for systematic architecture reconstruction of microservice-based software systems. Enterp. Bus.-Process Inf. Syst. Model. 387, 311–326.

Rahman, M.I., Panichella, S., Taibi, D., 2019. A curated dataset of microservices-based systems.

Rubin, J., Chechik, M., 2013. A Survey of Feature Location Techniques. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 29–58. http://dx.doi.org/10.1007/978-3-642-36654-3_2.

Savage, T., Revelle, M., Poshyvanyk, D., 2010. Flat3: feature location and textual tracing tool. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, Vol. 2. pp. 255–258. http://dx.doi.org/10.1145/1810295.1810345.

Schiewe, M., Curtis, J., Bushong, V., Černý, T., 2022. Advancing static code analysis with language-agnostic component identification. IEEE Access 10, 1. http://dx.doi.org/10.1109/ACCESS.2022.3160485.

Schneider, S., Özen, T., Chen, M., Scandariato, R., 2023. microSecEnD: A Dataset of Security-Enriched Dataflow Diagrams for Microservice Applications. In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR). http://dx.doi.org/10.5281/zenodo.7714926.

Schneider, S., Scandariato, R., 2023. Code2DFD git repository. URL https://github.com/tuhh-softsec/code2DFD.

Seiler, M., Paech, B., 2017. Using tags to support feature management across issue tracking systems and version control systems. In: Grünbacher, P., Perini, A. (Eds.), Requirements Engineering: Foundation for Software Quality. Springer International Publishing, Cham, pp. 174–180.

Shostack, A., 2014. Threat Modeling: Designing for Security, first ed. Wiley Publishing.

Sion, L., Van Landuyt, D., Yskout, K., Joosen, W., 2018a. Sparta: Security & privacy architecture through risk-driven threat assessment. In: 2018 IEEE International Conference on Software Architecture Companion (ICSA-C). pp. 89–92. http://dx.doi.org/10.1109/ICSA-C.2018.00032.

Sion, L., Yskout, K., Van Landuyt, D., Joosen, W., 2018b. Solution-aware data flow diagrams for security threat modeling. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC '18. Association for Computing Machinery, New York, NY, USA, pp. 1425–1432. http://dx.doi.org/10.1145/3167132.3167285.

Sion, L., Yskout, K., Van Landuyt, D., van den Berghe, A., Joosen, W., 2020. Security Threat Modeling: Are Data Flow Diagrams Enough?, ICSEW'20. Association for Computing Machinery, New York, NY, USA, pp. 254–257. http://dx.doi.org/10.1145/3387940.3392221.

Soldani, J., Muntoni, G., Neri, D., Brogi, A., 2021. The mtosca toolchain: mining, analyzing, and refactoring microservice-based architectures. Softw. - Pract. Exp. 51, http://dx.doi.org/10.1002/spe.2974.

Stojanovic, T.D., Lazarevic, S.D., Milic, M., Antovic, I., 2020. Identifying microservices using structured system analysis. In: 2020 24th International Conference on Information Technology. IT, pp. 1–4. http://dx.doi.org/10.1109/IT48810.2020.9070652.

Torr, P., 2005. Demystifying the threat modeling process. IEEE Secur. Priv. 3 (5), 66–70. http://dx.doi.org/10.1109/MSP.2005.119.

Tuma, K., Scandariato, R., Balliu, M., 2019. Flaws in flows: Unveiling design flaws via information flow analysis. In: 2019 IEEE International Conference on Software Architecture. ICSA, pp. 191–200. http://dx.doi.org/10.1109/ICSA.2019.00028.

Tuma, K., Scandariato, R., Widman, M., Sandberg, C., 2018. Towards security threats that matter. In: Katsikas, S.K., Cuppens, F., Cuppens, N., Lambrinoudakis, C., Kalloniatis, C., Mylopoulos, J., Antón, A., Gritzalis, S. (Eds.), Computer Security. Springer International Publishing, Cham, pp. 47–62.

Walker, A., Laird, I., Cerny, T., 2021. On automatic software architecture reconstruction of microservice applications. In: Kim, H., Kim, K.J., Park, S. (Eds.), Information Science and Applications. Springer Singapore, Singapore, pp. 223–234.

Yarygina, T., Bagge, A.H., 2018. Overcoming security challenges in microservice architectures. In: 2018 IEEE Symposium on Service-Oriented System Engineering. SOSE, pp. 11–20. http://dx.doi.org/10.1109/SOSE.2018.00011.

Zhao, W., Zhang, L., Liu, Y., Sun, J., Yang, F., 2004. Sniafl: towards a static non-interactive approach to feature location. In: Proceedings. 26th International Conference on Software Engineering. pp. 293–303. http://dx.doi.org/10.1109/ICSE.2004.1317452.

Simon Schneider is a Ph.D. student at the Institute of Software Security at the Hamburg University of Technology (TUHH), Germany. He works on lightweight techniques for the detection of security features in microservices' code and their automated analysis.

Riccardo Scandariato received his Ph.D. in Computer Science in 2004 from Politecnico di Torino, Italy. In his academic career he worked in several countries, including the United States (University of Virginia, 2003), Italy (Politecnico di Torino, 2004–2005), Belgium (KU Leuven, 2006–2014) and Sweden (University of Gothenburg, 2014–2020). Since late 2020, he is the head of the Institute of Software Security at the Hamburg University of Technology (TUHH), in Germany.