Scalable Compositional Static Taint Analysis for Sensitive Data Tracing on Industrial Micro-Services

Zexin Zhong†
University of Technology Sydney
Sydney, Australia
Ant Group
Hangzhou, China
zhongzexin.zzx@antgroup.com

Jiangchao Liu
Ant Group
Hangzhou, China
jiangchao.ljc@antgroup.com

Diyu Wu
Ant Group
Hangzhou, China
wudiyu.wdy@antgroup.com

Peng Di†
Ant Group
Hangzhou, China
dipeng.dp@antgroup.com

Yulei Sui
University of New South Wales
Sydney, Australia
yulei.sui@uts.edu.au

Alex X. Liu
Ant Group
Hangzhou, China
alexliu@antgroup.com

John C.S. Lui
Chinese University of Hong Kong
HongKong, China
cslui@cse.cuhk.edu.hk

***Abstract*—In recent years, there has been an increasing demand for sensitive data tracing for industrial microservices; these include change of governance, data breach detection, to data consistency validation. As an information tracking technique, Taint analysis is widely used to address these demands. This paper aims to share our experience in developing a scalable static taint analyzer on sensitive data tracing for large-scale industrial microservices. Although several taint analyzers have been proposed for Java applications, our experiments show that existing approaches are inefficient and/or ineffective (in terms of low recall/precision rates) for analyzing large-scale industrial microservices.**

Instead, we present CFTaint, a compositional field-based taint analyzer, to address the challenges for popular microservices running on industrial Fintech applications. CFTaint improves scalability by using a fast compositional function summary, which summarizes the data propagation of each function during the on-the-fly taint analysis. CFTaint also uses a novel filed-based algorithm to analyze the taint propagation based on specified sensitive fields to reduce false negatives. Our field-based algorithm maximizes the soundness of our approach even when the taint tracking is performed on an unsound call graph. Furthermore, we also propose an efficient code transformation method to model the behaviours of the containers, which allows our analysis to trace data propagation in a container environment. Experiments on numerous production microservices demonstrate the high recall (96.09%) rates and precision (93.51% for tracing sensitive data) of CFTaint with low time complexity (121.73 seconds).

***Index Terms*—program analysis, taint analysis, micro-services**

I. Introduction

Taint analysis is an information tracking technique that aims to analyze the control and data dependence from a source to a sink. It has been widely used in software analysis and testing to detect sensitive data leakage and various vulnerabilities (e.g., memory errors and code injection attacks) [1]–[8]. According to a recent study [9], data breaches incurred an average cost of 4.42 million US dollars in the year 2020. These data breach vulnerabilities also impose large FinTech enterprises to spend more than millions of US dollars annually to protect their software security. With the increasing sophistication of business logic and the continuous expansion of industrial microservices used in these FinTech companies (e.g., Ant Group, PayPal, etc.), there is an urgent demand for scalable taint analysis tools which can run on mega-scale microservices.

The static analysis aims to approximate runtime behaviour without running the program so that a vulnerability can be captured in the early stage of a software development cycle. This paper presents a "scalable" compositional static taint analyzer for industry-scale micro-services. Currently, there are several static taint analysis solutions for traditional software systems, such as FlowDroid [10], F4F [11], DroidInfer [12] and ANTaint [2]. We note that as a variant of FlowDroid, ANTaint is inadequate in analyzing large-scale micro-services applications. Built upon FlowDroid [10], ANTaint [2] improves its scalability via building the call-graph and propagating taints in an on-demand manner. To accurately analyze the traditional software systems, ANTaint requests a complete and precise call-graph and a self-defined taint propagation model. However, the scalability of ANTaint's using the IFDS framework faces many difficulties in analyzing micro-services applications with millions of lines of code. Here, we state the challenges in designing industrial-scale taint analysis tools:

Challenge 1: Recall. Recall is one of the major obstacles to precisely analyzing taint propagation on large-scale industrial micro-services. Most conventional static taint analyzers conduct their analysis based on a pre-built callgraph [2], [10], [12], [13], which means their analysis performances, such as precision and recall, rely solely on the soundness of the underlying call-graph. For example, FlowDroid [10], the state-of-the-art IFDS-based static program analysis study, executes its interprocedural program analysis by relying on the pre-built call-graph. However, it is extremely hard to build a sound call-graph statically for large-scale industrial micro-

† is corresponding author.

services applications due to the usage of complex framework behaviours such as AOP (Aspect Oriented Programming), message services, IOC (Inversion of Control), reflection, and events [14]. Although some studies claimed that the missing call-sites related to such framework behaviours could be supplemented to the call-graph by manually modelling these framework behaviours [2], it is inherently inefficient and costly for human resources. In addition, it is also error-prone and almost impossible to guarantee that all framework behaviors are properly modelled. Any deficiency of framework behaviors models can cause missing caller-callee relations of the related framework behavior invoked in the generated call graph. Therefore, it will significantly reduce the recall rate of the static analyzers.

Challenge 2: Scalability. Scalability is another major challenge for static taint analysis of industrial microservices. Previous static analyzers suffered from both high time and memory consumption for precise analyzing real-world industrial microservices. Note that modern industrial microservices typically consist of many modules with multiple libraries [14]. This means that the microservices are often of large-scale and are very complex. Despite the possibility of overcoming the recall challenge and obtaining a sound and precise call graph, the generated call graph is often huge. Performing a precise context-sensitive interprocedural analysis on such large-scale call graphs is computationally expensive. Hence, it is intractable for conventional analyzers to execute precise context-sensitive analysis on industrial microservices. Moreover, functions invoked under different contexts must be (re)analyzed repeatedly by tools that follow a top-down or bottom-up analysis pattern to perform their interprocedural analysis. In addition, for analyzers with field-sensitivity [10], memory requirements could be very demanding if the heap is abstracted precisely for all instances [14]. Field-insensitive analysis, which does not distinguish fields of an object and treats all fields as a single object, could provide a scalable alternative, but it can reduce the precision of the analysis [14]. Therefore, scalability is one of the major concerns for precise static taint analysis on industrial microservices.

Challenge 3: Precision. Precision is also a critical metric for static taint analysis. Complex containers, such as map, list, or JSON object, are heavily used in industrial micro-services applications. Such containers are often used in a sensitive data propagation scenario in industrial micro-services. For example, as shown in Figure 1, the object that contains sensitive data source (line 4 in Figure 1) is propagated and stored into a list. In this case, because most previous field-sensitive analyzers cannot trace the data propagation of such containers precisely, the whole list will be marked as tainted, which will cause a high number of false positives. For instances, the operation of `list.get(i)` or the operation of `list.get(i).f2` at the sink (line 5 in Figure 1) will be mistakenly labelled as data leakage in this scenario. Hence, conventional analysis results are often misleading for sensitive data tracing tasks and make the analysis results meaningless for developers due to the massive amount of false positives.

```
1 public void taint(){
2     Object obj = new Object();
3     obj.f1 = source();
4     list.add(obj);
5     sink(list.get(i).f2);
6 }
```

Fig. 1: Example of Challenge3

In this paper, we share our experience in addressing the above challenges in analyzing large-scale microservices applications. In order to make taint analysis efficient and accurate for large-scale microservices, we propose a new *"field-based compositional static taint analysis"* approach for tracking sensitive data. To address the recall problem (**Challenge 1**), we have implemented our analyzer based on the field-based technique rather than the field-sensitive approach. Our field-based taint analyzer, which distinguishes fields of the same type but merges all instances of the same type, relies less on the pre-built call graphs (which will be explained in detail in Sections 3 and 4) than previous field-sensitive approaches. This feature enables our approach to be less dependent on the impact of the missing caller-callee relations of the related framework behaviour invoked in the generated call graph, and maintain a high recall rate of our analysis.

To address scalability (**Challenge 2**), our approach leverages a concept in static program analysis called *"compositional program analysis"* and we extend it to the field-based compositional taint analysis to trace sensitive data in industrial microservices applications statically. Our approach adopts a function summary-based compositional analysis, which analyzes and summaries each function in the microservices separately so to abstract the data flow of each function. Each function will be analyzed once, and the function summary will be reused for later interprocedural analysis, which repeatedly analyzes function calls when computing the tainted value-flows. Unlike previous compositional program analyses, which overlook the context information [15], to maintain the precision of the compositional analysis when building the function summary, our compositional taint analyzer keeps and stores the valid context information in the function summary.

Furthermore, to guarantee the accuracy of sensitive data tracing within containers, we model the operation of the frequently used containers. Summary models are built so as to abstract the taint propagation features (**Challenge 3**).

In this paper, we present CFTaint, a highly scalable static taint analyzer for industrial microservices. CFTaint has a much higher recall rate while maintaining good precision when compared with other state-of-the-art. Specifically, we note the following contributions:

- We present a new field-based compositional static taint analyzer for sensitive data tracing on large-scale microservices.
- We compare the precision and recall of field-based with field-sensitive analysis. The comparison results demonstrate that the precision loss of field-based analysis on industrial microservices for tracing sensitive data is limited because of the strict usage rules of fields in industrial

microservices applications.

- We have implemented our approach in CFTaint and evaluated with the open-source micro-benchmark [2] and the production benchmark. The results demonstrate that CFTaint greatly outperforms the state-of-the-art ANTaint [2] and P/Taint [16] in scalability and with promising recall (96.09%) and precision (93.51%) rates.

II. Motivation and Approach Overview

The microservices architecture, a variant of the service-oriented architecture (SOA) structural style, allows developers to quickly build and deploy applications. Microservices applications consist of a collection of small, autonomous services for handling large-scale distributed transactions. Microservices architecture is widely used in the industry for integrating applications, i.e., SOFA [17], Spring Cloud [18] and Apache Camel [19] are some well-known open-source microservices architectures. Also, SOFA, a variant based on Spring, has been heavily used in FinTech systems. However, due to the complexity of these microservices, there are only a few practical taint analyzers available [2] for large-scale industrial microservices. However, the demand for tracking sensitive data on microservices has been increasing significantly. In this section, we illustrate the challenges and the fundamental idea of our field-based compositional taint analyzer using a sensitive data tracing in an industrial microservices framework.

```
1  // Facade layer
2  @SofaService(uniqueId = "refServiceId")
3  public class ServerFacadeImpl implements ServerFacade{
4    private SampleService service = new SampleServiceImpl();
5    @Override
6    public void querySampleResult(Request request){
7      Model model = new Model();
8      model.setPhoneNumber(request.getPhoneNumber());
9      service.doProcess(model);}
10 }
11 // Service layer
12 public class SampleServiceImpl implements SampleService{
13   @Override
14   public void doProcess(Model model){
15     SampleDO sampleDO = new SampleDO();
16     sampleDo.setPhoneNumber(model.getPhoneNumber());
17     // potential sensitive data leak
18     DAO.insert(sampleDO);}
19 }
20 // Interceptor class that configured in Spring XML
21 public class TaskInterceptor extend Interceptor {
22   public void interceptorInvoke (MethodInvocation invocation
         ){
23     Model interModel = invocation.getMethod().getParameter()
           .get(0);
24     // potential sensitive data leak
25     Log.print(interModel.getPhoneNumber());}
26 }
```

Fig. 2: An Illustrating Example

A. Motivating Example

Figure 2 shows a simple micro-services snippet that is written under the SOFA-RPC framework. The function `interceptorInvoke` of `TaskInterceptor` is an example of the interception function which has been configured in the Spring XML configuration, and it will be executed before the invocation of `doProcess()`. Consider an example that the service interface is exposed by the SofaService annotation with a unique service id refServiceId. The functionality of the operation at `querySampleResult(Request request)` is to read the sensitive user phone number from the request and store it into an object of `Model`. Then it calls `doProcess(Model model)` at the service layer, which writes the user phone to DB through `SampleDO`. The interception function `interceptorInvoke` of `TaskInterceptor` will then write the sensitive user phone number to a log file before the invocation of `doProcess()`. In this snippet example, there are two potential sensitive data leakage: one is to the DB in `doProcess()` (Line 18 in Figure 2); another potential sensitive data leakage is to the write to log file via the function `interceptorInvoke` at the `TaskInterceptor` (line 25 in Figure 2). To trace the sensitive data so to detect such potential data leakage, let us elaborate on the challenges for existing analysers and briefly introduce how our field-based compositional approach addresses these challenges.

B. Field-based Compositional Analysis

Field-based algorithm for interprocedural analysis. The field-based analysis is a technique to address the low recall rate of the state-of-the-art static taint analysers caused by the dependency on the unsound pre-built call graph. Most previous static program analysers require a precise and complete call graph to connect data propagation between methods in a function call for performing the interprocedural data flow analysis. However, as we mentioned in **Challenge 1**, it is difficult, if not impossible, to build a sound call-graph statically for large-scale industrial micro-services applications. For example, existing static taint analysers would fail to trace data propagation that caused the data leakage at the `interceptorInvoke` function (line 25 in Figure 2) when the interception mechanisms defined in Spring XML files are not properly modelled. This failure is caused by the missing caller-callee relation of function calls supported by the framework behaviours on the generated call graph. In order to maintain the taint analyzers' recall rate, we have adopted a field-based analysis, which is less dependent on the pre-built call graph than the field-sensitive analysis, to connect different methods for the interprocedural data flow analysis. Unlike previous static taint analysers, which depend on a precise pre-built call graph to perform their interprocedural analysis, our field-based analysis approach distinguishes fields of the same class of object, but merges all different instances of the same type to connect the interprocedural data propagation. Because sensitive data will be propagated by the specified fields in different methods, by connecting all propagations of the fields in different methods in the program, all data propagations of fields in the program could be traced without the call graph. For instance, for the example shown in Figure 2, our field-based analysis approach will treat the field `phoneNumber` of

the class `Model` as the same everywhere and merge all its data propagation in the `querySampleResult`, `doProcess` and `interceptorInvoke` functions. Therefore, using this approach, even if the pre-built call graph is incomplete due to the lack of modelling of the interception mechanism, our field-based could still find the sensitive data is propagated to both the DB and log file. The detailed introduction of this field-based algorithm will be introduced in Section IV-A.

Precision of Field-based analysis. Although merging instances in the field-based analysis may cause precision loss compared to field-sensitive analysis, this has a limited effect when applied to industrial microservices, especially in tracing sensitive data. In most industrial scenarios for tracing sensitive data on microservices applications, developers usually are more concerned about the property of leaked data than the source of leaked data [14]. In most tracing sensitive data cases, developers care more about whether the data exposed at the sink is a field used to hold the sensitive data in the microservices. However, the exact data stored in this field at the source is not of the main concern. Furthermore, based on our investigation of the industrial microservices application (which will be discussed detailed in Session IV-C), the usage of each field in the industrial microservices application has various strict rules. Each field in the microservices usually represents a concept that remains unchanged in all its usage in the application. For example, consider the sensitive fields `password` of the object user in the application, which is assigned with a specific concept: the user password. Developers will only use this field to store the user password and will never use it to carry other information, such as the user id. Similarly, other fields (e.g., id) are never used to hold the other sensitive data, i.e., user password. Thus, even though our field-based analysis is less precise than the field-sensitive analysis, these strict rules limit the precision loss. In addition, in order to compensate for the precision loss, we also model the operation of the frequently used containers (which will be discussed in Session IV-C), such as map, list and JSON object, to support the field-based algorithm to precisely trace sensitive data propagation within those containers by constant keys.

Compositional Analysis The compositional analysis, which combines the field-based algorithm, could provide faster and less memory consumption static taint analysis for more scalable sensitive data tracing on large-scale industrial microservices applications. Most existing static taint analysers follow a top-down analysis pattern to analyse programs starting from a specified entry function. These analysers process their taint analysis in the order based on the generated call graph. However, as we mentioned above, it is challenging to generate a precise call graph statically. Even though it is possible to generate a sound call graph, the scale of the generated call graph is usually huge due to the large-scale industrial microservices applications. According to the findings in [2], in a real-world industrial environment, independent of the size of the programs, even small programs may touch a large part of the libraries. Performing a precise context-sensitive interprocedural analysis on a large-scale call graph is inherently computationally expensive [14]. Moreover, unlike traditional applications, a microservices application usually has multiple entry functions since all its exposed methods could be the entry methods that other microservices could trigger. Thus, analysers that follow the top-down analysis pattern need to analyse the whole program with different entry methods to guarantee its coverage, and this introduces inefficiency. What is worse, if the heap is abstracted precisely for instances with field-based analysis, the memory consumption will be huge. Repeat analysis for the same function in different invokes with different context information is also required. Therefore, it is costly to run precise context-sensitive interprocedural analysis on such a large-scale call graph [14] based on the top-down analysis pattern with the field-sensitive analysis.

Hence, to make the analysis scalable for running on large-scale microservices applications, our proposed compositional analysis adopts a bottom-up analysis pattern instead of the top-down pattern (which will be introduced detailed in Session IV-B). In our bottom-up analysis pattern, the analysis does not need a specific entry method to start the analysis. It randomly picks up a function to start the analysis and executes the intraprocedural analysis on each method of the application to build the data propagation summary for each function. Because the intraprocedural analysis adopts the field-based analysis to summarise the data propagation for each function, the interprocedural data propagations are connected by these generated field-based functions summary. Thus, using the field-based algorithm, we could execute our interprocedural analysis based on the generated function summary without a precise call graph. Furthermore, our function summary has been designed to keep the input parameters and the method's return value for computing different data propagation results with different input parameters. The function summary is reusable for computing the data propagation in different invokes of the same function. Therefore, repeat analysis of functions in different context information is not required in our compositional analysis. Moreover, because the function summary construction is based on the field-based algorithm, which merges instances of the same type of object, the memory consumption is less than that of the field-sensitive analysis. Thus, our field-based compositional analysis provides faster and less memory consumption static taint analysis for more scalable sensitive data tracing on large-scale microservices.

III. System Architecture

In this section, we present the architecture of our static taint analysis tool CFTaint. Note that our proposed techniques can be extended to other platforms, currently, it is built over the SOFA/SOFABoot [17] framework, a well-known financial variant of the Spring/SpringBoot micro-services architecture.

As shown in Figure 3, our static taint analysis platform takes the production jar packages (including applications jar packages and their dependencies) of the micro-services applications as input. All these byte code jar packages will be processed by GraalVM [20] and be transformed to Graal IR.

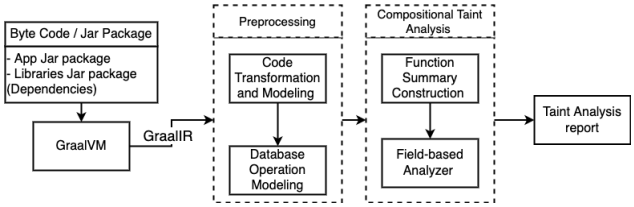

Fig. 3: System Architecture

Pre-processing Module. The pre-processing module aims to model the hidden code and configurations of the SOFA framework as we need and the middlewares such as Spring/-SOFA Bean or MyBatis configurations [17]. It analyzes the JAVA source code and the XML configuration files of the micro-services applications and reveals the essential information to the taint analysis module. For example, the pre-analysis will detect the entry interface methods, which other applications can access. The technical details of the preprocessing module will be provided and discussed in Section IV-C and also detailed in our technical engineering report [21].

Compositional Taint Analysis Module. The compositional taint analysis module is the core component of CFTaint. It is to perform the field-based compositional taint analysis task and to provide the taint propagations results of the sensitive fields. The "function summary construction" and "field-based analyzer" are two components of this module. The "function summary construction" analyzes the data propagation of all program functions and generates corresponding function summaries with the filed-based algorithm. The field-based analyzer then composes all the function summaries together to construct a field-based transition graph. Finally, the field-based analyzer will query the taint propagation facts on the generated taint value-flow graph based on the given sources and sinks to produce analyzing results. The technical details will be presented and discussed in Section IV-A and Section IV-B.

IV. OUR APPROACH

This section introduces the technical details of the field-based compositional static taint analyzer. We show how the compositional tool addresses the challenges in scalability without compromising too much on precision and explain how the field-based analysis can maintain a good recall rate. We also present how precision is maintained in our approach.

A. Field-based Analysis for Higher Recall Rate

Our proposed approach is implemented with a "field-based" algorithm rather than "field-sensitive". Using this technique, we aim to demonstrate how the field-based analysis solves the recall challenge (**Challenge 1**). Different from the field-sensitive approach, which distinguishes both fields and instances of objects, our field-based approach does not distinguish different instances of the same type but distinguishes fields of the same kind. Although most existing studies claim their proposed approach is sound, their soundness really depends on the soundness of the underlying call-graph. However, as we mentioned in **Challenge 1**, it is difficult, if not impossible, to construct a thoroughly sound call graph because there can be many missing caller-callee relations of function calls supported by the framework behaviours, such as the interceptor behaviours that are shown in the motivating example (Figure 2). As we mentioned, the field-sensitive analyzers, such as Flowdroid [10] and ANTaint [2], are required to distinguish both fields and instances of objects to maintain the precision of the analysis. In order to distinguish different declared instances in the inter-procedural analysis, the field-sensitive analyzers need a complete and precise call-graph to trace the context of the different instances to distinguish them. However, because our field-based analysis only distinguishes different fields of the same object type but merges all declared instances of the same type, our field-based analysis algorithm does not need to rely on the pre-build call graph to distinguish different instances. By using "field-based" rather than "field-sensitive", our field-based analysis relies less on a pre-built call graph than field-sensitive analysis. Even when the call-graph is incomplete, the field-based analysis supports the analyzer to perform the static analysis with a high recall rate. For example, refer to the study described above in Figure 2, two potential sensitive data leakage could be identified in this case: one is to the database at the `doProcess` (Line 18 in Figure 2); another data leakage is to the log file in the function `interceptorInvoke` (Line 25 in Figure 2). In this case, most static analyzers could easily find the first potential data leakage that exposes data to the database but fail to find the other cases if the interception mechanism configured in Spring XML files is not properly modelled. However, our approach benefits from the underlying mechanism of the field-based algorithm, which merges different instances of the same type but distinguishes fields of the same kind, the field `PhoneNumber` of the object `Model` will be seen as the same everywhere in the program. Thus, the field-based algorithm allows the analyzer to identify the data leakage to both the database and the log file in the example case without modelling the framework behaviour, such as the interception mechanism configured in the Spring XML. Although our field-based algorithm is less precise than the field-sensitive algorithm, which is because of the false negatives generated by merging all instances of the same type in some cases, the precision loss is limited when analyzing the industrial applications, especially in tracing sensitive data in industrial applications. We will introduce details in section IV-C.

B. Compositional Analysis for Scalable Taint Tracking

To maintain the scalability of sensitive data tracking in large-scale microservices, we adopted a compositional analysis algorithm for our analyzer. Algorithm 1 shows the new compositional inter-procedure taint analysis. Unlike the previous state-of-the-art IFDS-based taint analyzers, such as FlowDroid and ANTaint, which follow a top-down pattern to analyze the application from a specified entry function and process the taint analysis in order based on the call-graph, our compositional analysis algorithm follows a bottom-up analyzing pattern. Instead of analyzing from the entry-points of an application $\mathbb{A}$, CFTaint performs an out-of-order

$$\mathbb{T}_p = \{\langle p_i, f\rangle \in \mathbb{T}_p \mid p_i \in \mathbb{P} \quad f \in \mathbb{F} \quad p_i \xrightarrow{taint} f\}$$
$$\mathbb{R}_f = \{f \in \mathbb{R}_f \mid f \in \mathbb{F} \quad m \xrightarrow{return} f\}$$
$$\mathbb{R}_p = \{p_i \in \mathbb{R}_p \mid p_i \in \mathbb{P} \quad m \xrightarrow{return} p_i\}$$

Fig. 4: Summary Components

analysis for each method and generates a context-sensitive field-based information propagation graph $\mathbb{G}$, which is a set of tuple $\langle f, f'\rangle$ demonstrating that a field f could taint f' in the application. At the start of the algorithm, the information propagation graph $\mathbb{G}$ and the summary set $\mathbb{S}$ are initialized with empty set. Then we randomly build a summary for each method $m \in \mathbb{M}$, in which $\mathbb{M}$ is the set of all methods, using the function `BuildSummary()`. All methods are summarized once and reused for all their inter-procedural function invokes. During the summary building process, the information graph $\mathbb{G}$ will be updated. Once a summary $\mathcal{S}$ for a method m is built, it will be added to the set $\mathbb{S}$. The following sections will elaborate the summary-building process in detail.

1) Intra-Procedure Summary Building: For each summary $\mathcal{S}$, we maintain three sets, as shown in Figure 4, for describing the potential data propagation information for the method. Since of analysis is field-based, the method summary aims to trace the final behaviour of class fields and method parameters by merging the local objects' operations within the method. The set $\mathbb{T}_p$ is a set of tuple $\langle p_i, f\rangle$ where p_i is the *i-th* parameter of a method m, and f is a class field that may be potentially tainted by p_i. As for $\mathbb{R}_f$ and $\mathbb{R}_p$, they contain the class fields f and parameters p_i separately that could be returned by the method m.

For a given method $m \in \mathbb{M}$, the function `BuildSummary()` will first initialize the three sets $\mathbb{T}_p$, $\mathbb{R}_f$ and $\mathbb{R}_p$ for the generated summary $\mathcal{S}$. In addition, a middle state set $\mathbb{MID}$ is also created in order to store the transition relations that the local variable is invoked (line 7 in Algorithm 1). The intra-procedure analysis mainly focuses on each assignment statement and return statement in m, which is shown in lines 10-22. For an assignment statement in the form of $v \leftarrow v'$ that the variable v' is assigned to the variable v, our algorithm considers different scenarios. If the assigned variable v is a local variable, the algorithm updates the middle state set $\mathbb{MID}$ with a tuple $\langle v, v'\rangle$ at line 12. The variable v' could be in any form, including class field, parameter or local variable. When v is a class field $\_.f$, we query the data sources of the variable v' using $find(v', \mathbb{MID})$ at line 14. It will return all the previously stored data sources of v' in the set $\mathbb{MID}$. For each parameter data source p_i of variable v', the set $\mathbb{T}_p$ will be updated by adding a tuple $\langle p_i, f\rangle$ indicating the field f could be potentially tainted by the parameter p_i. For the condition that the data source is a class field $\_.f'$, the algorithm directly adds a tuple $\langle f', f\rangle$ to the propagation graph $\mathbb{G}$ at line 17, which means the field f' is able to taint f. Between lines 18 and 22, our algorithm handles the return statements of m. If a variable v is returned by a statement, the function $find()$ is also invoked to obtain all the data sources of v. The set $\mathbb{R}_p$ and $\mathbb{R}_f$ will be updated separately based on the different types of data sources.

Algorithm 1: Inter-Procedure Taint Analysis

Input : application $\mathbb{A}$
Output: $\mathbb{G}$

1 $\mathbb{G} = \emptyset,\ \mathbb{S} = \emptyset$
2 **foreach** $m \in \mathbb{M}$ **do**
3 $\quad \mathcal{S}_m = BuildSummary(m, \mathbb{G}, \mathbb{S})$
4 $\quad \mathbb{S} = \mathbb{S} \cup \{\mathcal{S}_m\}$
5 **return** $\mathbb{G}$
6 **Function** `BuildSummary(`$m, \mathbb{G}, \mathbb{S}$`)`**:**
7 $\quad \mathbb{T}_p = \emptyset,\ \mathbb{R}_f = \emptyset,\ \mathbb{R}_p = \emptyset,\ \mathbb{MID} = \emptyset$
8 $\quad \mathcal{S}_m = \{\mathbb{T}_p, \mathbb{R}_f, \mathbb{R}_p\}$
9 $\quad$ **foreach** $statement \in m$ **do**
10 $\quad\quad$ **if** $statement : v \leftarrow v'$ **then**
11 $\quad\quad\quad$ **if** v *is local* **then**
12 $\quad\quad\quad\quad \mathbb{MID} = \mathbb{MID} \cup \{\langle v, v'\rangle\}$
13 $\quad\quad\quad$ **else if** $v : \_.f$ **then**
14 $\quad\quad\quad\quad$ **foreach** $p_i \in find(v', \mathbb{MID})$ **do**
15 $\quad\quad\quad\quad\quad \mathbb{T}_p = \mathbb{T}_p \cup \{\langle p_i, f\rangle\}$
16 $\quad\quad\quad\quad$ **foreach** $\_.f' \in find(v', \mathbb{MID})$ **do**
17 $\quad\quad\quad\quad\quad \mathbb{G} = \mathbb{G} \cup \{\langle f', f\rangle\}$
18 $\quad\quad$ **else if** $statement : return\ v$ **then**
19 $\quad\quad\quad$ **foreach** $p_i \in find(v, \mathbb{MID})$ **do**
20 $\quad\quad\quad\quad \mathbb{R}_p = \mathbb{R}_f \cup \{p_i\}$
21 $\quad\quad\quad$ **foreach** $\_.f \in find(v, \mathbb{MID})$ **do**
22 $\quad\quad\quad\quad \mathbb{R}_f = \mathbb{R}_f \cup \{f\}$
23 $\quad\quad$ **else if** $statement : v = m'(v')$ **then**
24 $\quad\quad\quad \mathcal{S}_{m'} = generate(m', \mathbb{S})$
25 $\quad\quad\quad$ **foreach** $p_i \in find(v', \mathbb{MID})$ **do**
26 $\quad\quad\quad\quad$ **foreach** $\langle p'_i, f\rangle \in \mathbb{T}'_p$ **do**
27 $\quad\quad\quad\quad\quad \mathbb{T}_p = \mathbb{T}_p \cup \{\langle p_i, f\rangle\}$
28 $\quad\quad\quad$ **foreach** $\_.f' \in find(v', \mathbb{MID})$ **do**
29 $\quad\quad\quad\quad$ **foreach** $\langle p'_i, f\rangle \in \mathbb{T}'_p$ **do**
30 $\quad\quad\quad\quad\quad \mathbb{G} = \mathbb{G} \cup \{\langle f', f\rangle\}$
31 $\quad\quad\quad \mathbb{MID} = \mathbb{MID} \cup \{\langle v, src\rangle \mid src \in \mathbb{R}'_f \mid\mid src \in \mathbb{R}'_p\}$
32 **return** $\mathcal{S}_m$

2) Inter-procedure Summary Building: The inter-procedure summary building focuses on connecting the summaries of different methods, which is handled between lines 23 and 31 in Algorithm 1. When the analysis encounters an invoke statement $v = m'(v')$, where the variable v could be void if the method m' does not return anything. And there could be any number of v' based on how the parameters m' have. For finding the method target of m' in the presence of virtual invoke in JAVA, our tool applies a CHA-based strategy with the enhancement of an intra-procedure type analysis. At line 24, $generate()$ will either directly return the method summary $\mathcal{S}_{m'}$ if there is an existing summary in $\mathbb{S}$, or will call the function $BuildSummary()$ to create $\mathcal{S}_{m'}$ in a depth-first manner. After acquiring $\mathcal{S}_{m'}$, the algorithm finds the data sources of the parameter variable v' at lines 25 and 28. For each data source of v' is a parameter p_i, we get the set $\mathbb{T}'_p$

from $\mathcal{S}_{m'}$. Each tuple $\langle p'_i, f\rangle \in \mathbb{S}_{m'}$ that is related to v' in m', we update $\mathbb{T}_p$ in m with $\langle p_i, f\rangle$. If a data source of v' is field f', we add the tuple $\langle f', f\rangle$ to the graph $\mathbb{G}$ based on the information of $\mathbb{T}'_p$ of $\mathcal{S}_{m'}$. At last, if m' has return values, we every tuple $\langle v, src\rangle$ to the middle state set $\mathbb{MID}$ of m, where src is every variable in the set $\mathbb{R}'_f$ and $\mathbb{R}'_p$ from $\mathcal{S}_{m'}$.

The reusable function summary for each function provides better scalability for our compositional taint analysis. Our compositional taint analysis benefits from the reusable function summary, which includes the parameter set, return set, and field propagation set of each function. Even if a specified method is invoked in different inter-procedural function calls with different contexts, the generated function summary could be reused under different contexts for updating the data propagation. Thus, repeat analysis of the same method with the different contexts is not required in our compositional analysis, and this reduces the time consumption of the analysis and makes the analysis more efficient. It is also worth mentioning that because our intra-procedural summary building follows the rules of the underlying field-based algorithm that merges all different declared instances of the same class rather than using abstract heap precisely for different instances, the huge memory consumption that caused by the traditional field-sensitive analysis can be significantly reduced in our compositional taint analysis. Thus, our compositional analysis based on the context-sensitive field-based function summary improves the scalability of the static inter-procedural taint analysis, which reduces both the time consumption and memory usage.

C. Discussion on the Precision

Hence, we present how our field-based compositional analysis resolves the challenge in precision (**Challenge 3**). Even when distinguishing fields but merging all instances of the same type in the field-based analysis will lead to less accuracy than the field-sensitive analysis, this accuracy loss is limited in industry microservices, especially when tracking sensitive data [14]. This is because every field in the industrial program usually represents a specified concept that never changes in an application during its propagation. Sensitive data is propagated among the fields assigned with specific concepts [14]. For example, refer to the example shown in Figure 2, a specified concept 'the phone number of users' is assigned to the field `phoneNumber`, which will only be used to hold the phone number of users in the application. Developers will never use other fields (e.g. address) to carry user phone numbers. Thus, this phenomenon limits the precision loss of our field-based analysis, even where all declared instances of the same type are merged in our field-based algorithm.

Furthermore, in order to maintain the accuracy of the sensitive data propagation within containers, we have studied and modelled the operations of the frequently used containers, such as map, list and JSON. This container model is integrated into the summary constructor and will be used when building the data propagation for a function. Our container model will mode and replace all container operating invokes during the data propagation summary construction. Each sensitive data propagated to the container will be managed in the model by a specified constant key. Because the constant keys are used to manage the propagations of the sensitive data within the complex containers, our compositional function summary can trace sensitive data precisely within those containers. Thus, our container model improves the precision of our tool.

V. Adapting to enterprise-specific frameworks: a practice in SOFA-based FinTech micro-services

Different enterprises have different frameworks based on their business scenario and technology stack. In this section, we introduce how we resolve the framework problems in SOFA architecture. We first performed the modelling task on the target application before the compositional taint analysis. Then, the modelling result of the target applications will be processed by the taint analysis module for the analysis. This automated process includes preprocessing of the SOFA framework (the financial micro-services variant framework of Spring), RPC and database operation.

Preprocessing and Modeling. According to our experience, the AOP, Spring Bean injections, and SOFA Bean injections features that provided by the SpringBoot [22] and SOFA [17] framework usually lead to missing knowledge about the instance type of some object or the particular code that need to be executed at some function. AOP, Spring Beans, and SOFA Beans are usually configured in configuration XML files and are used in the form of annotation in the code at the beginning or end of the function. Thus, in our case, we will first automatically scan and extract the above information from the configuration XML files based on the semantics of the SOFA-RPC and Spring XML domain language. Then we model this information in a specified format by using the code transformation and modelling supporter. These models will be loaded into the memory for the further analysis task. In our compositional taint analysis process, when the specified annotation is found, the analyzer will query the model from memory based on the annotation details and use the loaded model for the taint analysis.

Entry Point Determination. Because our proposed approach, which used the compositional analysis approach, does not rely on the traditional top-down analysis mode, determining the specified entry points is not required for our analysis. Therefore, distinct from other approaches based on the traditional top-down analysis mode, we do not need to identify the specified entry point before the analysis. The analyzer will randomly select an entry analysis function and execute the analysis process on all functions in the program to cover the whole program.

Sources and Sinks Determination. According to the successful experience of ANTaint [2] in sources and sink determination, we also assume that all the incoming data in the program will be marked as sources. These incoming data include the input parameters of the entry points, the return value of the RPC function, the interested fields or parameters provided by the developers, and the fields or objects that query from the database by the 'Select' operation. Furthermore, we

assume that all the out-coming (output) data or the output functions which will expose the data to other environments or namespace will be marked as sinks. These out-coming (output) data include all the return values from the entry points, parameters for the RPC functions, the log or system printing functions, and the object or fields that will be updated to the database by the 'update', 'insert', and 'delete' operation.

VI. EVALUATION

We evaluated our tool on multiple dimensions with two benchmarks and compared it to some open-source tools. Due to space limitations, we list and discuss the key investigations. Our evaluation aims to answer the following questions:

- *RQ1: How well can CFTaint perform in terms of precision and recall, for sensitive data tracing on industrial microservices applications?*
- *RQ2: What is the performance of our approach when it is applied to industrial microservices applications?*
- *RQ3: what is the performance of CFTaint when we compare it to existing tools, such as ANTaint, P/Taint?*

A. Implementation and Datasets

Implementation. We have implemented the proposed field-based compositional taint analysis approach as a microservice on Java8 with GraalVM, based on the SOFA [17] architecture. When our implemented service receives the request from the gateway, it analyzes the bytecode of the target microservices and then returns the analysis result. The implementation has been deployed on a set of elastic cloud clusters, each of which is with eight 2.5GHz cores and a 64 GB RAM. The implementation executes with the following production and micro benchmarks for the experiment and evaluation.

Production-benchmark. The production benchmark comprises the core production microservices most frequently used in Ant Group. Microservices are usually the critical information systems that perform the fundamental business values for enterprises. Due to business and security concerns, enterprises usually will not publish such microservices. Thus, collecting a real-world production microservices benchmark outside the enterprise for evaluation is difficult if not impossible. Thus, we evaluated the static analyzers on the production microservice of Ant Group that are written in the SOFA [17] architecture. The SOFA architecture is the open-source FinTech microservices architecture used in many FinTech enterprises. As a variant constructed based on Spring/SpringBoot, SOFA is backward compatible with Spring and SpringBoot. Thus, our tool could also extend the outstanding performance to the Spring/SpringBoot architecture microservices.

Furthermore, it is difficult for an expert to manually identify and list all the data flows because there are usually thousands of data propagation paths for each production microservices, and some propagation paths may be complex and very long. Therefore, in our evaluation, we select six production applications. Then ask the site reliability engineers and the developers to list four sensitive fields they are most concerned about and manually identify all data flows targeted to those sensitive fields as the ground truth for validation. To investigate the impact of framework behaviour and container operations on precision and recall for the analysis, we have separated these four fields into two catalogues. The data propagation of two fields will be related to the usage of framework behaviours and container operations, and the other two will not. We have executed the evaluation on FlowDroid, ANTaint, P/Taint, CodeQL and CFTaint. Due to the extensive experimental results, listing all data in this paper is difficult. Thus, we group the tools based on their underlying analyzing technique and list the outstanding performance tools of each group in this report. As an extension of FlowDroid, ANTaint performs better analysis on larger-scale microservices. Thus, we show the results of ANTaint to represent the performance of the FlowDroid and ANTaint groups. The key results of the comparative experiment with ANTaint and P/Taint are listed in Table II For the detailed report, please refer to our experiment full-version report [21].

Micro-benchmark. The micro-benchmark is an open-source [23] benchmark, which provides by Wang et al. [2] on GitHub for static taint analysis evaluation. This micro-benchmark is contributed by the industry experts, and it contains the most relevant cases of the real-world industry scenarios that developers are concerned with. The evaluation of our approach running on the micro-benchmark is shown in Table I, and we compared these results with the performance results of ANTaint that was published by Wang et al. [2].

		ANTaint			CFTaint		
Micro-benchmark	Exp	Exa	FN	FP	Exa	FN	FP
queryForPageTaint	5	5	0	0	5	0	0
resolveFromReference	10	10	0	0	0	10	0
updateRidAll	18	18	0	0	18	0	0
queryAllTaint	15	15	0	0	15	0	0
allResolve	20	20	0	0	20	0	0
saveAndQuery	15	15	0	0	15	0	0
updateRidByName	10	10	0	0	10	0	0
queryByNamesTaint	9	9	0	0	9	0	0
saveSampleByResult	10	10	0	0	10	0	0
batchResolve	10	10	0	0	10	0	0
resolveSampleResult	10	10	0	0	0	10	0
queryByCallbacks	11	11	0	0	11	0	0
testMultiplePaths4	1	1	0	0	1	0	0
testLists	3	3	0	0	3	0	0
testDeepCopies	4	4	0	0	4	0	0
Total	151	151	0	0	131	20	0

TABLE I: Micro-benchmark and Results

B. *Methodology*

We evaluate our approach by comparing it with several state-of-the-art taint analyzers, FlowDroid, ANTaint and P/-Taint, which have been used in most industrial scenarios [2]. **To answer RQ1**, we evaluate the effectiveness of CFTaint by running it on the micro-benchmark and applications of the production-benchmark, and evaluate it in terms of precision and recall. The maximum memory for each running case is set as 16 GB. Since code frequently changes in industrial scenarios, so developers would like to receive the analysis report as quickly as possible to judge the risks of the

code change. Because of this, it is unacceptable to ask the developers to wait for a long time to run a taint analysis task. Thus, we also limit the maximum running time of each analysis task to 3 hours (10800 seconds). **To answer RQ2**, we observe and collect the analysis's performance, including the time consumption and the true positives, false-positive, and false-negative taint propagation paths when running on the production benchmark. **To answer RQ3**, we compare the effectiveness and the performance of CFTaint, ANTaint and P/Taint on the production benchmark. Note that the time consumption of CFTaint is calculated based on the whole analysis duration, including the code transformation and modelling, which starts from the jar package input and ends when the final taint analysis report is generated. Table II includes two experiments. *Experiment 1* demonstrates the performance of all tools when the propagation of sensitive data includes the propagation with container operations and the framework behaviours. *Experiment 2* shows the results of all tools when the propagation of sensitive data includes the propagation without container operations and the framework behaviours.

```
1  public void failTracing () {
2      Object result = source();
3      SampleDTO dto = (SampleDTO) result;
4      sink(dto.id);}
5  public void successTrace(){
6      SampleDTO dto = new SampleDTO();
7      dto.id = source();
8      Object result = dto;
9      SampleDTO dtoClone = (SampleDTO) result;
10     sink(dtoClone.id);}
```

Fig. 5: Usage super-class object as source

C. Experimental Results and Analysis

RQ1: Effectiveness and Precision. The results in Table I demonstrate the precision of CFTaint in the micro-benchmark and compare it with ANTaint [2]. The column `'Exp.'` indicates the ground truth of the number of the taint propagation paths for the test case; the column `'Exa.'`, the column `'FN'` and column `'FP'` indicate the actual taint propagation paths found by the tools, the false-negative and false-positive results that are identified from the analysis results respectively. Although CFTaint is implemented with field-based analysis, it can process the field tracing task with high precision. As listed in Table I, CFTaint passes 93% (25 out of 27 cases) of the test scenarios in the micro-benchmark with a precision of 100%. An interesting note is that we found there are two test scenarios (highlighted in Table I) that CFTaint could not find any taint propagation paths of the sensitive field. The main cause that triggers this problem is using a super-class object as a source for the analysis, which is also the limitation of our field-based algorithm. For example, as shown in the case `failTracing()` in Figure 5, because the SampleDTO (line 3 in Figure 5) is obtained from the source, which is a super-class object, the sensitive field `dto.id` (line 4 in Figure 5) is unknown in the super-class type of source (`Object result`, line 2 in Figure 5). Therefore, we are not able to obtain where the sink `dto.id` come from, and the field-based algorithm will fail to trace the field from the source to the sink. However, if the source type is specified, which is not a super-class object type, then our field-based mechanism is able to trace sensitive fields from the source to the sink even though the tainted data is propagated to a super-class object. For instance, as shown in the `'successTrace()'` example in Figure 5, although the tainted data is propagated to a super-class object, the propagations between the source and the sink are still traceable in our field-based approach because the taint source (line 8 in Figure 5) is specified, and the same field (`dto.id`) is propagated at the sink.

RQ2: Performance. Our experiment on the production benchmarks demonstrates that CFTait can provide scalable and precise taint analysis on large-scale microservices for sensitive data tracing with limited time consumption. As one can see from the listing results (in Table II), the average time consumption of this tool is 127.31 seconds. Overall, CFTaint could process and provide the taint analysis results at the minute level in most cases, while CFTaint completes the analysis task in 199 seconds in the worst case. Benefiting from the field-based analysis, CFTaint can perform the interprocedural analysis without dependency on the call-graph. Thus, CFTaint will not be affected by the unsound call graph. It has achieved high recall rates (96.09% in *Experiment 1* and 94.35% in *Experiment 2*) in our experiments. Only 8 FNs, which are caused by using a super-class object as the taint source, are found in these scenarios. 45 FPs are found in *Experiment 1* (Table II), and 28 FPs are found in *Experiment 2* (Table II), which reduces the precision (73.2% in *Experiment 1* and 69.76% in *Experiment 2*) of our tools. However, our investigation of these FPs has revealed that this large number of FP is caused by merging instances in the field-based analysis because most of these FPs have the same concept as the one assigned to the field at the source (as indicated as `BTP` in Table II). Only 1 FP in *Experiment 1* was caused by the data propagation between containers with a non-constant value as the key. As we mentioned early, in most of the sensitive data tracing scenarios in the industry, developers usually care more about the property of the tainted data at the sink rather than the source of the tainted data. Thus, the BTP will usually be treated as correct in such scenarios. As a result, when applying the analysis for sensitive data tracing tasks in most industrial scenarios, CFTaint performs with high precision (93.51% in *Experiment 1* and 99.17% in *Experiment 2*).

RQ3: Comparison with ANTaint and P/Taint. To answer RQ3, we compare the effectiveness and performance of our tool with ANTaint and P/Taint on the micro-benchmark and production-benchmark. We have the following conclusion:

Effectiveness. From the evaluation results of the micro-benchmark, the same as ANTaint, in most scenarios, CFTaint can trace the sensitive field from source to sink without any false-negative or false-positive. However, when using a super-class type object as the taint source, 20 false-negative results are reported due to the limitation of the field-based

				P/Taint						ANTaint						CFTaint							
Micro	App (MB)	Lib (MB)	#LOC	BTP	FN	FP	R(%)	P(%)	Time(s)	BTP	FN	FP	R(%)	P(%)	Time(s)	BTP	TN	FN	FP	R(%)	P(%)	PITC(%)	Time(s)
				Sensitive data propagated with container operations and framework behaviors																			
M1	5.4	118.1	268K	13	13	8	50%	61.90%	5937	17	9	4	65.38%	80.95%	2192	25	7	1	9	96.15%	73.53%	92.59%	54
M2	9.1	165.1	250K	18	15	15	54.55%	54.55%	9273.6	24	9	12	72.73%	66.67%	5512	30	0	3	4	90.91%	88.24%	88.24%	150
M3	11.5	106.2	314K	8	6	10	57.14%	44.44%	6815	9	5	8	64.29%	52.94%	2768	14	5	0	5	100%	73.68%	100%	70
M4	21.9	29.9	709K	19	19	14	50%	57.58%	7120	22	16	13	57.89%	62.86%	3120	34	7	4	13	89.47%	72.34%	85%	199
M5	29.8	49.6	1145K	13	7	8	65%	61.90%	6368	16	4	8	80.00%	66.67%	3248	20	7	0	8	100%	71.43%	95.24%	81
M6	40.9	85.8	2454K	5	4	6	55.56%	45.45%	4068	7	2	5	77.78	58.34%	2065	9	6	0	6	100%	60%	100%	167.4
Avg	19.77	125.78	857K				55.37%	54.30%	6596.93				69.68%	64.74%	3150.83					96.09%	73.2%	93.51%	121.73
Micro	App	Lib	#LOC	Sensitive data propagated without container operations and framework behaviors																			
M1	5.4	118.1	268K	4	0	1	100%	80%	5840	4	0	1	100%	80%	2104	4	2	0	2	100%	66.67%	100%	54
M2	9.1	165.1	250K	15	5	2	75%	82.24%	9106	19	1	3	95%	86.36%	5236	19	3	1	3	95%	82.61%	95%	150
M3	11.5	106.2	314K	8	1	0	88.89%	100%	6474	8	1	0	88.89%	100%	2491	8	3	1	3	88.89%	72.73%	100%	70
M4	21.9	29.9	709K	16	3	4	84.21%	80%	6835	17	2	2	89.47%	89.47%	2901.6	17	8	2	8	89.47%	68%	100%	199
M5	29.8	49.6	1145K	14	0	4	100%	77.78%	6202	14	0	4	100%	77.78%	3183	13	8	1	8	92.86%	61.90%	100%	81
M6	40.9	85.8	2454K	5	1	2	83.33%	71.43%	3897	6	0	0	100%	100%	1861.2	6	3	0	3	100%	66.67%	100%	167.4
Avg	19.77	125.78	857K				88.57%	82.91%	6392.67				95.56%	88.94%	2962.8					94.37%	69.76%	99.17%	121.73

TABLE II: Production-benchmark and results (Note: TP indicates the identified correct/true paths; BTP(Broad True Positive) indicates false positives which have the correct/same field concept with the source; FN indicates the missed correct path (false-negatives); FP indicates the incorrect paths that have been marked as tainted (false-positive); P indicates the precision rate of the analysis; Recall indicates the recall rate; PITC indicates the precision which treats the BTP as correct (formula: TP/(TP+FP-BTP)); App indicates the application jar packages size in MB; Lib indicates the library jar packages size)

algorithm, which traces the data propagation based on the specified field. Although this problem will cause the FN in the analysis, according to the inspection of the experimental results on the production-benchmark, the effect is limited because using the super-class object as a taint source is rare in real-world industrial micro-services applications. According to the experimental results of the production-benchmark, only 8 FN (4.32%) in *Experiment 1* and 5 FN (5%) in *Experiment 2*, which are caused by the usage of a super-class object as a source, are found for CFTaint. Furthermore, 45 FN (23.31%) is found for ANTaint, and 64 FN (25.6%) is found for P/Taint on the production-benchmark in *Experiment 1*. The main reason for this enormous FN in ANTaint and P/Taint is their dependency on the unsound call-graph in their inter-procedure analysis, leading to the missing data propagation tracing to the framework behaviours (i.e., AOP, IoC, Message services, etc.).

Performance. As shown in Table II, when running on scenarios that do not contain any containers and framework behaviours during the data propagation (*Experimental 2*), conventional field-sensitive analysis, which abstracted the heap of instances, achieves higher precision than our field-based analysis. However, based on our investigation, most fields in industrial microservices will be propagated to framework behaviour calls or containers during their data propagation. Thus, as shown in the experimental results of *Experiment 1*, when framework behaviours or container operations are invoked in the data propagation of the sensitive data, the precision and recall rate of our CFTaint is higher than ANTaint and P/Taint. Experiment on the production microservices applications of *Experiment 1* indicates the high recall (96.09%) rates and precision (73.2%) of CFTaint with low time consumption (121.73 seconds on average). It outperforms the state-of-the-art ANTaint (with a precision of 64.74% and a recall of 69.68%) and P/Taint (with a precision of 54.3% and a recall of 55.37%). Although the precision of CFTaint is only 73.2% in *Experiment 1* and 69.76% in *Experiment 2*, most of the false-positives could be considered as BTPs since it has the same field concept that was assigned by the source. Thus, when considering the precision for tracing sensitive data on industrial microservices applications, CFTaint has a high precision (as indicated as `PITC` in Table II)(93.51% in *Experiment 1* and 99.17% in *Experiment 2*). Furthermore, CFTaint has the highest recall rate than other conventional analyzers in most cases. Moreover, the overall time consumption of CFTaint is 3.86% of ANTaint and 1.85% of P/Taint. For the worst case, CFTaint found all taint propagation in 199 seconds while P/Taint requires 7120 seconds and ANTaint requires 3120 seconds.

In conclusion, although the field-based algorithm is less precise than the field-sensitive, CFTaint still achieves a higher precision and recall rate than ANTaint and P/Taint when running on industrial microservices. These precision and recall rate gaps are caused by the heavily used framework behaviours and containers in industrial microservices. The heavily used framework behaviours could worsen the precision of the generated call-graph that conventional analyzers need for interprocedural analysis. The inaccuracy of data tracing in containers will amplify the tainted scope. Even though the precision of CFTaint is only 73.2% in *Experiment 1*, when considering the precision for sensitive data tracing on industrial microservices, the precision (PITC) of CFTaint (93.51%) will be much higher than ANTaint and P/Taint. In other words, it is 28.77% more accurate than ANTaint and 39.21% more accurate than P/Taint for tracing sensitive data. Furthermore, in the micro-benchmark, CFTaint can accurately trace all data propagation of the specified field in most scenarios based on the field-based algorithm. However, due to the limitation of the field-based algorithm, CFTaint will be unable to trace data propagation when using a super-class type object as the source. Overall, CFTaint takes 96.13% less time than ANTaint and 98.15% less than P/Taint.

VII. Industry Application

The introduced field-based compositional static taint analysis approach is used in various industry scenarios of tracing sensitive data for change governance, data breach detection, and data consistency validation at Ant Group.

Change governance. Consider there is a database that contains a transaction history table, and the database will

be queried by multiple services more than 10 million times per day. The primary key *'transaction_id'* in the transaction history table is an 'int' type field, which will be auto-increased for each new transaction record. With the increasing transaction volume, an 'int' type *'transaction_id'* is not enough for future transactions (maximum 4,294,967,295 records in total). To address this issue, the database team needs to migrate the data to a new database table and update the data type of the *'transaction_id'* from 'int' to 'BigInt'. In this case, developers need to notify those micro-services that will query data from this table with the *'transaction_id'*, to make corresponding updates. Developers need to update the data type of parameter related to the *'transaction_id'* in the micro-services from 'int' to 'long' or 'BigInteger'. Otherwise, the overflow error will be triggered when loading the 'BigInt' type of data of the *'transaction_id'* field from the database. Manually identifying all these related parameters and all data flows is a complicated task, which is time-consuming due to the scale and complexity of the industry micro-services applications. Furthermore, although manually identifying is possible in some cases, the accuracy needs to be verified. For the site reliability requirement, to guarantee the reliability of the system, a verification mechanism needs to be used to verify these changes. Therefore, developers need to verify that all parameters related to the change are manually identified correctly and updated in the micro-services. In the above scenes, a scalable analysis tool that can trace the sensitive data or specified field in the large-scale complex micro-services is necessary for any change governance. The analysis can help identify and automatically verify all parameters related to the change in the micro-services.

VIII. Related Work

In the past, the software engineering community has contributed widely to taint analysis. The mainstream studies can be divided into two categories according to the analytical techniques: the static and dynamic taint analysis. We limited our discussion to the most related work of CFTaint.

Static taint analysis tools. Many related works on static taint analysis could be found. The IFDS framework [24] is one of the most popular frameworks used for solving inter-procedural, finite, distributive subset problems in a flow-sensitive and fully context-sensitive way. FlowDroid [10] is one of the best practices that target analyzing Android applications, and provide accuracy by performing context-sensitive, field-sensitive, and flow-sensitive analysis based on the IFDS framework. It provides precision in terms of context-sensitive, flow-sensitive, and field-sensitive. Although FlowDroid has the benefit of precision from the scalable point-to analysis and one-demand alias, scalability is still a challenge for FlowDroid [2] in the running on the microservices applications with millions of lines of the code. Massive context information needs to be maintained from the source for each function, and repeated analysis is required for the same function when used with different context information. ANTaint [2] is the previous static taint analysis tool used in Ant Group, which is built atop of FlowDroid. The main aim of ANTaint is to address the scalability challenge of static taint analysis on industrial micro-services applications by resolving the concrete problems in the existence of implicity dependency. However, because the unsound call-graph, which lacks tracing data propagations in framework behaviours, is provided for the inter-procedural analysis, many data propagation paths are missing in the taint analysis results. Furthermore, CHEX [25], LeakMiner [1], and SCanDroid [26] are tools for reasoning data flows that are similar to FlowDroid. Leakminer tries to solve the scalability challenges by using an incremental method to construct the call graph for applications. However, the analysis precision of such tools is lower than FlowDroid [2].

Datalog is another underlying technology that is used for taint analysis [27]. Livshits [28] proposed a taint analyzer that uses Datalog. However, the proposed approach uses Datalog without elements that unify pointer analysis. P/Taint [16] also uses Datalog for static taint analysis, but P/Taint combines Datalog with the information-flow and point-to analysis, which provides a more accurate analysis than Livishits' proposed approach. Although P/Taint can provide accurate taint analysis based on the underlying information-flow and point-to analysis, it is time costly to run on large-scale microservices. Furthermore, the program slicing or dependence [29] technique is also the favoured technique for static taint analysis. TAJ [30] is an example of a taint analyzer that uses program slicing for static taint analysis. TAJ will analyze the taint propagation based on the information-flow analysis based on the program slicing. However, it is limited in scalability and unsound for analyzing large-scale microservices.

IX. Conclusion

In this paper, we present CFTaint, a new field-based compositional static taint analyzer for scalable static sensitive data tracing on large-scale industrial micro-services applications. It uses the underlying techniques, such as the function summary, field-based algorithm, code transformation, and modelling supports, to perform precision and scalable dataflow tracing for taint analysis. As a proof of concept, based on the deployment in Ant Group, we validated this proposed approach. Based on the evaluation of the micro-benchmark provided by Wang et al. [2] and the evaluation of the production-benchmark, compared with ANTaint, CFTaint has significantly improved the accuracy and efficiency of taint analysis.

X. Acknowledgments

This work was supported by Ant Group through Ant Research Program. This research is partially supported by NSFC No.62002363.

REFERENCES

[1] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *Proceedings of the 2012 Third World Congress on Software Engineering*, ser. WCSE '12, 2012, p. 101–104.

[2] J. Wang, Y. Wu, G. Zhou, Y. Yu, Z. Guo, and Y. Xiong, "Scaling static taint analysis to industrial soa applications: A case study at alibaba," in *ESEC/FSE'20*, 2020, p. 1477–1486.

[3] P. Wang, W. J. Chao, K.-M. Chao, and C.-C. Lo, "Using taint analysis for threat risk of cloud applications," in *2014 IEEE 11th International Conference on e-Business Engineering*, 2014, pp. 185–190.

[4] R. Zhang, S. Huang, Z. Qi, and H. Guan, "Combining static and dynamic analysis to discover software vulnerabilities," in *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2011, pp. 175–181.

[5] Y. Feng, I. Dillig, S. Anand, and A. Aiken, "Apposcopy: Automated detection of android malware (invited talk)," in *Proceedings of the 2nd International Workshop on Software Development Lifecycle for Mobile*, ser. DeMobile 2014, 2014, p. 13–14. [Online]. Available: https://doi.org/10.1145/2661694.2661697

[6] X. Cui, J. Wang, L. C. K. Hui, Z. Xie, T. Zeng, and S. M. Yiu, "Wechecker: Efficient and precise detection of privilege escalation vulnerabilities in android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '15, 2015.

[7] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17, 2017, p. 71–85.

[8] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015*, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe

[9] I. Security and the Ponemon Institute, "Cost of a data breach report 2021," IBM Corporation, Tech. Rep., 2021, https://www.ibm.com/downloads/cas/OJDVQGRY.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, p. 259–269.

[11] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, "F4f: Taint analysis of framework-based web applications," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11, 2011, p. 1053–1068.

[12] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, 2015, p. 106–117.

[13] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.

[14] Z. Zhong, J. Liu, D. Wu, P. Di, Y. Sui, and A. X. Liu, "Field-based static taint analysis for industrial microservices," in *Proceedings of the 44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22)*, 2022.

[15] S. B. Andarzian and B. T. Ladani, "Compositional taint analysis of native codes for security vetting of android applications," in *2020 10th International Conference on Computer and Knowledge Engineering (ICCKE)*, 2020, pp. 567–572.

[16] N. Grech and Y. Smaragdakis, "P/taint: Unified points-to and taint analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133926

[17] SOFAStack, *SOFAStack*, 2021. [Online]. Available: https://www.sofastack.tech/en/

[18] S. Cloud, *Spring Cloud*, 2021. [Online]. Available: https://spring.io/projects/spring-cloud

[19] A. C. 3.0, *Camel 3.0*, 2021. [Online]. Available: https://camel.apache.org/

[20] Oracle, *GraalVM*, 2021. [Online]. Available: https://www.graalvm.org/

[21] *Engineering Report for the Scalable Compositional Static Taint Analysis for Sensitive Data Tracing on Industrial Micro-Services Research*, 2022. [Online]. Available: https://github.com/JasonZhongZexin/JasonZhongZexin.github.io/blob/main/Engineering_Report_for_the_Scalable_Compositional_Static_Taint_Analysis_for_Sensitive_Data_Tracing_on_Industrial_Micro_Services_Research.pdf

[22] R. Johnson, *Spring*, 2021. [Online]. Available: https://spring.io/

[23] *aliflow micro bench*, 2020. [Online]. Available: https://github.com/af-static-toolchains/aliflow-microbenchmark

[24] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995, p. 49–61.

[25] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012, p. 229–240.

[26] R. Spreitzer, G. Palfinger, and S. Mangard, "Scandroid: Automated side-channel analysis of android apis," in *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '18, 2018, p. 224–235.

[27] *Datalog*, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Datalog

[28] B. Livshits, "Improving software security with precise static and runtime analysis," Ph.D. dissertation, Stanford, CA, USA, 2006, aAI3242585.

[29] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81, 1981, p. 439–449.

[30] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: Effective taint analysis of web applications," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, 2009, p. 87–97.