Bridge the Islands: Pointer Analysis for Microservice Systems

TENG ZHANG, Nanjing University, China
YUFEI LIANG, Nanjing University, China
GANLIN LI, Nanjing University, China
TIAN TAN*, Nanjing University, China
CHANG XU, Nanjing University, China
YUE LI*, Nanjing University, China

Microservice architecture has revolutionized enterprise software, providing scalability and flexibility by decomposing applications into loosely coupled services. However, this paradigm shift introduces unique challenges for pointer analysis, a foundational static analysis crucial for supporting various client analyses. Existing fundamental analyses, primarily designed for monolithic enterprise applications, fall short in handling complex service communications—such as remote procedure call and message-based communication—and essential programming paradigms, like dependency injection and web endpoint configuration. This paper introduces MICANS, the first pointer analysis specifically crafted to address these challenges in microservice systems, capable of constructing comprehensive value flows across services. We extensively evaluated MICANS on real-world benchmarks from multiple domains, focusing on its effectiveness in resolving service communications, constructing essential program information like call graphs, and supporting client analyses such as taint analysis. MICANS consistently and significantly outperforms state-of-the-art approaches, demonstrating its capacity to handle complex cross-service communications and diverse programming paradigms. These results highlight MICANS' potential as a robust foundational analysis, advancing static analysis capabilities to meet the complexities of modern microservices.

CCS Concepts: • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: Static Analysis, Pointer Analysis, Microservice, Java

ACM Reference Format:
Teng Zhang, Yufei Liang, Ganlin Li, Tian Tan, Chang Xu, and Yue Li. 2025. Bridge the Islands: Pointer Analysis for Microservice Systems. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA023 (July 2025), 23 pages. https://doi.org/10.1145/3728896

1 Introduction

In the rapidly evolving software landscape, microservice has emerged as a leading practice, designed to accommodate the ever-increasing complexity of modern Java enterprise systems [JetBrains 2023]. By decomposing traditional monolithic applications into a collection of small, loosely coupled

*Corresponding authors.

Authors' Contact Information: Teng Zhang, dz21330044@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Yufei Liang, 602024330013@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Ganlin Li, 502022320006@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Tian Tan, tiantan@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Chang Xu, changxu@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Yue Li, yueli@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China.

ACM 2994-970X/2025/7-ARTISSTA023
https://doi.org/10.1145/3728896

services, the microservices enhance scalability, flexibility, and modularity [Fowler and Lewis 2014]. Many successful companies like Uber, Twitter, and Alibaba have adopted microservices to improve their agility and handle growing demands [Jackson 2022; Luo et al. 2021; Uber 2022].

In real-world enterprise practices, while the loosely coupled nature of microservice systems offers clear advantages, it also introduces complexity due to increased cross-process communications. Interactions between different modules within a large monolithic enterprise application, which were originally achieved through in-process method calls, are now orchestrated as cross-process communications between various small services, using Remote Procedure Call (RPC) and Message-based Communication (MBC). In such scenarios, a service typically relies on various other services developed by different teams or individuals. As a result, the developer invoking these services may not have detailed knowledge of their implementations. This lack of transparency poses challenges in achieving a comprehensive understanding of the entire system, which is crucial for tasks such as optimization, debugging, and vulnerability detection within the microservice systems.

Static analysis is widely recognized for providing information for understanding program behavior, greatly facilitating tasks like optimization, debugging, and vulnerability detection [Arzt et al. 2014; Grech and Smaragdakis 2017; Li et al. 2016; Toman and Grossman 2019; Zhang et al. 2014]. Pointer analysis, one of the most fundamental static analyses upon which virtually all others are built [Smaragdakis and Balatsouras 2015], computes the objects that each variable in the program may point to, enabling the derivation of information such as value flows and call graphs essential for various client analyses.

However, conducting pointer analysis for microservice systems is a complex task, as illustrated in Fig. 1. Specifically, service communication mechanisms, such as RPC and MBC, set each service apart from traditional monolithic applications, introducing unique challenges for pointer analysis. Moreover, effectively addressing these mechanisms also requires consideration of traditionally examined language features (e.g., reflection and native code) and various programming paradigms crucial to enterprise applications, such as dependency injection and web endpoint configuration, which remain underexplored in the existing literature. Additionally, these aspects are interconnected, meaning inadequate handling of one can adversely affect others. For example, insufficient analysis of service communications can lead to reduced code coverage and a lack of cross-service value flows, impacting the analysis of internal application code. Conversely, if the application code is not analyzed correctly, it can hinder the resolution of service communications. To the best of our knowledge, there is currently no pointer analysis algorithm capable of adequately addressing the complex features inherent in microservice systems. Furthermore, the development of such an algorithm that effectively balances soundness, precision, and efficiency in real-world microservice systems is even more challenging.

To date, several static analysis clients have been developed for microservice systems. However, unlike whole-program pointer analysis, which computes comprehensive foundational information such as value flows and call graphs for various potential clients, these clients employ different methodologies to address specific tasks, such as security analysis [Wang et al. 2020; Zhong et al. 2023] and code smell detection [Walker et al. 2020]. Additionally, they offer highly limited capabilities in resolving critical framework features of microservice systems, such as RPC, MBC, and dependency injection. Turning to pointer analysis, existing approaches (JackEE [Antoniadis et al. 2020] and Jasmine [Chen et al. 2022]) designed for enterprise applications [Oracle 2021] can comprehensively resolve language features like reflection and native code and handle programming paradigms in sophisticated frameworks like Spring Framework [Johnson et al. 2023a]. However, they cannot address service communications in microservice systems and fall short in managing the intricate value flows arising from complex usage scenarios involving dependency injection and web endpoint configuration. In this work, we introduce Micans, the first pointer analysis

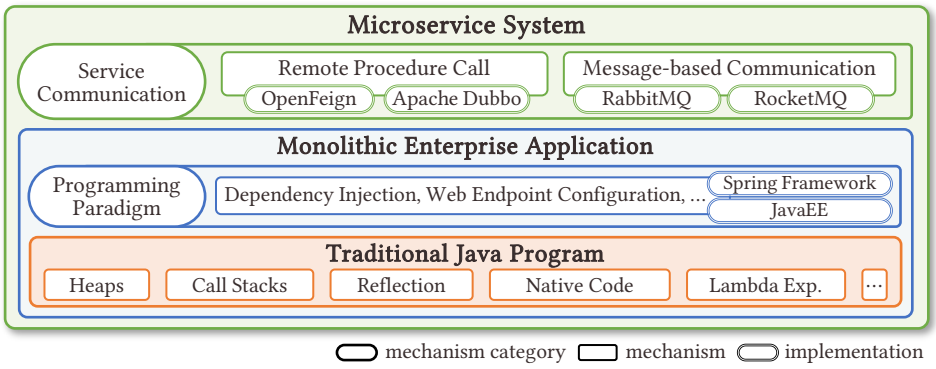

Fig. 1. Pointer Analysis Perspective for Microservice Systems

specifically for microservice systems. MICANS enables the analysis of complex service communication mechanisms that are intricately interwoven with advanced programming paradigms used in enterprise applications, effectively balancing soundness, precision, and efficiency in analyzing real-world microservice systems.

Specifically, in this work, we make the following contributions.

- We introduce MICANS, the first pointer analysis for microservice systems, capable of constructing comprehensive foundational information such as value flows and call graphs.
- We formalize MICANS by defining pointer analysis rules that cover critical service communication mechanisms, including RPC and MBC, which are widely adopted in microservice systems.
- We provide a comprehensive real-world open-source production benchmark suite, encompassing ten microservice systems across diverse business scenarios. We meticulously containerized these systems and provided additional test drivers, comprising over 4,000 lines of code for service orchestration and test scripts, facilitating quick deployment and promoting coverage of diverse microservice behaviors, which we believe will be useful for future research.
- We extensively evaluated MICANS on real-world benchmarks across multiple domains, focusing on its effectiveness in resolving service communications, constructing foundational program information, such as call graphs, and supporting clients like taint analysis. MICANS consistently outperformed state-of-the-art tools, demonstrating its capability to handle complex cross-service communications and diverse programming paradigms. For example, MICANS achieved an average recall rate of 85.53% in resolving call graph edges, compared to 53.85% for JackEE and 56.02% for Jasmine. For certain benchmarks such as *onemall*, this means that MICANS resolved an additional 62,357 real call graph edges that were missed by both other analyses. Moreover, MICANS identified *all* 335 taint flows, surpassing JackEE's 260 and Jasmine's 70, and effectively capturing intricate taint flows that the other tools could not resolve.
- We built MICANS on top of Tai-e [Tan and Li 2023], a state-of-the-art static analysis framework for Java, and have fully open-sourced both MICANS and the real-world microservice benchmarks as part of a publicly accessible artifact (see Section on Data Availability). MICANS will also be released and actively maintained on Tai-e.

2 Motivating Example

In this section, we use an example (simplified from real-world microservice systems) to briefly introduce key concepts and motivate our methodology. First, we describe the dynamic behavior of

microservice systems in this example. Next, we highlight the challenges traditional pointer analysis faces in such systems. Finally, we summarize how our approach addresses these challenges.

Overview. Microservices communicate through Remote Procedure Call (RPC) and Message-based Communication (MBC), often utilizing advanced programming paradigms, especially Dependency Injection (DI) and Web Endpoint Configuration (WEC), to improve flexibility and configurability in practical development. Like RPC and MBC, DI and WEC are supported by frameworks that automate their integration and usage. Specifically, DI handles the automatic instantiation and injection of objects at runtime, ensuring that the required dependencies are supplied to the appropriate fields and variables. WEC standardizes the configuration of methods that handle specific network requests.

Fig. 2 presents a simplified code snippet derived from real-world code, demonstrating a sensitive information leak. It shows the combined use of RPC, MBC, DI, and WEC, using OpenFeign, RabbitMQ, Spring IoC, and Spring Web, which are among the most popular implementations [Froeder et al. 2021; Johnson et al. 2023b,c; Klishin et al. 2021].

In this example, three microservices—`App1` (lines 1–10), `App2` (lines 11–19), and `App3` (lines 20–32)—communicate, forming the information flow from `App1` to `App2` via RPC (the invocation of `c.foo()` at line 5), and from `App2` to `App3` via MBC (the invocation of `rt.convertAndSend()` at line 18), as indicated by the arrows in Fig. 2. During this process, sensitive information (the user's password (`"password"`) at line 5 in `App1`) is propagated and eventually leaked at line 31 in `App3`.

In the following, we provide a detailed explanation of the two RPC and MBC calls involved in this leak, and then explain why traditional pointer analysis struggles to address these communications, leading security analysis clients to potentially miss the information leak at line 31.

Dynamic Behavior in RPC and MBC Invocations. We examine the RPC invocation at line 5 and the MBC invocation at line 18 to elucidate their dynamic behaviors during execution.

Remote Procedure Call (RPC). RPC is a communication protocol that allows one application to invoke a method in another application as if it were a local method, despite the target method being executed on a remote server. This abstraction enables seamless service communications in microservice systems. However, to enable remote invocation, certain mechanisms, such as stub objects, are necessary to handle the complexities of network communication.

In an RPC, the caller uses a stub object to invoke a method on the receiver (the RPC target) via a network request, enabling communication between two applications. The stub serves as a local proxy for the remote method, abstracting network transmission and request formatting. This process includes generating the stub, sending the network request, and invoking the receiver method. We demonstrate this with the invocation `c.foo("user","password")` (line 5).

- *Stub Object Creation* (Fig. 2(a)). At the RPC callsite `c.foo()` (line 5), `c` refers to a stub object generated and managed by DI. The stub object enables the caller to interact with the remote method as if it were local, handling the complexities of remote communication behind the scenes, such as marshaling parameters, sending network requests, and receiving responses. Specifically, when DI uses reflection to inspect the `App2Client` interface and finds it annotated with `@FeignClient` (line 7), it creates a dynamic proxy object (i.e., the stub) for the interface. This stub is then stored in the DI container, which manages objects created by DI. When DI detects that the field `c` is annotated with `@Autowired` (line 3), it retrieves the stub object from the DI container using type matching and injects it into the field `c` via reflection.
- *Request Sending* (Fig. 2(a)(b)). When `App1`'s `c.foo("user","password")` is invoked, it triggers the `foo` method on the stub object, which constructs and sends an HTTP request to `App2` in accordance with WEC annotations. Specifically, the `@FeignClient("App2")` annotation identifies the target application (`App2`), the `@GetMapping("/app2/foo")` annotation (line 8) specifies the

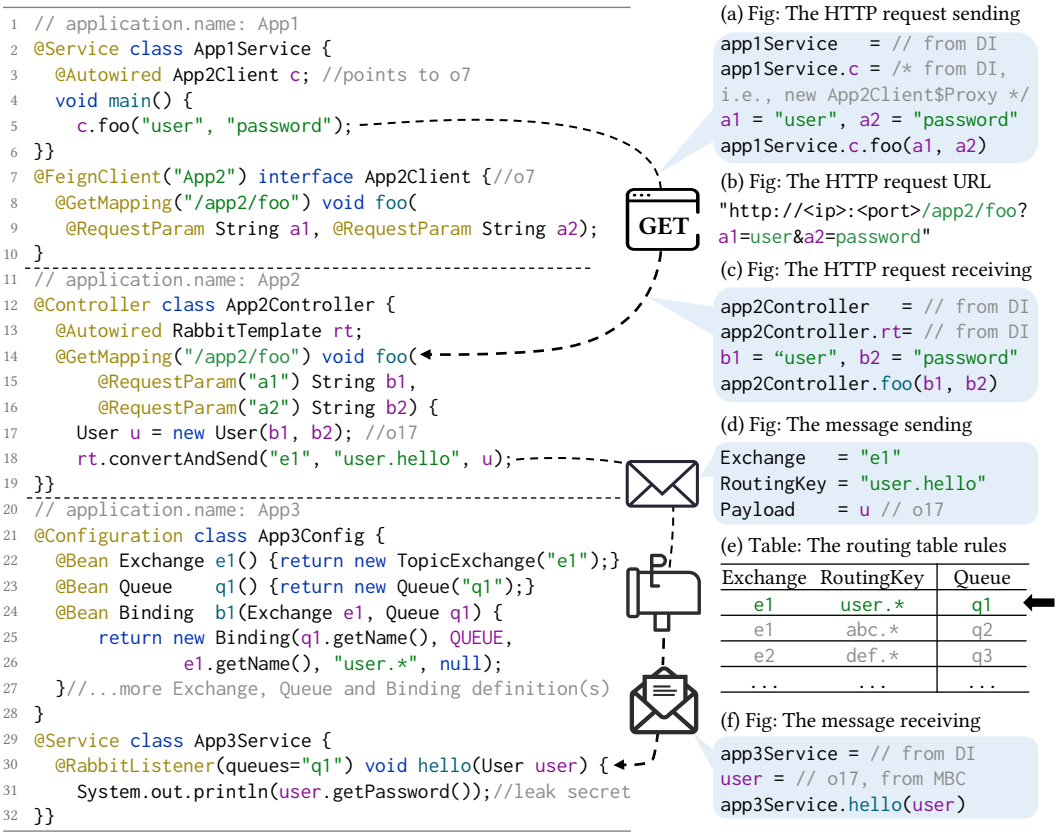

Fig. 2. A code example (left) illustrating RPC, MBC, DI, and WEC, with a runtime behavior description (right), presented in simplified Java code (excluding variable declarations) and highlighted in blue.

request path, and the `@RequestParam` annotations (line 9) map method parameters to request parameters. The resulting HTTP request is *"http://<ip>:<port>/app2/foo?a1=user&a2=password"*, where *<ip>* and *<port>* refer to App2's actual network address.

- *Receiver Method Invocation* (Fig. 2(c)). The receiver method for the HTTP request is `foo` in App2 (line 14), as its path is specified by the WEC annotation `@GetMapping("/app2/foo")`, which matches the request path. The WEC mechanism routes the request to `foo`. Additionally, the `@RequestParam("a1")` and `@RequestParam("a2")` annotations bind the request parameters `"user"` and `"password"` to the method parameters `b1` and `b2`, respectively.

Message-based Communication (MBC). MBC is an asynchronous communication pattern where services exchange information by sending messages through a messaging system. Unlike RPC, which directly invokes remote methods, MBC decouples services by routing messages through exchanges and queues, allowing for more flexible communication between microservices. Messages are sent to an exchange, routed based on predefined rules, and received by a listening service.

At an MBC callsite, the sender sends a message containing routing information, which is directed to the receiver method according to the routing table, a set of predefined routing rules. This process involves sending the message, creating the routing table, and invoking the receiver method. We illustrate this with the invocation `rt.convertAndSend("e1","user.hello",u)` (line 18).

- *Message Sending* (Fig. 2(d)). At line 13, the field `rt`, injected by DI via `@Autowired`, is of type `RabbitTemplate`, whose method `convertAndSend` sends messages. The message at line 18 is constructed from the exchange name `"e1"`, the routing key `"user.hello"`, and the payload `o17`. The exchange name and routing key determine the routing of the message, as explained below.
- *Routing Table Creation* (Fig. 2(e)). In MBC, routing table maps an exchange name and routing key to a message queue. This is achieved through a `Binding` object, created at line 25, which links the exchange `"e1"`, the routing key `"user.*"`, and the queue `"q1"`. The `Exchange` and `Queue` objects are instantiated and injected by DI (lines 22–23), forming the routing table (Fig. 2(e)).
- *Receiver Method Invocation* (Fig. 2(f)). After routing, the message is sent to the queue(s) matching the exchange name and routing key. In this case, the message with exchange `"e1"` and routing key `"user.hello"` is routed to queue `"q1"` due to the wildcard match `"user.*"` in the routing table (Fig. 2(e)). The receiver method `hello` in `App3` listens to this queue (`@RabbitListener(queues="q1")` at line 30), and upon receiving the message, passes the payload `o17` to the parameter `user`. At line 31, the user's password is printed, causing a sensitive information leak.

Challenges for Traditional Pointer Analysis. The main challenges posed by microservice systems for pointer analysis stem from their complexity, particularly the *dynamic nature of various mechanisms and their interdependencies*. These mechanisms rely heavily on annotations, configuration, reflection, and dynamic proxy to customize runtime behavior, which undermines the effectiveness of pointer analysis. For example, the RPC mechanism dynamically generates proxy objects that transform local method calls into remote procedure invocations. In `App1` (line 5), the method call `c.foo()` does not resolve to a local method but to a dynamically generated method that triggers an HTTP request to `App2`. Furthermore, these mechanisms are often interwoven. For instance, the dispatch and targeting of RPC requests depend on metadata from WEC annotations (e.g., HTTP paths), while MBC message routing is based on routing table rules constructed through DI. If the analysis of one mechanism is incomplete or inaccurate, it can impact the analysis of others.

Current state-of-the-art tools for analyzing enterprise applications, such as JackEE and Jasmine, are primarily designed for monolithic applications and are ineffective at analyzing complex microservice systems like the example presented. *These tools have two major limitations: they completely overlook the intricate and diverse communication mechanisms in microservices, which involve various types of network requests, and they have limited capabilities in handling DI and WEC, failing to address their usage in microservice systems.* Each limitation is discussed in detail below.

First, with respect to microservice communication mechanisms, existing tools fail to capture value and control flows across services. For example, when analyzing the RPC call at line 5, they are unable to detect the remote invocation and the passing of arguments to the target method `foo` in `App2` (line 14). A similar limitation arises when analyzing the MBC call at line 18.

Second, these tools are inadequate in their analysis of DI and WEC, failing to properly account for their usage in microservice systems. For instance, their handling of DI is incomplete, as they overlook object injections for methods annotated with `@Bean`, a common pattern in DI. This shortcoming is evident in the construction of the `Binding` object at line 24 in Fig. 2, where JackEE and Jasmine ignore the objects created at lines 22 and 23. Consequently, the `Binding` object misses the values `"e1"` and `"q1"`, which are crucial for constructing the routing rules.

These combined limitations prevent current tools from detecting the information leak in Fig. 2.

Our Solution. In this work, we propose MICANS, the first pointer analysis designed for microservice systems. MICANS provides a comprehensive analysis of the entire microservice system, effectively handling complex service communication mechanisms and offering thorough support for

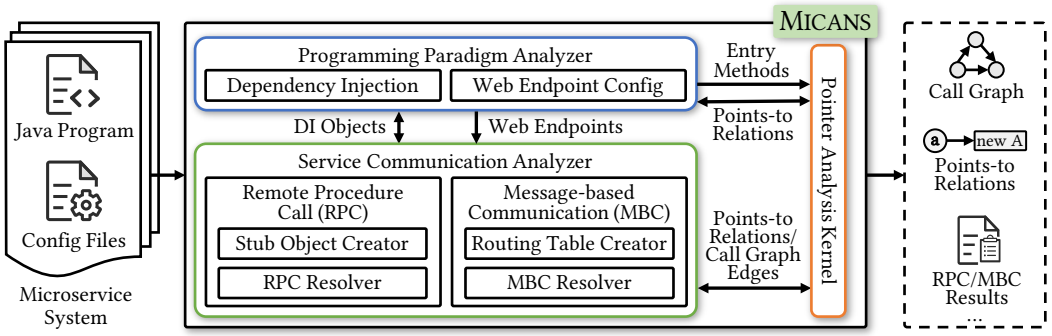

Fig. 3. The overview of MICANS.

advanced programming paradigms. It integrates these features to perform on-the-fly pointer analysis across the entire microservice system. As a result, MICANS is capable of thoroughly analyzing the system's behavior and detecting issues such as the information leak in the example.

3 Methodology

Fig. 3 provides an overview of MICANS, which takes microservice programs and configuration files as input and produces a call graph, points-to relations, and the resolved service communications (RPC and MBC) within the programs. MICANS consists of three key components: a programming paradigm analyzer, a service communication analyzer, and a pointer analysis kernel. The two analyzers interact with the pointer analysis kernel on the fly. The programming paradigm analyzer handles DI and WEC, identifying entry methods and providing the corresponding points-to relations for the pointer analysis kernel. Using this information, the pointer analysis kernel analyzes each service, building the intra-service call graph and points-to relations while identifying RPC and MBC callsites. The service communication analyzer then resolves these callsites to construct cross-service call graph edges, which are fed back into the pointer analysis kernel.

The key innovation of MICANS lies in its design. First, the two analyzers are purpose-built to handle DI, WEC, RPC, and MBC, drawing on developer experience and our extensive study to cover the most common and essential features in microservice systems. Second, the analyzers' inputs and outputs are expressed in terms of points-to relations and call graph edges, enabling seamless interaction with the pointer analysis kernel. This design allows MICANS to manage the complex interdependencies between these features, resulting in a more complete call graph. This comprehensive call graph provides a stronger foundation for downstream analysis clients, such as security analysis, for microservice systems.

Below, we present the programming paradigm and service communication analyzers in MICANS. These analyzers are complex due to the intricate microservice features they address; however, given space limitations, we focus only on their most essential aspects.

3.1 Programming Paradigm Analyzer

3.1.1 DI Analysis. As described in Section 2, DI is a mechanism that automatically manages the creation and injection of specific objects at runtime. This process is highly dynamic, as DI frameworks typically rely on reflection, annotations, and configuration files, posing significant challenges for pointer analysis. To effectively handle DI, our DI analysis does not analyze the DI framework code directly; instead, it models the behaviors and side effects of the DI mechanism.

This analysis involves two main tasks: (1) creating DI objects and (2) injecting these created objects into the appropriate fields and variables (referred to as DI pointers).

DI Object Creation. DI frameworks typically create objects specified by annotations, either on the methods that instantiate new objects (e.g., methods annotated with `@Bean`) or on the classes of objects to be created (e.g., classes annotated with `@Service` and `@Controller`). Fig. 2 illustrates both approaches. MICANS scans the program to identify these annotated methods and classes. For annotated methods, MICANS analyzes them to create the required objects; for annotated classes, MICANS generates mock objects. All DI objects are then stored in a DI container within MICANS, simulating the DI container created at runtime.

DI Object Injection. The DI objects created during analysis need to be injected into the corresponding DI pointers. MICANS first scans the program to identify the DI pointers, including fields annotated with `@Autowired`, parameters of methods annotated with `@Bean`, and other injection points supported by the DI frameworks. MICANS then selects the appropriate DI objects from the DI container based on the constraints defined by the DI pointers (for instance, using the `@Qualifier` annotation to specify the id of DI object) and adds them to points-to sets of the DI pointers. For instance, at line 24 of Fig. 2, MICANS adds the objects created at lines 22 and 23 into points-to sets of parameters `e1` and `q1`, respectively, accurately modeling the behavior of DI object injection.

3.1.2 WEC Analysis. The WEC mechanism determines which methods are invoked in response to network requests. These methods, known as receiver methods or web endpoints, process incoming requests but typically lack explicit callsites in the program, as they are invoked by the WEC framework. To analyze these receiver methods, WEC analysis identifies them, marks them as entry methods, and notifies the pointer analysis kernel so they can be reached and analyzed.

Additionally, network requests often carry metadata that determines request routing. WEC analysis collects this request metadata and associates it with the relevant receiver methods, which will be utilized by the service communication analyzer as discussed in Section 3.2.

3.2 Service Communication Analyzer

3.2.1 RPC Analysis. RPC Analysis constructs cross-service call graph edges and manages argument passing for RPC calls. It consists of two components: the *Stub Object Creator*, which generates stub objects, and the *RPC Resolver*, which models both the network request transmission and the target method invocation at the RPC callsite. We illustrate these components using the RPC example in Fig. 2, based on OpenFeign [Froeder et al. 2021] (though the same principles apply to other RPC frameworks).

Stub Object Creator. The Stub Object Creator identifies stub classes (e.g., classes annotated with `@FeignClient`) and generates stub objects that act as proxies created by the RPC framework, helping RPC analysis recognize RPC calls where the receiver is a stub object. During stub creation, the Stub Object Creator also collects network request metadata specified in the stub class annotations, maps it to methods within the class, and associates this mapping with the created stub object. This mapping is later used by the RPC Resolver. For example, when analyzing the stub class at line 7 in Fig. 2, the Stub Object Creator generates a stub object and associates request metadata, such as `"App2"` and `"/app2/foo"` from the annotations on method `foo`, with `foo`. These generated stub objects are then managed in the DI container and injected into the appropriate pointers.

RPC Resolver. The RPC Resolver is activated when MICANS identifies an RPC. It uses the network request metadata associated with the stub object, along with endpoint information (e.g., request metadata) provided by WEC analysis, to identify the matching receiver method for the RPC. Once the

RPC Resolver finds a receiver method m for a callsite c, it constructs a cross-service call graph edge from c to m, and propagates the points-to sets of c's arguments to the corresponding parameters of m. For example, for the callsite `c.foo` in `App1Service` (line 5 of Fig. 2), the RPC Resolver identifies the target as the `foo` method in `App2Controller` (line 14) by matching request metadata from the stub object (line 8) with the receiver method's metadata (line 14). It then constructs the call graph edge from `c.foo` to `foo` and propagates the arguments `"user"` and `"password"`.

3.2.2 MBC Analysis. MBC Analysis, similar to RPC Analysis, handles MBC invocations to construct cross-service call graph edges and manage points-to relations for message routing. It consists of two components: the *Routing Table Creator*, which establishes routing rules in the routing table, and the *MBC Resolver*, which models message routing and receiver method invocation at the MBC callsite. We illustrate these components using the MBC example in Fig. 2, based on RabbitMQ [Klishin et al. 2021] (though the same principles apply to other MBC frameworks).

Routing Table Creator. The Routing Table Creator collects routing information from the program and builds the routing table by modeling the behavior of the MBC framework. In RabbitMQ, each routing rule consists of three string values: exchange name, routing key, and queue name, derived from the fields of a `Binding` object. The Routing Table Creator retrieves points-to information for these fields from the pointer analysis kernel to establish routing rules. In practice, the exchange and queue names used in `Binding` objects are often sourced from fields in `Exchange` and `Queue` objects (e.g., lines 25–26), with DI analysis and the pointer analysis kernel managing their creation and passing. Since multiple `Binding` objects may be created at the same allocation site, MICANS employs context sensitivity for these objects to differentiate their contents for better precision.

MBC Resolver. The MBC Resolver is activated when MICANS identifies an MBC callsite (for RabbitMQ, this is the `convertAndSend` method on a `RabbitTemplate` object). The MBC Resolver constructs messages from the callsite arguments, and models message routing. Each message includes the exchange name, routing key, and payload. Using the routing table, the MBC Resolver identifies the target message queue and all associated receiver methods, which are marked by the `@RabbitListener` annotation. The MBC Resolver then establishes call graph edges from the MBC callsite to these receiver methods and propagates message payloads to them. For example, for the callsite `rt.convertAndSend` in `App2Controller` (line 18 in Fig. 2), the MBC Resolver identifies the target as the `hello` method in `App3Service` (line 30) by matching the message's exchange name `"e1"` and routing key `"user.hello"` with the first rule in the routing table (Fig. 2(e)). Since multiple messages may be sent from the same MBC callsite, MICANS applies context sensitivity at the callsite to distinguish different messages for better precision.

4 Formalism

We formalize MICANS as introduced in Section 3. The analysis logic of MICANS is complex, reflecting the intricate microservice features it handles. Due to space constraints, we give only the core rules of its two new components, the programming paradigm analyzer (Section 4.1) and service communication analyzer (Section 4.2), in a context-insensitive form. The implementation, however, employs a context-sensitive pointer analysis [Ma et al. 2023; Smaragdakis and Balatsouras 2015] for enhanced precision. Additionally, we present our rules with several helper functions, briefly describing their inputs and outputs while omitting internal details due to space. MICANS has been released as open source, providing full transparency and accessibility to all implementation details.

Fig. 4 presents the domains and notations used in this section, most of which are self-explanatory. Additionally, we define two special objects: DI object o_{DI} and RPC stub object o_{stub}, as introduced in Sections 3.1.1 and 3.2.1. Here, m_{ret} and m_{p0} represent the *return* and *this* variable of method m,

$$\begin{array}{cccccc} \text{instruction labels} & i, j \in \mathbb{L} & \text{variables} & x, y, m_{\text{ret}}, a_k, m_{pk} \in \mathbb{V} & \text{objects} & o, o_{\text{DI}}, o_{\text{stub}} \in \mathbb{O} \\ \text{class types} & t \in \mathbb{T} & \text{methods} & m \in \mathbb{M} & \text{fields} & f \in \mathbb{F} \\ \text{pointers} & x, o.f \in \mathbb{P} = \mathbb{V} \cup (\mathbb{O} \times \mathbb{F}) & \text{points-to relations} & pt : \mathbb{P} \rightarrow \mathcal{P}(\mathbb{O}) & & \end{array}$$

Fig. 4. Domains and notations used in formalism.

$$\frac{t \in \widehat{DIClasses} \quad m = getDIConstructor(t)}{\begin{array}{c} mockDI(t) \in DIContainer \\ m \in InjectedMethods \end{array}} \text{[DiCre-Clz]} \qquad \frac{m \in \widehat{DIMethods} \quad o_{\text{DI}} \in pt(m_{\text{ret}})}{\begin{array}{c} o_{\text{DI}} \in DIContainer \\ m \in InjectedMethods \end{array}} \text{[DiCre-Meth]}$$

$$\frac{\begin{array}{c} o_{\text{DI}} \in DIContainer \quad t = type(o_{\text{DI}}) \\ f \in getInjectedFields(t) \\ o'_{\text{DI}} \in getDIObjs(f) \end{array}}{o'_{\text{DI}} \in pt(o_{\text{DI}}.f)} \text{[DiInj-Field]} \qquad \frac{\begin{array}{c} m \in InjectedMethods \\ 0 \leq k \leq n \\ o_{\text{DI}} \in getDIObjs(m_{pk}) \end{array}}{o_{\text{DI}} \in pt(m_{pk})} \text{[DiInj-Param]}$$

Fig. 5. Rules for DI analysis (DI object creation and injection).

while a_k and m_{pk} represent the k-th (k > 0) argument and parameter of a method call. A symbol with a hat (e.g., $\widehat{S}$) indicates a set of annotated or configured program elements, such as classes.

4.1 Programming Paradigm Analyzer

We formalize the programming paradigm analyzer introduced in Section 3.1. WEC analysis, which primarily identifies network receiver methods and gathers metadata to produce *WebEndpoints* as input for RPC analysis, is relatively straightforward. For brevity, we omit the rules for WEC analysis and present only the rules for DI analysis below.

Fig. 5 formalizes the DI analysis introduced in Section 3.1.1. To facilitate this analysis, we define two sets: *DIContainer*, which holds DI objects managed by the analysis, and *InjectedMethods*, which contains methods with parameters requiring injection of DI objects.

DI Object Creation. Rules [DiCre-Clz] and [DiCre-Meth] specify how Micans creates and manages DI objects for classes ($\widehat{DIClasses}$) and methods ($\widehat{DIMethods}$) based on annotations (e.g., `@Service` for classes and `@Bean` for methods) and configuration settings. Rule [DiCre-Clz] uses helper functions $mockDI(t)$ to create a mock object of each class t in $\widehat{DIClasses}$ and $getDIConstructor(t)$ to retrieve t's proper constructor, such as the primary or annotated (e.g., by `@Autowired`) constructor. Rule [DiCre-Meth] collects objects returned by methods in $\widehat{DIMethods}$ as DI objects. These DI objects are then added to the *DIContainer*. Additionally, Micans includes DI-related methods, such as annotated methods and constructors requiring injection, in *InjectedMethods*, which are also added to the set of reachable methods in the pointer analysis kernel (omitted here for brevity).

DI Object Injection. Rules [DiInj-Field] and [DiInj-Param] specify how Micans injects DI objects into fields and method parameters, respectively. In [DiInj-Field], Micans scans all DI objects in the container to identify fields requiring injection. Key helper functions support this process: $getInjectedFields(t)$ retrieves annotated fields in class t (e.g., fields annotated with `@Autowired` or `@Resource`); $type()$ retrieves an object's type; and $getDIObjs()$ selects matching DI objects from the container based on the declaring type of the input element (e.g., a field or variable). In [DiInj-Param], Micans injects DI objects into the appropriate method parameters, including the *this* variable.

4.2 Service Communication Analyzer

4.2.1 RPC Analysis. Fig. 6 present the rules for RPC analysis as introduced in Section 3.2.1. We define *MetaMapping*, which maps each stub object to pairs of methods and their associated request

$$\frac{t \in \widehat{StubClasses} \quad m \text{ is declared in } t \quad o_{\text{stub}} = mockStub(t)}{o_{\text{stub}} \in DIContainer \quad \langle m, build(m,t)\rangle \in MetaMapping(o_{\text{stub}})} \text{ [StubObjCreation]}$$

$$\frac{\begin{array}{c} i: y = x.c(a_1,...,a_n) \quad o_{\text{stub}} \in pt(x) \quad m = dispatch(o_{\text{stub}}, c) \\ \langle m, meta\rangle \in MetaMapping(o_{\text{stub}}) \quad \langle m', meta'\rangle \in WebEndpoints \quad match(meta, meta') \end{array}}{\langle i, m, meta, m', meta'\rangle \in RPCEdges} \text{ [RpcResolution-1]}$$

$$\frac{\begin{array}{c} i: y = x.c(a_1,...,a_n) \quad \langle i, m, meta, m', meta'\rangle \in RPCEdges \\ \langle a_j, m'_{pk}\rangle \in getRelation(meta, meta') \quad o \in pt(a_j) \quad o' \in pt(m'_{\text{ret}}) \end{array}}{getDIObjs(m'_{p0}) \in pt(m'_{p0}) \quad o \in pt(m'_{pk}) \quad o' \in pt(y)} \text{ [RpcResolution-2]}$$

Fig. 6. Rules for RPC analysis.

metadata. Additionally, *RPCEdges* consists of tuples that include an RPC callsite, the local method (on the stub object), the remote receiver method (target method), and the metadata linking them.

Stub Object Creation. Rule [StubObjCreation] specifies how Micans creates stub objects and gathers request metadata from stub class annotations. For each stub class (e.g., annotated with `@FeignClient`), Micans creates a stub object o_{stub}, storing it in *DIContainer* for injection into appropriate variables. Key helper functions include *mockStub*, which generates a mock stub object of type t, and $build(m, t)$, which constructs request metadata for method m in stub class t, including request type, path, and parameters. [StubObjCreation] also generates request data mappings for o_{stub}.

RPC Resolution. Rules [RpcResolution-1] and [RpcResolution-2] define how Micans identifies and resolves RPC calls to construct cross-service call graph edges and manage argument passing. In [RpcResolution-1], Micans identifies the RPC callsites (where the receiver variable points to a stub object o_{stub}), and uses the request metadata from *MetaMapping*, along with the web endpoints (*WebEndpoints*) provided by WEC analysis, to locate the matching receiver method and construct an RPC edge, stored in *RPCEdges*. The helper function *match*() determines whether the two given request metadata match by comparing their request types and paths. Rule [RpcResolution-2] manages argument and return value passing along the constructed RPC edges. The function $getRelation(meta, meta')$ establishes the correspondence between RPC callsite arguments and receiver method parameters, indicating that objects pointed to by argument a_j at the callsite flow to parameter m_{pk} in the receiver method.

4.2.2 MBC Analysis. Fig. 7 present the rules for MBC analysis as introduced in Section 3.2.2.

Routing Table Creation. Rule [RoutingTableCreation] specifies how Micans constructs routing table rules based on values collected from the arguments for object creation (e.g., arguments of `new Binding()`). The function *getStr*() retrieves the string content from a string object. The collected strings from parameters $\langle a_{\text{exch}}, a_{\text{key}}, a_{\text{queue}}\rangle$ are stored as a rule in *RoutingTable*, allowing Micans to accurately represent and maintain the routing table during analysis.

MBC Resolution. Rule [MbcResolution] defines how Micans identifies and resolves MBC calls, handling argument passing by modeling message-sending behavior. For an MBC call $x.send(a_{\text{exch}}, a_{\text{key}}, payload_1, \ldots, payload_n)$ (where *send* is a predefined message-sending method), Micans searches *RoutingTable* for the queue matching the given exchange and routing key. Specifically, the helper function $match(str_{\text{key}}, str'_{\text{key}})$ verifies if a message's routing key str_{key} matches the routing table's wildcard routing key str'_{key}. Then, using function $listen(str_{\text{queue}})$, Micans identify receiver methods

$$\frac{\begin{array}{c} new\ Binding(a_{\text{queue}}, a_{\text{exch}}, a_{\text{key}}) \quad o_{\text{q}} \in pt(a_{\text{queue}}) \quad o_{\text{e}} \in pt(a_{\text{exch}}) \quad o_{\text{k}} \in pt(a_{\text{key}}) \\ str_{\text{queue}} = getStr(o_{\text{q}}) \quad str_{\text{exch}} = getStr(o_{\text{e}}) \quad str_{\text{key}} = getStr(o_{\text{k}}) \end{array}}{\langle str_{\text{exch}}, str_{\text{key}}, str_{\text{queue}} \rangle \in RoutingTable} \text{ [RoutingTableCreation]}$$

$$\frac{\begin{array}{c} i : x.send(a_{\text{exch}}, a_{\text{key}}, payload_1, ..., payload_{\text{n}}) \quad o_{\text{e}} \in pt(a_{\text{exch}}) \quad o_{\text{k}} \in pt(a_{\text{key}}) \\ str_{\text{exch}} = getStr(o_{\text{e}}) \quad str_{\text{key}} = getStr(o_{\text{k}}) \quad \langle str_{\text{exch}}, str'_{\text{key}}, str_{\text{queue}} \rangle \in RoutingTable \\ match(str_{\text{key}}, str'_{\text{key}}) \quad m \in listen(str_{\text{queue}}) \quad 1 \le k \le n \quad o \in pt(payload_k) \end{array}}{\langle i, m \rangle \in MBCEdges \quad getDIObjs(m_{p0}) \in pt(m_{p0}) \quad o \in pt(m_{pk})} \text{ [MbcResolution]}$$

Fig. 7. Rules for MBC analysis.

(e.g., methods annotated with `@RabbitListener`) listening to the queue, passing each $payload_k$ to the corresponding parameter m_{pk} of these receiver methods.

5 Evaluation

In this section, we investigate the following research questions for evaluating Micans.

RQ1. How effective is Micans in resolving service communication mechanisms (e.g., RPC and MBC) for real-world microservice systems?

RQ2. Compared to SOTAs, How well does Micans perform in constructing fundamental information such as call graph edges and reachable methods for real-world microservice systems?

RQ3. Compared to SOTAs, How well does Micans support analysis clients, e.g., taint analysis?

5.1 Experimental Setup

We have implemented Micans as a stand-alone open-source tool, building on the Tai-e framework [Tan and Li 2023], a state-of-the-art static analysis framework for Java that offers powerful and highly extensible pointer analysis, which significantly facilitate the development of our analysis. All experiments were conducted in a Docker environment on a machine equipped with two Intel(R) Xeon(R) Gold 6430 CPUs @ 2.10GHz (32 cores) and 512GB of 4800MT/s RDIMM RAM. All evaluation-related materials, including source code, binary artifacts, orchestration scripts, and test scripts, have been made publicly available in the artifact (see Section on Data Availability).

Benchmarks. Preparing a real-world benchmark for microservice systems is a complex and time-intensive undertaking [Aderaldo et al. 2017]. Successfully deploying these systems necessitates the coordination of service dependencies, management of configurations, network setup, and data storage. Even minor configuration errors can lead to instability or service failures, complicating the deployment process. Furthermore, validating static analysis requires dynamic data from microservice systems. High-quality test drivers that comprehensively cover system behaviors are essential but also demand a deep understanding of the internal logic of each service and the interactions among services. Challenges like deployment and test driver creation impede the construction of real-world benchmarks. Consequently, while existing handcrafted benchmarks [Gan et al. 2019; Zhou et al. 2018] effectively showcasing basic microservice functionalities, they fall short in capturing the diversity and complexity of real-world modern microservice scenarios.

As shown in Table 1, we have provided a real-world benchmark consisting of ten popular open-source microservice systems. The benchmarks average 4.8K stars on platforms like GitHub and span diverse business domains such as cloud storage, blogging, e-reading, e-commerce, and online judge. Additionally, we offer substantial program size metrics, with an average of 49.8K lines of application code and 3.7 million lines of intermediate representation (IR) code (including library code). We containerized these systems using Docker and crafted test drivers comprising over 4,000

Table 1. Real-world microservice benchmarks. #App LoC: lines of application code. #Lib Classes (excl. JDK): number of library classes (excluding JDK). #IR LoC: lines of intermediate representation (IR) code.

Benchmark	#Stars	Business Type	#App LoC	#App Classes	#Lib Classes (excl. JDK)	#IR LoC
netdisk	1.5k	Cloud Drive	26.9k	441	41790	2.35M
xmall	7.2k	E-shop	48.1k	257	49419	2.33M
onemall	17.1k	E-commerce	68.2k	894	63098	3.98M
mogu	1.7k	Blog	45.2k	470	80567	4.27M
basemall	9k	Supply Chain System	78.1k	1052	68751	3.07M
youlai	2.2k	E-shop	26.8k	423	96888	6.88M
novel	1.1k	E-reading	20.2k	170	67424	4.33M
sduoj	507	Online Judge	27.6k	477	55638	1.92M
roncoo	1.4k	E-education	111.9k	1310	72424	4.75M
mall4cloud	6.5k	E-commerce	45.1k	538	71193	3.78M

lines of code for service orchestration and testing scripts. This setup facilitates quick deployment and dynamic information collection, which we expect to be useful for future research.

5.2 RQ1 — Effectiveness of MICANS in Resolving Service Communications

In this section, we evaluate the effectiveness of MICANS in resolving the most critical feature in microservice systems: service communication mechanisms, specifically RPC and MBC, as addressed in this paper. To the best of our knowledge, existing works either lack support for service communication resolution [Antoniadis et al. 2020; Chen et al. 2022] or cannot be compared because they are proprietary internal tools not available as open-source [Wang et al. 2020; Zhong et al. 2023]. The only exception is a code smell detection tool [Walker et al. 2020]; however, upon inspecting its source code and conducting further experiments, we discovered that it only handles outdated RPC implementations that are absent from recent real-world microservice systems. Consequently, it resolves none of the RPC callees in our benchmark. Therefore, in our RQ1 experiments, we focus solely on assessing MICANS' capability to resolve service communications, evaluating both RPC and MBC, as detailed below.

5.2.1 Remote Procedure Call (RPC). Resolving an RPC entails solving the following tuple: $\langle callsite, callee, [\langle arg_i, param_j \rangle, ...] \rangle$. To resolve RPC, a static analysis first reaches an RPC callsite, then resolves the corresponding RPC callee for the callsite, and finally constructs the value flow from the arguments at the callsite to the corresponding parameters at the callee.

Table 2 presents MICANS' results in resolving RPC communication. The "#GT" column represents the number of real RPC tuples involved in the microservice system. We carefully examined the source code of all microservice systems to manually collect the RPC tuples, which served as the ground truth for our experiment. The "#TP" column (true positive) denotes the number of real RPC tuples reported by MICANS while the "#FP" indicates the number of false positive.

Understanding the Results. As shown in Table 2, MICANS achieves excellent soundness and precision, with an average recall rate ("#TP"/"#GT") of 94.9% and an average precision rate ("#TP"/"#TP+#FP") of 99.8%. These promising results are attributable to MICANS' effectiveness in addressing the three key aspects of RPC resolution: reaching RPC callsites, resolving RPC targets, and passing parameters, as detailed below.

Table 2. MICANS' resolution result of service communications (RPC and MBC). #GT represents the ground truth number of real service communications in each benchmark. #TP(#FP) indicates the true (false) positives reported by MICANS. Recall is the recall rate (#TP/#GT), and Prec. is the precision rate (#TP/(#TP+#FP)).

Benchmark	Remote Procedure Call (RPC)					Message-based Communication (MBC)								
	#GT	Context Insensitivity				#GT	Context Insensitivity				Context Sensitivity			
		#TP	#FP	Recall	Prec.		#TP	#FP	Recall	Prec.	#TP	#FP	Recall	Prec.
netdisk	131	126	0	96.2%	100%	-	-	-	-	-	-	-	-	-
xmall	158	153	0	96.8%	100%	4	0	0	0%	0%	0	0	0%	0%
onemall	164	159	0	97.0%	100%	3	2	1	66.6%	66.6%	2	1	66.6%	66.6%
mogu	52	52	0	100%	100%	9	9	18	100%	33.3%	9	0	100%	100%
basemall	71	67	1	94.4%	98.5%	27	22	5457	81.5%	0.4%	27	0	100%	100%
youlai	16	14	0	87.5%	100%	1	0	1	0%	0%	0	1	0%	0%
novel	14	14	0	100%	100%	2	2	0	100%	100%	2	0	100%	100%
sduoj	60	57	0	95.0%	100%	4	4	0	100%	100%	4	0	100%	100%
roncoo	55	55	0	100%	100%	-	-	-	-	-	-	-	-	-
mall4cloud	57	47	0	82.5%	100%	5	5	0	100%	100%	5	0	100%	100%
AVERAGE	77.8	74.4	0.1	94.9%	99.8%	7	5.5	685	68.5%	50.1%	6.1	0.25	70.8%	70.8%

For soundness, comprehensive call graphs are crucial for reaching RPC callsites. This requires a thorough handling of both the underlying language features (e.g., reflection) and programming paradigms (e.g., dependency injection and web endpoint configuration). Pointer analysis offers an effective solution by resolving these aspects on-the-fly, enabling extensive coverage of program behavior and thereby reaching more RPC callsites. Furthermore, as detailed in Section 3, MICANS makes concerted efforts to resolve service communications (RPC and MBC) and integrate related value flows, further expanding the coverage of RPC callsites. This positive feedback loop progressively enhances the soundness of the analysis. Note that resolving RPC is crucial for static analysis, as it can reveal numerous actual program behaviors. For instance, in the *basemall* benchmark, MICANS resolves 18,060 real call graph edges through call chains originating from a single RPC call site invoked at runtime.

However, due to the complexity of real-world microservice systems, there are scenarios that MICANS is currently unable to handle. For instance, the lack of support for lifecycle processes in the Spring Framework—such as post-initialization—results in certain RPC callsites being unreachable in the *sduoj* benchmark. Additionally, in the *basemall* benchmark, an RPC callsite's argument type and its callee's parameter type exhibit a non-subtyping relationship, which is uncommon. While the RPC mechanism accommodates this due to the serialization and deserialization processes, MICANS currently does not address such translations, leading to a few unresolved cases.

Regarding precision, we note that developers often follow pattern-based practices when implementing RPC, unlike MBC. Consequently, when conducting pointer analysis, we take full advantage of these patterns to precisely resolve RPC targets and manage the abstract values flowing to RPC parameters, significantly contributing to the high precision achieved.

5.2.2 Message-Based Communication (MBC). Resolving an MBC entails solving the following tuple: $\langle callsite, messageKey, callee, [\langle payload_i, param_j \rangle, ...] \rangle$, with four aspects outlined below. To resolve MBC, a static analysis first reaches the MBC callsite, then determines the message key used at the callsite, and identifies the message receiver (callee) based on the message key and routing rules. Finally, it constructs the value flow from the payloads at the callsite to the corresponding parameters at the callee. The right section of Table 2 displays MICANS' results in resolving MBC.

Understanding the Results. For soundness, MICANS effectively handles the four aspects of MBC resolution, achieving commendable soundness by successfully resolving 49 out of 55 total MBC callees. It's important to note that even a single MBC callee can be significant for static analysis, as it may uncover numerous true behaviors of the program at runtime. For instance, in the *basemall* benchmark, MICANS successfully resolves 17,997 real call graph edges and identifies 8,487 real methods invoked at runtime from one MBC callee. Similar to its approach with RPC, MICANS meticulously models MBC mechanisms, and addresses underlying features to maximize the coverage of MBC callsites, thus enabling a comprehensive analysis of the MBC mechanism. However, certain specific scenarios remain beyond MICANS' current capabilities. For example, in the *youlai* benchmark, routing rules are dynamically retrieved from the Internet at runtime, making it hard for MICANS to statically collect these routing rules, which impacts the resolution of MBC.

Regarding precision, when operating under the context-insensitive (CI) configuration, the precision of MICANS is significantly compromised. This stems from the merging of message keys and payloads passed to MBC callsites from different methods under CI. For instance, in context c_1, $messageKey_1$ corresponds to $payload_1$, while in context c_2, $messageKey_2$ corresponds to $payload_2$. With a CI approach, however, $messageKey_1$ becomes associated with both $payload_1$ and $payload_2$, resulting in imprecision. Additionally, MBC routing rules are often constructed using a builder pattern, which causes further merging of routing rules under CI. This accounts for the significant number of false MBC callees resolved in the *basemall* benchmark.

However, a key advantage of MICANS is that it is designed and implemented in a context-sensitive (CS) manner, even though its formal rules in Section 4 are presented as context-insensitive for simplicity. This means that MICANS can achieve better precision by simply being run with the CS option, without any other changes. For example, by adopting a selective context-sensitive strategy, such as 4-object sensitivity for application code [Li et al. 2020; Smaragdakis et al. 2014; Tan et al. 2021], MICANS effectively resolves the imprecision issues by distinguishing between distinct messages and routing rules under CS. As a result, this approach leads to an average precision increase to 70.8% (see the rightmost column of Table 2), with only up to a 10% increase in analysis time, adding only dozens of seconds.

5.3 RQ2 — Effectiveness of MICANS in Constructing Call Graphs

As a foundational analysis, we evaluate MICANS' capability in constructing call graphs, a critical piece of information necessary for various inter-procedural static analyses [Samhi et al. 2024; Smaragdakis and Balatsouras 2015]. In our comparison, we include two state-of-the-art pointer analysis tools, JackEE [Antoniadis et al. 2020] and Jasmine [Chen et al. 2022], which are specifically designed to analyze enterprise applications featuring framework elements such as dependency injection, lifecycle, and aspect-oriented programming. In microservice systems, business processes initiate at web endpoints and proceed through method calls within services and cross-service communications. Therefore, we assess the effectiveness of these tools in constructing call graphs from each web endpoint, focusing on two primary metrics: call graph edges and reachable methods, which are essential for evaluating the quality of call graph construction [Li et al. 2018, 2019; Smaragdakis et al. 2011, 2014; Tan et al. 2017].

Table 3 presents our results, focusing solely on the application scope due to space limitations (excluding "Lib/Jdk-Lib/Jdk" entries). "App-App" indicates call graph edges where both the callsite and callee are within app methods. "App-Lib/Jdk" refers to edges where the callsite is in an app method but the callee is in a library or JDK method. "App" denotes reachable methods within the app code, while "Lib/Jdk" corresponds to reachable methods in the library or JDK code. The "Total" column shows the *actual* number of call graph edges or reachable methods collected at runtime. The "Recall" column represents the number of call edges or reachable methods reported by the static

Table 3. Call graph construction results and analysis time. "Recall", "Total" and "R/T" mean the real results resolved by a static analysis tool, the real results collected by running dynamic analysis and the recall rate.

Benchmark	Tool	Call Graph Edges						Reachable Methods						Time (s)
		App-App			App-Lib/Jdk			App			Lib/Jdk			
		Recall	Total	R/T	Recall	Total	R/T	Recall	Total	R/T	Recall	Total	R/T	
netdisk	JackEE	39		14.29%	3		0.99%	50		17.30%	2752		29.87%	242
	Jasmine	39	273	14.29%	3	303	0.99%	50	289	17.30%	2543	9212	27.61%	245
	MICANS	187		68.50%	133		43.89%	184		63.67%	3882		42.14%	117
xmall	JackEE	186		79.49%	98		77.17%	191		83.04%	4169		47.39%	330
	Jasmine	181	234	77.35%	83	127	65.35%	186	230	80.87%	3672	8798	41.74%	346
	MICANS	205		87.61%	99		77.95%	206		89.57%	4269		48.52%	138
onemall	JackEE	139		7.97%	46		8.38%	164		8.87%	7697		9.00%	249
	Jasmine	154	1745	8.83%	56	549	10.20%	176	1848	9.52%	17906	85546	20.93%	529
	MICANS	1565		89.68%	336		61.20%	1565		84.69%	45249		52.89%	317
mogu	JackEE	604		50.04%	1225		59.73%	615		49.04%	65315		42.82%	1020
	Jasmine	627	1207	51.95%	1372	2051	66.89%	693	1254	55.26%	67624	152551	44.33%	1345
	MICANS	1093		90.56%	1698		82.79%	1031		82.22%	76701		50.28%	290
basemall	JackEE	588		40.89%	368		28.86%	581		37.70%	37118		26.80%	814
	Jasmine	924	1438	64.26%	643	1275	50.43%	958	1541	62.17%	70054	138524	50.57%	996
	MICANS	1153		80.18%	878		68.86%	1282		83.19%	72884		52.61%	254
youlai	JackEE	376		70.28%	256		44.37%	419		67.47%	29002		45.95%	1529
	Jasmine	376	535	70.28%	255	577	44.19%	419	621	67.47%	29414	63115	46.60%	1864
	MICANS	486		90.84%	391		67.76%	536		86.31%	34289		54.33%	603
novel	JackEE	106		76.26%	138		41.57%	120		55.81%	23935		45.83%	697
	Jasmine	106	139	76.26%	138	332	41.57%	120	215	55.81%	23866	52224	45.70%	822
	MICANS	117		84.17%	204		61.45%	142		66.05%	30421		58.25%	171
sduoj	JackEE	556		40.03%	385		39.77%	626		41.71%	36325		45.62%	461
	Jasmine	556	1389	40.03%	375	968	38.74%	631	1501	42.04%	35727	79628	44.87%	630
	MICANS	1349		97.12%	899		92.87%	1330		88.61%	46697		58.64%	194
roncoo	JackEE	383		74.37%	227		79.09%	404		68.01%	23426		44.41%	439
	Jasmine	383	515	74.37%	230	287	80.14%	404	594	68.01%	27088	52744	51.36%	701
	MICANS	390		75.73%	264		91.99%	409		68.86%	29800		56.50%	139
mall4cloud	JackEE	327		84.94%	162		55.67%	297		62.13%	22419		41.11%	506
	Jasmine	318	385	82.60%	176	291	60.48%	292	478	61.09%	31159	54535	57.14%	792
	MICANS	350		90.91%	190		65.29%	318		66.53%	33511		61.45%	317
Average recall rate and analysis time (in seconds)														
JackEE		53.85%			43.56%			49.11%			37.88%			629
Jasmine		56.02%			45.90%			51.96%			43.08%			827
MICANS		85.53%			71.41%			77.97%			53.56%			254

analysis tool that are verified as actual, within the scope of those dynamically collected. Please note that we dedicated substantial effort to developing a diverse set of test cases to thoroughly exercise the core business logic dynamically. Additionally, to collect call graph edges and reachable methods for the recall experiment, we implemented a Java Agent to gather runtime information.

Understanding the Results. Unlike the RQ1 experiment, where RPC and MBC callees were manually verified for ground truth, assessing call graph precision is particularly challenging because: (1) a call graph edge or reachable method reported by static analysis cannot be considered false solely because it is absent from the limited runtime data, and (2) the volume of call graph edges and reachable methods is far greater than in RQ1, making manual verification practically unfeasible. As a result, this experiment focuses primarily on evaluating soundness and efficiency.

Table 3 displays the detailed results. Overall, MICANS achieved a significantly higher recall rate (reflecting superior soundness) compared to JackEE and Jasmine, in terms of both call graph edges and reachable methods across all benchmarks. Specifically, in app scope, MICANS achieved recall rates of 85.53% and 77.97%, compared to 53.85% and 49.11% for JackEE, and 56.02% and 51.96% for Jasmine. The improvement in soundness is substantial, indicating that many more actual

Table 4. Taint analysis results of different tools. #VerifiedFlow represents the ground truth taint flows. The numbers following each tool indicate reported taint flows (with true taint flows in parentheses). Jasmine^{D} and Jasmine^{F} refer to Jasmine's Doop and FlowDroid versions, respectively. MICANS and MICANS$_{CS}$ represent MICANS with context-insensitive and context-sensitive strategies, respectively.

Benchmark	#Source	#Sink	#VerifiedFlow	JackEE	Jasmine^{D}	Jasmine^{F}	MICANS	MICANS$_{CS}$
netdisk	163	31	10	4 (4)	2 (2)	0 (0)	23 (10)	19 (10)
xmall	110	20	15	15 (15)	1 (1)	0 (0)	19 (15)	15 (15)
onemall	85	70	58	21 (21)	4 (4)	21 (21)	58 (58)	58 (58)
mogu	240	61	59	56 (56)	9 (9)	3 (3)	609 (59)	60 (59)
basemall	328	62	36	29 (29)	1 (1)	0 (0)	194 (36)	93 (36)
youlai	103	20	17	15 (15)	1 (1)	15 (15)	46 (17)	25 (17)
novel	16	19	13	11 (11)	7 (7)	11 (11)	50 (13)	13 (13)
sduoj	87	23	40	22 (22)	1 (1)	0 (0)	289 (40)	101 (40)
roncoo	499	61	64	64 (64)	0 (0)	0 (0)	244 (64)	117 (64)
mall4cloud	102	41	23	23 (23)	0 (0)	20 (20)	69 (23)	23 (23)
TOTAL	1733	408	**335**	260 (**260**)	26 (26)	70 (**70**)	1601 (335)	524 (**335**)

runtime behaviors are successfully captured and incorporated into the static analysis. For example, in *onemall* benchmark, MICANS resolved an additional 62,357 real call graph edges and 28,732 reachable methods (including the "Lib/JDK-Lib/JDK" cases not shown in the paper, as explained above) that were collected at runtime but missed by the other two state-of-the-art tools.

MICANS outperformed the other two tools in soundness for two key reasons. First, MICANS effectively handles service communication mechanisms (as demonstrated in RQ1), enabling coverage of RPC and MBC callees, which further increases the number of call graph edges and reachable methods. Second, it handles programming paradigms like dependency injection (DI) more comprehensively, leading to greater coverage.

We also identified two main factors contributing to soundness loss in MICANS. First, similar to JackEE and Jasmine, MICANS does not analyze the internal code of certain framework APIs, as their complex implementations make analysis difficult. Instead, it models their side effects for pointer analysis to strike a better trade-off. However, in the recall experiment, some of these APIs may trigger dynamic proxy invocations and related callsites. While the overall side effects are modeled, these dynamically collected callsites remain unfiltered in our recall experiment to reflect real-world scenarios. Second, MICANS does not yet fully handle certain framework features, such as Aspect-Oriented Programming (AOP) and the Spring lifecycle, which also impact its soundness. However, these features are orthogonal, and MICANS can be extended to handle them in the future to further enhance its soundness.

Although efficiency is not the primary focus of MICANS, it runs on average 1.48X faster than JackEE and 2.26X faster than Jasmine, demonstrating good performance. Identifying the exact reasons for improved efficiency in such complex microservice systems among those sophisticated analysis tools is challenging, as many factors can influence the analysis time. However, we think a major contributor to MICANS' efficiency is its foundation on the Tai-e framework, which provides faster analysis speeds in handling language features [Tan and Li 2023]. Additionally, MICANS is designed with a strong focus on balancing soundness and precision, as seen in its RPC resolution approach discussed in RQ1, where it uses code patterns to resolve RPC calls as precisely as possible. This careful trade-off also contributes to its overall efficiency.

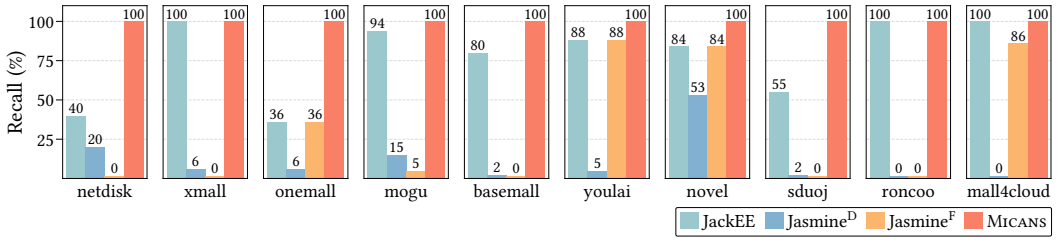

Fig. 8. Recall rates for taint analysis achieved by different tools across 10 real-world benchmarks.

5.4 RQ3 — Support of MICANS for Analysis Client: A Case Study on Taint Analysis

Taint analysis is used to detect security vulnerabilities, such as injection attacks and data leaks, by tracing external input or sensitive data that flows into functions potentially executing or exposing this data. As an analysis client, taint analysis relies heavily on the result of foundational analysis. The more program behaviors (e.g., value flows) the foundational analysis can capture, the better taint analysis can identify vulnerabilities. To assess the effectiveness of taint analysis supported by MICANS, we follow previous work [Alqaradaghi and Kozsik 2024] to conduct a simulated injection experiment. Specifically, we simulated vulnerabilities in real-world benchmarks by tagging string fields of classes likely to receive external input (mainly from web requests) as sources, and injecting sink calls into randomly chosen methods across classes handling application logic and triggered by external requests. These sources and sinks are spread over different services. This setup maximizes potential taint flows for comprehensive coverage.

In our comparison, we also include JackEE [Antoniadis et al. 2020] and Jasmine [Chen et al. 2022], both of which offer advanced taint analysis based on pointer analysis. Other security analysis tools for microservice systems [Wang et al. 2020; Zhong et al. 2023] were not included in this comparison, as they are proprietary internal tools and are not available as open-source.

Understanding the Results. Table 4 presents the taint analysis results for each tool. Considerable effort was invested in manually verifying the true taint flows (shown as #VerifiedFlow in the table) formed between the given sources and sinks. Each tool's column lists the total taint flows detected, with verified true flows in parentheses. MICANS identified all 335 taint flows, outperforming JackEE's 260 and Jasmine's 70, exhibiting significantly higher recall rates, as further displayed in Fig. 8. Notably, although MICANS generated over a thousand taint flows under the context-insensitive approach—making manual verification potentially time-consuming and error-prone—the underlying Tai-e framework [Tan and Li 2023] provides a taint flow graph that visualizes the propagation of taint. This significantly streamlines the verification process, enabling efficient and reliable validation of the flows within a relatively short period.

We observed that the taint flows detected by JackEE and Jasmine tend to be simpler, with call chain lengths no greater than three. In contrast, MICANS effectively detects more complex flows. For instance, in the *sduoj* benchmark, MICANS successfully identified a taint flow with a call chain length of 20 across three distinct services, a level of complexity the other tools could not handle. The primary limitation of JackEE and Jasmine in detecting complex flows stems from their limited capacity to support certain programming paradigms, such as dependency injection (DI), and their inability to handle service communication mechanisms. This constraint hinders their ability to detect cross-service taint flows that MICANS can capture. For example, in the *basemall* benchmark, a critical sink is within the `evaluateOrder` method, which forms a cross-service taint flow only if this method is invoked. However, the object invoking this method is injected into a field in the `MiniOrderController` class via DI using `@Resource` annotation. Due to limited DI resolution capabilities, JackEE fails to resolve this injected object and, as a result, misses this taint flow.

Regarding precision, it is noteworthy that JackEE achieves full precision, outperforming MICANS in this regard. Our investigation revealed two main reasons for this outcome. First, JackEE's underlying Doop framework provides precise modeling for collections, which helps avoid precision loss when analyzing collection-related utilities. By default, MICANS uses a context-insensitive approach, which cannot distinguish value flows that converge at collection-involved method calls. This limitation is evident when MICANS is analyzed with context sensitivity (e.g., 1-object sensitivity), as its precision improves significantly, reducing detected flows from 1601 to 524, as shown in the $\text{MICANS}_{\text{CS}}$ column of the table. Second, as MICANS covers a broader range of complex taint flows than JackEE, it encounters more intricate scenarios that often result in merged value flows within static analysis. These complexities make it challenging for $\text{MICANS}_{\text{CS}}$ to fully reduce false positives, even under context sensitivity. Nonetheless, MICANS demonstrates a good balance between soundness and precision in this security analysis client, detecting all 335 true taint flows (100% recall) with a solid precision (64%, or 335 out of 524).

5.5 Threat to Validity

The primary threats to the validity of our evaluation arise from two aspects: benchmark selection and test case construction.

First, benchmark selection may raise concerns regarding the coverage of RPC and MBC frameworks. While it is impractical to include all categories of RPC and MBC frameworks, we mitigate this threat by incorporating the most widely adopted ones—namely, Apache Dubbo (RPC), Spring Cloud OpenFeign (RPC), RabbitMQ (MBC), and RocketMQ (MBC). We believe this selection sufficiently represents mainstream usage patterns in real-world microservice systems.

Second, for the RQ2 experiment, dynamic call graphs were collected by triggering system behaviors through test cases. Due to the inherent complexity of microservice systems, achieving full behavioral coverage is infeasible. To mitigate this, we supplemented each benchmark's original test suite with dozens of additional test cases aimed at exercising core functionalities through web-based interactions and other common entry points. We believe this strategy ensures that our evaluation remains both representative and reliable.

6 Related Work

Pointer analysis for monolithic enterprise applications. JackEE [Antoniadis et al. 2020] and Jasmine [Chen et al. 2022] utilize pointer analysis to analyze monolithic enterprise applications, computing the objects that each variable in the program may point to, enabling the derivation of information such as value flows and call graphs that are essential for various client analyses. JackEE addresses the unique requirements of enterprise applications by introducing methods to identify and model application entry points, handle programming paradigms such as Dependency Injection, and deliver a sound-modulo-analysis framework for Java data structures, including `Map`. These techniques collectively enhance the soundness, precision, and scalability of static analysis within monolithic applications. Jasmine, on the other hand, specifically targets the challenges posed by Dependency Injection and Aspect-Oriented Programming within Spring applications. By modifying the intermediate representation code, Jasmine improves call graph completeness. However, both tools lack support for service communications in microservices and struggle with intricate value flows arising from complex usage scenarios involving Dependency Injection and Web Endpoint Configuration.

Static analysis clients for microservice systems and monolithic enterprise applications. CFTaint [Zhong et al. 2023] performs compositional field-based taint analysis tailored for microservices, enhancing scalability and precision through the use of function summaries. MSANose [Walker et al. 2020] detects eleven microservices-specific code smells in microservice systems. Liu et al. [Liu et al.

2022] proposed an approach for recording and replaying microservices traffic. They utilized static analysis to identify RPC callsites, which aids dynamic analysis in capturing RPC return values. ANTaint [Wang et al. 2020] tackles challenges from extensive libraries, native methods, and enterprise frameworks in FlowDroid, improving the soundness and scalability of taint analysis by expanding the call graph and applying on-demand taint propagation for libraries. TAJ [Tripp et al. 2009] extends taint analysis to web applications by modeling core JavaEE concepts and frameworks, and leveraging sound-modulo-analysis simplifications of data structures. IBM's F4F [Sridharan et al. 2011] system enhances taint analysis for framework-based web applications by supporting the WAFL language, enabling accurate JavaEE configuration modeling. However, all these static analysis clients focus solely on specific tasks, lacking the capability to compute comprehensive foundational information and offering limited support for essential microservice features, such as RPC, MBC, and Dependency Injection.

Dynamic analysis for microservice systems. Several tools use dynamic approaches to capture runtime behaviors in microservices. SkyWalking [Wu et al. 2023] is an open-source tool for monitoring distributed systems, dynamically tracking method calls across microservices via runtime instrumentation and distributed tracing. MirrorTaint [Ouyang et al. 2023], a non-intrusive dynamic taint analysis tool for JVM-based microservices, tracks the propagation of taints in microservice systems using a mirrored JVM space. Saioc et al. [Saioc et al. 2024] utilize dynamic analysis to detect deadlocks during unit testing and in production for Go-based microservices. Peng et al. [Peng et al. 2022] introduce a trace-based method for measuring microservice systems. Despite their utility, these dynamic techniques introduce additional runtime overhead and often struggle to provide the comprehensive coverage that static analysis can achieve.

7 Conclusion

We introduced MICANS, the first pointer analysis specifically designed for microservice systems. By effectively resolving service communication mechanisms like RPC and MBC and supporting essential paradigms such as DI, MICANS addresses distinct challenges in microservice-based architectures. Our evaluation on real-world benchmarks demonstrates that MICANS achieves remarkable gains in soundness and precision trade-off over existing tools, with significantly enhanced resolution of comprehensive call graphs and complex taint flows. These advancements underscore MICANS' potential as a foundational analysis for microservices. Future work can extend MICANS to broaden its applicability to more intricate microservice systems and a wider range of analysis clients.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. This work is supported in part by National Key R&D Program of China under Grant No. 2023YFB4503804, National Natural Science Foundation of China under Grant No. 62402210, the Leading-edge Technology Program of Jiangsu Natural Science Foundation under Grant No. BK20202001, and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Tian Tan, the co-corresponding author, is also supported by Xiaomi Foundation.

Data-Availability Statement

We have prepared an artifact [Zhang et al. 2025] to reproduce all experimental data presented in Section 5. This artifact includes detailed documentation, as well as the source code, deployment scripts, and test cases for the real-world Microservice systems used in our experiments, which we believe will be useful for future researchers. The artifact can be downloaded from the following URL: https://doi.org/10.5281/zenodo.14043593.

References

Carlos M. Aderaldo, Nabor C. Mendonça, Claus Pahl, and Pooyan Jamshidi. 2017. Benchmark Requirements for Microservices Architecture Research. In *1st IEEE/ACM International Workshop on Establishing the Community-Wide Infrecaseructure for Architecture-Based Software Engineering, ECASE@ICSE 2017, Buenos Aires, Argentina, May 22, 2017*. IEEE, 8–13. doi:10.1109/ECASE.2017.4

Midya Alqaradaghi and Tamás Kozsik. 2024. Comprehensive Evaluation of Static Analysis Tools for Their Performance in Finding Vulnerabilities in Java Code. *IEEE Access* 12 (2024), 55824–55842. doi:10.1109/ACCESS.2024.3389955

Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 794–807. doi:10.1145/3385412.3386026

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. doi:10.1145/2594291.2594299

Miao Chen, Tengfei Tu, Hua Zhang, Qiaoyan Wen, and Weihang Wang. 2022. Jasmine: A Static Analysis Framework for Spring Core Technologies. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 60:1–60:13. doi:10.1145/3551349.3556910

Martin Fowler and James Lewis. 2014. Microservices. Retrieved November 1, 2021 from https://martinfowler.com/articles/microservices.html

Marvin Froeder et al. 2021. OpenFeign. Retrieved November 1, 2021 from https://github.com/OpenFeign/feign

Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 3–18. doi:10.1145/3297858.3304013

Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 102:1–102:28. doi:10.1145/3133926

Joab Jackson. 2022. How airbnb and twitter cut back on microservice complexities. Retrieved November 1, 2022 from https://thenewstack.io/how-airbnb-and-twitter-cut-back-on-microservice-complexities/

JetBrains. 2023. The State of Developer Ecosystem 2023 by JetBrains. Retrieved December 1, 2023 from https://www.jetbrains.com/lp/devecosystem-2023/

Rod Johnson et al. 2023a. Spring Framework Documentation. Retrieved November 1, 2023 from https://docs.spring.io/spring-framework/reference/index.html

Rod Johnson et al. 2023b. Spring Framework Documentation - The IoC Container. Retrieved November 1, 2023 from https://docs.spring.io/spring-framework/reference/core/beans.html

Rod Johnson et al. 2023c. Spring Framework Documentation - Web on Servlet Stack. Retrieved November 1, 2023 from https://docs.spring.io/spring-framework/reference/web.html

Michael Klishin et al. 2021. RabbitMQ. Retrieved November 1, 2021 from https://github.com/rabbitmq/rabbitmq-server

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 129–140. doi:10.1145/3236024.3236041

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 10:1–10:40. doi:10.1145/3381915

Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and analyzing java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 2 (2019), 1–50.

Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program Tailoring: Slicing by Sequential Criteria. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:27. doi:10.4230/LIPICS.ECOOP.2016.15

Jiangchao Liu, Jierui Liu, Peng Di, Alex X. Liu, and Zexin Zhong. 2022. Record and replay of online traffic for microservices with automatic mocking point identification. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New

York, NY, USA, 221–230. doi:10.1145/3510457.3513029

Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, Carlo Curino, Georgia Koutrika, and Ravi Netravali (Eds.). ACM, 412–426. doi:10.1145/3472883.3487003

Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. 2023. Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis. *Proc. ACM Program. Lang.* 7, PLDI (2023), 539–564. doi:10.1145/3591242

Oracle. 2021. Overview of Enterprise Applications. Retrieved November 1, 2021 from https://docs.oracle.com/javaee/7/firstcup/java-ee001.htm

Yicheng Ouyang, Kailai Shao, Kunqiu Chen, Ruobing Shen, Chao Chen, Mingze Xu, Yuqun Zhang, and Lingming Zhang. 2023. MirrorTaint: Practical Non-intrusive Dynamic Taint Tracking for JVM-based Microservice Systems. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2514–2526. doi:10.1109/ICSE48619.2023.00210

Xin Peng, Chenxi Zhang, Zhongyuan Zhao, Akasaka Isami, Xiaofeng Guo, and Yunna Cui. 2022. Trace analysis based microservice architecture measurement. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 1589–1599. doi:10.1145/3540250.3558951

Georgian-Vlad Saioc, Dmitriy Shirchenko, and Milind Chabbi. 2024. Unveiling and Vanquishing Goroutine Leaks in Enterprise Microservices: A Dynamic Analysis Approach. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 411–422.

Jordan Samhi, René Just, Tegawendé F. Bissyandé, Michael D. Ernst, and Jacques Klein. 2024. Call Graph Soundness in Android Static Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 945–957. doi:10.1145/3650212.3680333

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (2015), 1–69. doi:10.1561/2500000014

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 17–30. doi:10.1145/1926385.1926390

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 485–495. doi:10.1145/2594291.2594320

Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. 2011. F4F: taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 1053–1068.

Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1093–1105. doi:10.1145/3597926.3598120

Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. doi:10.1145/3485524

Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 278–291. doi:10.1145/3062341.3062360

John Toman and Dan Grossman. 2019. Concerto: a framework for combined concrete and abstract interpretation. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. *ACM Sigplan Notices* 44, 6 (2009), 87–97.

Uber. 2022. The uber engineering tech stack, part i. Retrieved November 1, 2022 from https://eng.uber.com/tech-stack-part-one-foundation/

Andrew Walker, Dipta Das, and Tomas Cerny. 2020. Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study. *Applied Sciences* 10, 21 (2020). doi:10.3390/app10217800

Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. 2020. Scaling static taint analysis to industrial SOA applications: a case study at Alibaba. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1477–1486. doi:10.1145/3368089.3417059

Sheng Wu et al. 2023. SkyWalking: an APM (Application Performance Monitoring) system, especially designed for microservices, cloud native and container-based architectures. Retrieved November 1, 2023 from https://github.com/apache/skywalking

Teng Zhang, Yufei Liang, Ganlin Li, Tian Tan, Chang Xu, and Yue Li. 2025. *Bridge the Islands: Pointer Analysis for Microservice Systems (Artifact).* doi:10.5281/zenodo.15050505

Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. Hybrid top-down and bottom-up interprocedural analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 249–258. doi:10.1145/2594291.2594328

Zexin Zhong, Jiangchao Liu, Diyu Wu, Peng Di, Yulei Sui, Alex X. Liu, and John C. S. Lui. 2023. Scalable Compositional Static Taint Analysis for Sensitive Data Tracing on Industrial Micro-Services. In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 110–121. doi:10.1109/ICSE-SEIP58684.2023.00015

Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking microservice systems for software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 323–324. doi:10.1145/3183440.3194991

Received 2024-10-30; accepted 2025-03-31