Towards Security-Aware Microservices: On Extracting Endpoint Data Access Operations to Determine Access Rights

Amr S. Abdelfattah[a], Micah Schiewe[b], Jacob Curtis[c], Tomas Cerny[d] and Eunjee Song[e]
Computer Science, ECS, Baylor University, One Bear Place #97141, Waco, TX 76798-7356, U.S.A.

Keywords: Static Analysis, Microservices, Access Rights.

Abstract: Security policies are typically defined centrally for a particular system. However, the current mainstream architecture - microservices - introduces decentralization with self-contained interacting parts. This brings better evolution autonomy to individual microservices but introduces new challenges with consistency. The most basic security perspective is the setting of access rights; we typically enforce access rights at system endpoints. Given the self-contained and decentralized microservice nature, each microservice has to implement these policies individually. Considering that different development teams are involved in microservice development, likely the access rights are not consistently implemented across the system. Moreover, as the system evolves, it can quickly become cumbersome to identify a holistic view of the full set of access rights applied in the system. Various issues can emerge from inconsistent settings and potentially lead to security vulnerabilities and unintended bugs, such as incorrectly granting write or read access to system data. This paper presents an approach aiding a human-centered access right analysis of system endpoints in microservices. It identifies the system data that a particular endpoint accesses throughout its call paths and determines which operations are performed on these data across the call paths. In addition, it takes into account inter-service communication across microservices, which brings a great and novel instrument to practitioners who would otherwise need to perform a thorough code review of self-contained codebases to extract such information from the system. The presented approach has broad potential related to security analysis, further detailed in the paper.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Proper access control enforcement is difficult for decentralized computing. One major challenge in microservice systems is the lacking of a centralized view of the system. This leads to inconsistent data access, violation of privacy, and disruption of data integrity, etc. While documentation and security policies indicate the intended access rights for a system, documentation can quickly fall out of date, and policies can be incorrectly implemented in the ever-changing decentralized environment, leaving the source code of the system the only reliable source to determine access rights. This type of analysis is very important since separate teams typically implement microservices that interact with data that need to consistently enforce Create, Read, Update, and Delete (CRUD) authorization security policies in the system.

We recognize Role-Based Access Control (RBAC) involvement, but still, to determine which roles to apply on which system endpoint, a lower-level granularity must be considered broadly driven by the underlying system data. Most role-based access control enforcements are specifically in place to prevent illegal access to data, represented as data entities in databases. Thus, in more detail, by identifying CRUD operations performed on the various data entities, one can get a robust idea of the permissions allotted to specific control flows.

This paper proposes a methodology to analyze microservice systems' REST endpoints with respect to the CRUD operations performed on data entities associated with these endpoints. It is set to account for entities across inter-service calls to reflect the decentralized system nature. Having such information has broad application potential, such as being able to determine differences between the expected set of re-

[a] https://orcid.org/0000-0001-7702-0059
[b] https://orcid.org/0000-0001-6173-7397
[c] https://orcid.org/0000-0002-0295-564X
[d] https://orcid.org/0000-0002-5882-5502
[e] https://orcid.org/0000-0002-8680-9411

Abdelfattah, A., Schiewe, M., Curtis, J., Cerny, T. and Song, E.
Towards Security-Aware Microservices: On Extracting Endpoint Data Access Operations to Determine Access Rights.
DOI: 10.5220/0011707500003488
In *Proceedings of the 13th International Conference on Cloud Computing and Services Science (CLOSER 2023)*, pages 15-23
ISBN: 978-989-758-650-7; ISSN: 2184-5042
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

quired permissions as defined in the code on a given endpoint and the inferred set of CRUD rules given as output by our tool.

The approach analyzes the system code to (1) gather information about inter-service calls and to (2) infer what CRUD operations are performed based on specified data-source calls within each endpoint's call paths. Next, (3) we integrate these results and compute the full set of CRUD accesses performed by each endpoint by analyzing the inter-service call flow to determine the extra CRUD operations each endpoint runs indirectly via called services.

To assess our approach, we implemented a prototype tool and performed a case study on an established microservice benchmark with 41 microservices. We compared our results to the ground truth to assess our results. Our results are promising for being able to successfully accomplish this in a large microservice codebase and indicate high accuracy for the identification of the full set of per-entity CRUD operations in the microservices-based system.

In summary, the approach provides analysts with a centralized view of endpoints and their data access operations to determine proper access rights in microservice systems. Therefore, our approach aims to support a human-centered security analysis by providing the following benefits: (1) help security analysts to determine access roles to be applied on endpoints of new microservices; (2) help to analyze inconsistent access rights across already specified endpoint access roles by assessing the intra-service and inter-service inconsistencies across microservices; (3) help to promote security-aware system evolution by detecting whether performed code changes promoted changes in data access operations and if they should lead to endpoint access rights updates.

This paper is organized as follows: Section 2 introduces related work, while Section 3 introduces the proposed approach analysis, design and implementation details. Section 4 describes performed case study followed by discussion on limitations and threats to validity in Section 5. Conclusions are given in Section 6.

2 RELATED WORK

Modern software has seen a shift to highlight the importance of maintaining robust authorization security in its applications. One example of this is commonly seen in microservices with RBAC corresponding to CRUD permissions which are used for controlling access to resources such as database tables or entities. As a result of this, we have seen increased importance in uncovering bugs related to incorrect permissions being applied in code bases or locating permission errors by omissions within the source code. Another important topic is inferring access permissions in software systems, which we will also outline later.

Modern microservice systems face many challenges (Bogner et al., 2021). These relevant to our work include: lack of a centralized system view, technological heterogeneity, insufficient documentation, and decentralized team coordination. We aim to alleviate some of these challenges by providing a centralized view of all CRUD operations executed by endpoints within a microservice system in a language-agnostic fashion; this will yield better automated documentation and team coordination.

A method to identify conflicting applications of RBAC at the endpoint level in a microservices (Das et al., 2021) utilizes the roles defined on the JSR-375 endpoints and, similar to our approach, utilizes inter-service calls to detect inconsistencies. In contrast, this paper focuses on the CRUD accesses each endpoint makes as opposed to the explicit roles defined on the endpoints. In particular, we aim to infer the complete set of CRUD accesses made on various entities to determine both the security policies and the roles explicitly defined on endpoints.

Solutions using machine learning were proposed in (Le et al., 2015) to infer access roles for web apps. However, they determine access permissions for users based on dynamic analysis using existing user roles to crawl the webpages. The approach deals with inconsistent access rules by human intervention, which is another downside of this approach. Our work automates the inference of CRUD access roles only by analyzing the source code of the systems involved.

An access log mining solution (Xiang et al., 2019) tracked changing access permissions in systems and determined access control policy changes. However, the system must have been deployed, and thus potentially vulnerable to malicious attacks if there was an unintended access control bug. By shifting the analysis from run-time to only source code, we can avoid these sorts of vulnerabilities up front.

A static analysis tool named FixMeUp finds and repairs access control bugs in PHP applications (Son et al., 2013) since it is not uncommon for developers to accidentally omit these when writing applications. High-level specifications for identifying access control statements are provided as input to the system. However, their approach does not guarantee the resulting program is semantically correct and recommends reviewing the changes made by the tool. While their approach also seeks to infer the access permissions on endpoints as we do, here the goals and as-

sumptions are radically different. FixMeUp seeks to create a single solution for PHP applications, assuming access permissions are incorrect and attempting a repair. Our approach creates a generic solution adaptable to different languages and frameworks, assuming the software's functionality is correct and seeking to determine the access permissions that exist for consumption by another application.

Detection of potential security holes ("security policy differencing") (Srivastava et al., 2011) was addressed by comparing multiple implementations of the same API. The approach uses control flow in order to determine the conditions that must be met when sensitive statements are executed in an API, which are predefined. Inconsistent access conditions result in identifying a potential bug in one of the API implementations. Our work does not identify differences between implementations, but infers permissions that can be a result of some deeply nested code or external service call that has additional required permissions.

Finally, ROLECAST statically analyzes code for "missing" security checks during sensitive operations in web apps, such as database writes (Son et al., 2011). Their work is relevant because it is similar in that we are identifying implied or possibly "missing" security roles on endpoints. However, we also uncover "inconsistencies" by detecting differences between the set of permissions required to perform an action (an endpoint call) and the inferred set of CRUD permissions from our tool.

3 APPROACH ANALYSIS AND DESIGN

Microservice systems consist of multiple codebases, such as *MS-1, MS-2, etc.*, and each microservice typically contains multiple endpoints. Identifying CRUD accesses and allotting them to endpoints requires multiple steps. These steps are abstracted in Fig. 1.

The process starts with extracting the endpoints from the whole system, such as *EP-1, EP-M, etc*. After that, it identifies the CRUD operations used per each endpoint; it analyzes the endpoint source code, converts it to a call-graph, and explores the different accesses to data entities. The third step considers inter-service calls. It aggregates the holistic endpoints dependency graph — generated in a separate process— with the identified CRUD operations per microservice. It accumulates the individual CRUD accesses from an endpoint with the CRUD operations used in its dependencies endpoints (other microservices).

Highlighting the usage of each step is shown in Fig. 2. It illustrates a simplified use-case for reservations in a hotel, such that the *RentalService* has an endpoint besides an UPDATE operation used to start a room rent process; this endpoint calls two other endpoints on *RoomService* and *ClientService*, each has different CRUD operations as well.

Applying the process starts by extracting these endpoints from the following three microservices: `POST /rent/room`, `POST /rooms/reserve`, and `GET /client/blacklist`. Then, it identifies entities CRUD operations run by individual endpoints as shown as `UPDATE ROOM`, `CREATE reservation`, and `READ client`. Afterward, it utilizes the service dependency graph to analyze which system endpoints interact with each other and aggregates the CRUD operations accordingly. Finally, Fig. 3 concludes the expected outcome that the request access right to *RentalService* should satisfy the access of the three CRUD operations to function properly, however, both *RoomService* and *ClientService* do not require additional CRUD operations access to proceed.

3.1 Implementation Details

Analyzing microservices-based systems presents a steep challenge since these contain many distributed parts that evolve independently. Also, the microservices architecture encourages the heterogeneity between the included services (Cerny et al., 2020).

The proposed approach is implemented based on ReSSA (Relative Static Structure Analyzer) methodology (Schiewe et al., 2022). ReSSA extracts microservice high-level conceptual components using static code analysis; it achieves this by operating on a language-agnostic version of an abstract syntax tree dubbed a Language-Agnostic Abstract Syntax Tree (LAAST). This enables the analysis of heterogeneous microservices-based systems, such that it keeps extracting the matched components from the analyzed source code. ReSSA operates using a list of hierarchical parsers that run against all nodes in LAAST. Upon a parser matching the information stored within the tree, it runs all sub-parsers against the descendent nodes of the node that was matched.

Our implementation targets the Spring-Boot framework-based systems to identify CRUD accesses through the endpoint calls. However, ReSSA is generalizable to other platforms which utilized component-based development, as demonstrated for `C++` microservice benchmark in (Schiewe et al., 2022). Furthermore, most libraries offer consistent API for accessing and interacting with data entities; this emphasizes the potential of ReSSA to be utilized in our approach; it could identify these CRUD accesses hetero-

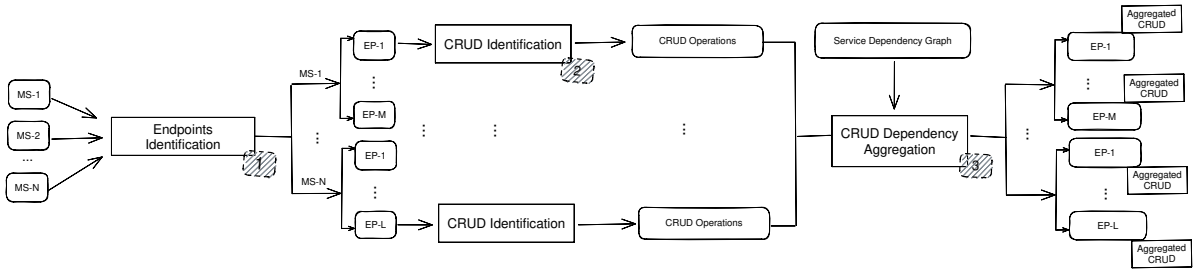

Figure 1: Microservices CRUD accesses detection process.

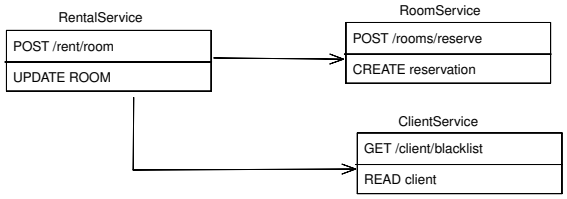

Figure 2: Microservices use-case example.

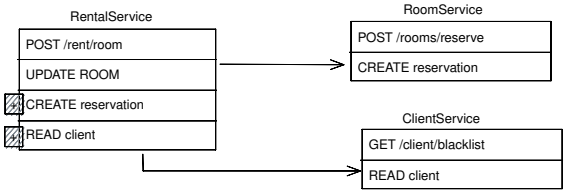

Figure 3: An example after Processing.

geneously.

The following subsections walk through the steps mentioned above in Fig. 1 to highlight the implementation perspective of our approach[1].

3.1.1 Endpoint Identification

The proposed approach utilizes the capabilities of ReSSA to construct a service dependency graph. It produces a format interface with our approach as shown in Listing 1. It is structured per each microservice to list service endpoints and user flows. It shows the endpoints and the calls performed from each endpoint to others.

ReSSA uses a pattern-matching technique to detect the endpoints in the source code. As shown in Table 1, the pattern used in Spring-Boot framework that identifies endpoints as any method annotated `@.*Mapping` (covering both `@PostMapping` and `@RequestMapping`–Spring-Boot's endpoint-defining annotations) within a class annotated `@RestController`.

Listing 1: Selected Schema for Determining Endpoint Calls.

```
{
  "services": [
    {
      "calls": [
        {
          "endpoint": "/called/endpoint/slug",
          "from": ["VERB /calling/endpoint/slug"],
                // All endpoints perform this call.
          "method": "VERB", // i.e. GET, POST, PUT
                , etc.
          "service": "called-service-name"
        },
        ...
      ],
      "endpoints": [
        {
          "code_mapping": "...", // Identifier for
                in-code endpoint definition of the
                endpoint.
          "method": "VERB",
          "name": "/slug"
        },
        ...
      ],
      "name": "service-name"
    }
  ]
}
```

3.1.2 CRUD Identification

This phase includes three parts of information to link together as follows: the performed CRUD operations, the target entities of these operations, and their called endpoints.

ReSSA (Schiewe et al., 2022) has demonstrated that it is adept at creating a design that can be easily expanded to accommodate more frameworks, we extended its components to add CRUD operations to match against as shown in Listing 2. However, every framework has a unique API for accessing database entities, so extracting these individual CRUD operations entails some form of analysis of the API the underlying library uses. The patterns used for Spring Data framework are summarized in Table 1. It identifies four patterns of CRUD methods as follows: `find.*` that matches against many variations

[1]Implementation available at http://github.com/cloudhubs/authz-flow-analysis, accessed on 2/13/2023.

of `find`, where all of which logically group together for our purposes, `remove`, `delete`, and `save`. In addition, these operations are required to be called through an `@Autowired .*Repository`. Mapping these APIs with the CRUD operations, such that `find` corresponds to the `READ` operation, while `remove` and `delete` correspond to the `DELETE` operation. `save` is tricky; it corresponds to both `CREATE` and `UPDATE`, depending on whether the entity it saves already exists, as illustrated in Listing 3.

Identifying the target entities of these CRUD operations is the second information to extract. Inferring the entity's name is based on the returned type from methods, such as find, get, etc. For example, `Foo f = repository.get(id)` implies the entity name as `Foo`. In the case of the absence of the return type, the `*.Repository` naming convention-based heuristic is employed, for example, `FooRepository` implies that `Foo` is an entity name.

Listing 2: Selected Schema for Entity CRUD Operations.

```
{
  "endpoint_method": {
    "entity": {
      "CREATE": boolean,
      "READ": boolean,
      "UPDATE": boolean,
      "DELETE": boolean
    },    ...
  },  ...
}
```

Listing 3: Common difference between CREATE and UPDATE usage in TrainTicket Benchmark (from contacts.service.ContactsServiceImpl).

```
// Method performing a CREATE
@Override
public Response createContacts(Contacts contacts,
    HttpHeaders headers) {
    Contacts contactsTmp =contactsRepository.
        findById(contacts.getId());
    if (contactsTmp != null) {
        // ...
        return new Response<>(0, "Already Exists",
            contactsTmp);
    } else {
        contactsRepository.save(contacts);
        return new Response<>(1, "Create Success",
            null);
    }
}

// Method performing an UPDATE
@Override
public Response modify(Contacts contacts,
    HttpHeaders headers) {
    Response oldContactResponse = findContactsById
        (/* */);
    // ...
    Contacts oldContacts = (Contacts)
        oldContactResponse.getData();
    if (oldContacts == null) {
        // ...
        return new Response<>(0, "Contacts not
            found", null);
    } else {
        oldContacts.setName(contacts.getName());
        oldContacts.setDocumentType(contacts.
            getDocumentType());
        oldContacts.setDocumentNumber(contacts.
            getDocumentNumber());
        oldContacts.setPhoneNumber(contacts.
            getPhoneNumber());
        contactsRepository.save(oldContacts);
        // ...
        return new Response<>(1, "Modify success",
            oldContacts);
    }
}
```

The third part of the information is achieved by linking the extracted CRUD operations with their corresponding endpoint. The approach matching between the `code_mapping` field from Listing 1 with the `endpoint_method` field from Listing 2 to indicate the [contains] relationship between them. We generate a simple call-graph based on method names, it identifies what interface methods call what helper methods to run business logic. This allows us to trace any possible attempted CRUD operation back to its originating endpoint, generating the desired output mapping. A summary of the general structure that we are matching can be found in fig. 4, it describes the flow starting from the `@RestController` that contains the endpoints' implementations, then matched the `@Service` related call to its implementation, then identifies the `localCall` that performs the CRUD operations calls.

3.1.3 CRUD Dependency Aggregation

Once we have the output for both the endpoint calls and the CRUD operations per endpoint, we can begin the final step to infer CRUD operations for endpoints in the whole system. It analyzes the flow of calls through the system and their entity CRUD operations. The approach introduces an algorithm as shown in Listing 4. It iterates as follows for every endpoint in every microservice controller: visit all of the recorded calls to other endpoints that occur within the endpoint. Merging the operations between the callee is simply adding the operations from the called endpoint to the callee endpoint, and never removing entity CRUD operations. We continue this iterative process over each endpoint until a full pass over all of the endpoints results in no endpoint's entity CRUD operation changes, or until it has been repeated over

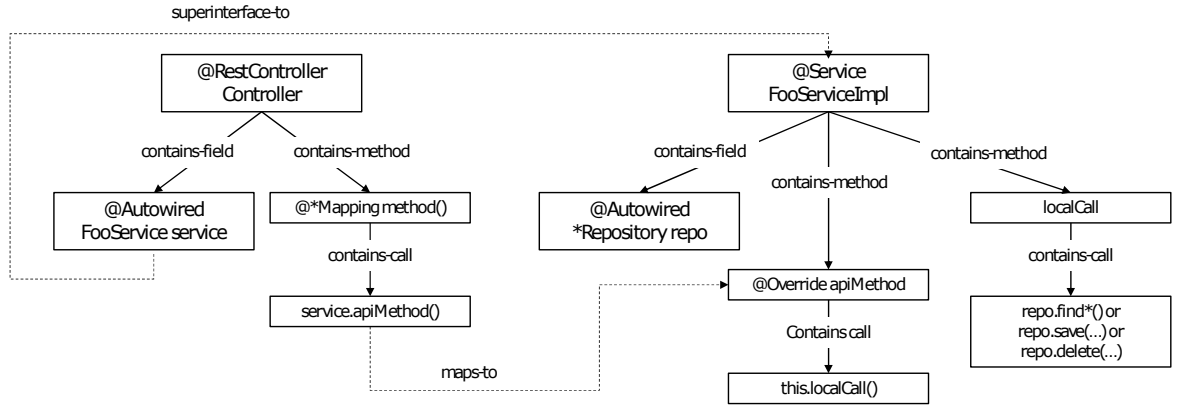

Figure 4: Architecture of in-code structure containing CRUD accesses.

all the endpoints in the graph. This allows us to continue adding CRUD operations as changes are being rippled throughout the endpoints in the services.

Listing 4: Pseudocode: identify the set of CRUD operations using inter-service call flow.

```
procedure merge_crud_access(e1, e2):
  if !e1.crud_access:
    e1.crud_access = e2.crud_access
    return

  // Add each CRUD operation for every entity
  for each entity in e2.crud_access:
    for each crud_op in entity.crud_operations:
      e1.crud_access[entity] += crud_op

// Initialize the service endpoint's base CRUD
     access
// operations
for each service endpoint e:
  e.crud_access = endpoint_crud_accesses[e.
       code_mapping]

let crud_access_changed = false
for |endpoints|:
  for each service endpoint e:
    // Add all CRUD access rules from every called
    // service
    for each service call c made by e:
      let old_crud_access = e.crud_access
      merge_crud_access(e, c.endpoint)
      crud_access_changed =
        (e.crud_access != old_crud_access)
        || crud_access_changed

  if !crud_access_changed: break

return endpoints with merged CRUD operations
```

4 CASE STUDY

To test our proposed approach, we implemented a proof-of-concept for our three-step process. We then tested this proof-of-concept on the TrainTicket testbed (Zhou et al., 2018)[2]. This testbed has a total of 41 microservices that were implemented using the Java Spring-Boot framework.

We ran the case study against TrainTicket's codebase and validated this against a ground truth that we determined. This ground truth was partially reconstructed by manually identifying where CRUD accesses were and what services called each other. A simple script was created to tie user flows together and copy/paste CRUD operations as needed. It produces a total of 1364 CRUD operations identified. The proposed technique identified 1322 of them automatically, yielding 96.9% accuracy overall.

4.1 Discussion and Limitations

Let us discuss more those 42 CRUD operations that were missing in our identification result. Four CRUD operations were misinferred because they mismatched the heuristic naming convention technique that we used to identify the entity names. Once inheritance hierarchy information is matchable within ReSSA, this error will be resolved because it will be a more reliable way to identify the targeted entity name.

The remaining 38 misinferred CRUD operations are resulted from the classification of `CREATE` and `UPDATE` operations. In our initial implementation, we

[2]TrainTicket testbed: https://github.com/FudanSELab/train-ticket, used commit hash a4ed2433b0b6ab6e0d60115fc19efecb2548c6cd, accessed on 2/13/2023.

Table 1: The Matching Patterns for Spring-Boot Framework.

Identified Components	Matching Patterns

CRUD Operations	`find.*`
	`remove`
	`delete`
	`save`

Repository Instance	`@Autowired .*Repository`
Endpoint Methods	Class: `@RestController` Method: `@.*Mapping`

attempted to delineate them based on a common pattern of usage: that saving an object retrieved from a database is always an `UPDATE`, and that saving an object not retrieved from a database is always a `CREATE`, as illustrated before in Listing 3. Comparing the total set of `CREATE` and `UPDATE` operations that are inferred using this heuristic yields 74% accuracy against the ground truth. This is not a particularly compelling result. However, there are a few more cases that are related to this `CREATE` and `UPDATE` heuristic.

One case is shown in Listing 5. This call to `save` is both a `CREATE` and an `UPDATE`. The heuristic fails to recognize it since it could depend on control flow (whether `repository.findByIndex(0)` is null or not). This could be solved using a similar method handling multiple endpoint targets for inter-service calls in the service call-graph generating ReSSA. On the other hand, recording it could be proposed instead of attempting to classify it in a binary fashion. However, this would not solve all misidentified entity CRUD operations.

Listing 5: TrainTicket `UPDATE` not matching our heuristic (from consignprice.service.ConsignPriceServiceImpl).

```
@Override
public Response createAndModifyPrice(ConsignPrice
    config,
        HttpHeaders headers) {
    // ...omitted for space...
    //update price
    ConsignPrice originalConfig;
    if (repository.findByIndex(0) != null) {
        originalConfig = repository.findByIndex(0)
            ;
    } else {
        originalConfig = new ConsignPrice();
    }
    originalConfig.setId(config.getId());
    // ...assign over attributes from parameter
        config...
    repository.save(originalConfig);
    return new Response<>(1, success,
        originalConfig);
}
```

Another case the heuristic did not account for is shown in Listing 6. The update is performed by validating that the database contains the desired information, followed by creating a new data object containing the validated information and saving this new object to perform the update. While it is an odd way of performing an `UPDATE` (and arguably risky, given the chances of accidentally overwriting attribute values with `null` if the data schema changed and you didn't start setting the new attributes as well), it is still a valid way of performing an `UPDATE` that we must consider in the future.

Listing 6: TrainTicket `UPDATE` not matching our heuristic (from config.service.ConfigServiceImpl).

```
@Override
public Response update(Config info, HttpHeaders
    headers) {
    if (repository.findByName(info.getName()) ==
        null) {
        String result = config0 + info.getName() +
            " doesn't exist.";
        return new Response<>(0, result, null);
    } else {
        Config config = new Config(info.getName(),
            info.getValue(), info.getDescription()
                );
        repository.save(config);
        return new Response<>(1, "Update success",
            config);
    }
}
```

It is clear that the heuristic used to differentiate between the use of Spring's `save` as a `CREATE` or `UPDATE` is flawed based on the noted cases and others. If this heuristic were refined, our approach would capitalize on it immediately to produce stronger results. Moreover, we consider merging `CREATE` and `UPDATE` operations in the same group of access, thus the algorithm does not differentiate between them. This leaves all the other CRUD operations inferred property, except those four that are related to the misidentified entities.

In conclusion, our case study shows very promising results for identifying CRUD entity access, while raising some framework-specific challenges with our current toolset. It is also only limited to accessing database entities and does not support other technologies like Redis or message queues such as Kafka yet.

5 THREATS TO VALIDITY

Using Wohlin's taxonomy (Wohlin et al., 2000), we examined four types of validity threats: construct validity, internal validity, external validity, and conclusion validity. We also elaborate on our custom thresholds used in the proposed method.

5.1 Construct Validity

Construct validity refers to the accuracy of measures in reflecting real-world constructs. To assess this, we used a third-party enterprise system benchmark based on microservices and a well-established development framework that adheres to up-to-date development standards and components. This benchmark is widely used by the microservice community to demonstrate research outcomes, thus reflecting realistic conditions for our approach. However, our results should be considered in the context of other frameworks that may yield alternative results. Moreover, the selected benchmark may introduce a biased representation of coding practice; we countered this by performing manual analysis.

Our method evaluates components such as CRUD operations, repositories, and endpoint methods that are widely recognized across platforms. We also considered REST calls to these endpoints, although we omitted messaging systems from our assessment. Therefore, our method is not suitable for general programs that do not consider those components (i.e., Java Standard Edition).

5.2 Internal Validity

Internal validity challenges the study methods and data analysis. Our approach demonstrates a high level of reliability in inferring entity CRUD operations by following endpoint calls. However, it raises questions that require further research, such as whether it is always possible to link CRUD accesses to the endpoint that triggered them.

Another threat to validity is the lack of support for tracing the control flow and inheritance structures within a program and the simplifying assumptions we had to make to solve them. Consider, for example, the way TrainTicket links its controllers to its business logic using an `@Autowired` service implementation. This auto-wired field is the generic interface for the service, not the concrete service class itself. It is a simple logical leap for a human to skip from the interface to the implementation; however, at this time, ReSSA does not support tracing and matching over such control flow structures. While one can perform simple matching over class and method names in a system, this is snarled up by duplicated class names or method overloads. This makes it difficult to trace the execution from the controller to business logic or among helper methods. Although adding support for such a connection in ReSSA would be valuable, it was not that critical in our approach to identifying CRUD rights.

Our approach utilizes manual analysis as the ground truth for evaluating the performance of the proposed method. Specifically, two authors conducted the manual analysis, and the results were later validated by a third author to reduce any potential bias in the outcomes.

5.3 External Validity

These threats concern the generalization of our results. Our case study focused on the Java platform in this prototyping, given its significant presence in the microservice domain. However, this raises questions about the generalization of our results. Specifically, can we generate fine-grained user flow mappings for every technology and programming language? While microservice systems support heterogeneity, our case study shows that we can analyze a Java-based system following its standards, but we cannot conclude that the method is generalizable in the same setting. Nevertheless, our method, ReSSA, demonstrated potential results working for a C++ microservice benchmark in a customized pattern.

On the other hand, it is difficult for the method to work for applications that do not adhere to standard components and best practices. In such cases, modifications would be necessary to consider the application-specific naming conventions used across architectural layers.

5.4 Conclusion Validity

The threats concern whether the conclusions are based on the available data and metrics. We measured the accuracy between the manual analysis results and the produced outcome in our work, but we did not include the measurements of precision and recall because all operations that exist in the system were identified, which makes computing the precision and recall uninformative given 100% recall. Nevertheless, to examine the significance of the results statistically, more experiments are required over the data distribution.

Our study's trustworthiness may have been impacted by the restricted variation in the chosen system benchmark, restricting our conclusions to our obser-

vations in the case study. Nevertheless, we have made our approach available as open-source, which facilitates the replication of the study.

6 CONCLUSION

We presented a novel methodology for identifying the full set of entity CRUD operations needed for each endpoint in microservices-based systems. Such CRUD operations are fundamental to ensuring any proper endpoint access control. While the system may initially have had correct access rights on all endpoints, its distributed evolution can easily cause them to be outdated, requiring periodic verification of all applied access rights to prevent vulnerability. Our three-step solution generates a mapping of endpoint methods to their needed CRUD operations. It then identifies the call paths present within a system and synthesizes the preceding outputs to generate a mapping of all endpoint calls to the full set of implied entity CRUD operations that are needed to call this endpoint. We have approached the problem in a generalizable manner using ReSSA methodology simplifying the extension to accommodate new platforms by analyzing a unified intermediate program representation. We tested the proposed approach against the TrainTicket testbed, a Java microservice system built on the Spring-Boot framework, and verified the results against a manually-reconstructed ground truth. This yielded promising accuracy for identifying CRUD operations for each endpoint.

In future work, we will statically analyze business rules from the system source code and combine these two solutions to generate an authorization service enforcing authorization throughout the system uniformly. Such a service would allow the cross-cutting concern of authorization to be factored out and handled in an automated fashion by stopping any CRUD operation mismatches between an endpoint and what it executes before becoming a security vulnerability.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research, https://research.redhat.com.

REFERENCES

Bogner, J., Fritzsch, J., Wagner, S., and Zimmermann, A. (2021). Industry practices and challenges for the evolvability assurance of microservices. *Empirical Software Engineering*, 26(5):104.

Cerny, T., Svacina, J., Das, D., Bushong, V., Bures, M., Tisnovsky, P., Frajtak, K., Shin, D., and Huang, J. (2020). On code analysis opportunities and challenges for enterprise systems and microservices. *IEEE Access*, pages 1–22.

Das, D., Walker, A., Bushong, V., Svacina, J., Černý, T., and Matyas, V. (2021). On automated rbac assessment by constructing a centralized perspective for microservice mesh. *PeerJ Computer Science*, 7:e376.

Le, H. T., Nguyen, C. D., Briand, L., and Hourte, B. (2015). Automated inference of access control policies for web applications. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, SACMAT '15, page 27–37, New York, NY, USA. ACM.

Schiewe, M., Curtis, J., Bushong, V., and Cerny, T. (2022). Advancing static code analysis with language-agnostic component identification. *IEEE Access*, 10:30743–30761.

Son, S., McKinley, K. S., and Shmatikov, V. (2011). Rolecast: Finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 1069–1084, NY, USA. ACM.

Son, S., McKinley, K. S., and Shmatikov, V. (2013). Fix me up: Repairing access-control bugs in web applications. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society.

Srivastava, V., Bond, M. D., McKinley, K. S., and Shmatikov, V. (2011). A security policy oracle: Detecting security holes using multiple api implementations. *SIGPLAN Not.*, 46(6):343–354.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*. The Kluwer International Series In Software Engineering. Springer, Germany.

Xiang, C., Wu, Y., Shen, B., Shen, M., Huang, H., Xu, T., Zhou, Y., Moore, C., Jin, X., and Sheng, T. (2019). Towards continuous access control validation and forensics. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 113–129, New York, NY, USA. Association for Computing Machinery.

Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., and Zhao, W. (2018). Benchmarking microservice systems for software engineering research. In Chaudron, M., Crnkovic, I., Chechik, M., and Harman, M., editors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 323–324. ACM.