Using static analysis for microservices

Tomas Cerny
Baylor University (USA)
ABOUT

▪ Tomas Cerny
  ▪ Assistant professor at Baylor University
    ▪ last year in my tenure
  ▪ Research
    ▪ Cloud-native systems
    ▪ Static and dynamic analysis

▪ Long-term collaboration with Red Hat Research
  ▪ Over 35 Baylor students on research projects with Red Hat

NSF award #1854049 IRES Track I: U.S.-Czech Student Research Experience on Software Test Automation and Quality Assurance.
OUTLINE

Static analysis for microservices

▪ Why do we want this?
▪ What is the challenge?
▪ How do we address it
▪ What do we get out of this?
MOTIVATION

- In the context of Microservices, do you recognize what this is?
- Why do we need it, and how did we get it?
WHY DO WE GET THIS GRAPH?

- To understand how the holistic system works,
  - assess the dependencies, avoid a ripple effect, etc.
- get a centralized system perspective,
- reason about the system
HOW DO WE GET THIS GRAPH?

- We currently obtain this graph through dynamic system analysis
  - DevOps
  - OpenTelemetry, etc.
  - Event trace with a Correlation ID
  - We extract this from a deployed system

- Challenges
  - Traffic or tests needed
  - Delay: development vs. deployed system
  - No direct feedback to developers
  - Incomplete system coverage
COULD WE GET THIS GRAPH USING STATIC ANALYSIS?

- There would be no delay
  - DevOps/Developers
- No traffic needed
- More convenient for developers
- Code-change impact analysis

**Challenges**
- No tools for decentralized systems
- Language heterogeneity
- How do we connect it?
CHALLENGES IN MICROSERVICES

- Documentation quickly becomes outdated
- Poor dependency overview
  - No centralized view of the system
- Ripple effect
  - System-part change impacts other system parts
- Too complex systems
  - Heterogeneity | team coordination | the architecture itself
    - Descriptive vs. prescriptive architecture – are they the same?
- Separation of Duty: Architects / Dev Teams / DevOps

Industry practices and challenges for the evolvability assurance of microservices, DOI: 10.1007/s10664-021-09999-9
BASICS OF STATIC ANALYSIS

- Parsing code / codebase / bytecode / mining software repository
- Abstract Syntax Tree / Control Flow Graph / Program Dependence Graph
  - Or other Intermediate Representation (IR)

Applications
- Pattern matching
- Code style
- Bad smells
- Vulnerability checks
- Technical debt
- Other reasoning

Microservice code repositories
CHALLENGE: CULTURAL CLASH

STATIC ANALYSIS IN MICROSERVICES

- Static analysis
  - Plain ‘low-level’ code used
    - Low-level intermediate representation
  - Limited to a single codebase
    - Processing linearly and combining results does not work
      - Dependencies between microservices
  - Language-specific

- Microservices
  - Decentralized systems with decentralized codebases per Microservice
  - Heterogeneous system parts
INITIAL THOUGHTS

- How to represent a system?
  - Intermediate Representation?
  - Graph?

- Support for a single platform?
  - Yes, but…? Actually, no multiple..

- How do we merge microservices? Based on their dependencies?
  - If so, can we generalize it across platforms?

- We assessed common testbenches and frameworks for microservices
  - **Observation**: Common best practices are cross-platform applicable
WHAT DOES MICROSERVICE CODE LOOK LIKE?

**Observation:** code uses high-level structures -> components

- endpoints/controllers,
- services,
- repositories,
- **remote calls,**
- messaging,
- entities,
- data-transfer objects
- ...

What static analysis deals with?
- Low-level code, no components
- **Can we analyze code for components?**
WHAT ARE THE BEST PRACTICES?

▪ i.e., 12-Factor app tells us how to design, build and operate cloud-native systems.

▪ What else can we conclude?
  ▪ Many patterns for design, communication, robustness, resilience, routing, discovery, authorization/authentication, ..

▪ How do we use best practices in the source code?
  ▪ Components and high-level structures!
    ▪ Nice separation of concerns
WHAT ARE THE BEST PRACTICES?

▪ i.e., 12-Factor app tells us how to design, build and operate cloud-native systems.

▪ What else can we conclude?
  ▪ Many patterns for design, communication, robustness, resilience, routing, discovery, authorization/authentication, ..

▪ How do we use best practices in the source code?
  ▪ Components and high-level structures!
    ▪ Nice separation of concerns

Do static analysis tools operate with such components? No!
GOAL: IMPROVE STATIC ANALYSIS CAPABILITIES

- Recognize **high-level structures/components** in code
- **Combine** results across analyzed codebases
- Operate on **heterogeneous** platforms

Choosing the proper system **Intermediate Representation (IR)**
- If it was **component-based**, it likely fits many platforms
  - We use a **component dependency graph** as IR
1. How to operate on heterogeneous platforms
2. How can we recognize high-level structures?
3. How to combine multiple codebases
1. OPERATE ON HETEROGENEOUS PLATFORMS

- Low-Level Virtual Machine (LLVM)*
  - Designed around a language-independent Intermediate Representation (IR)
  - The IR serves as a portable, high-level assembly language that can be optimized with a variety of transformations
  - Meant for compilers – **removes the high-level language features**
    - Suits heterogeneous platforms
    - **Unsuitable for component detection**
- No good alternative exists that would make it simple to detect components.

*LLVM https://llvm.org/
1. OPERATE ON HETEROGENEOUS PLATFORMS

- Our response

Our own solution **Language-Agnostic Abstract Syntax Tree** (LAAST)

- Recently published at IEEE Access’22
- Using RUST language-based core that can parse multiple languages
  - The benefit of LAAST – all languages parsed into the same IR

---

1. Advancing Static Code Analysis With Language-Agnostic Component Identification, DOI: 10.1109/ACCESS.2022.3160485
2. RECOGNIZING HIGH-LEVEL STRUCTURES

Typically, components or coding conventions.

- **Generalized mechanism: component detectors**
  - Platform-specific patterns to detect components in platform unspecific approach

- **Language-Agnostic Abstract Syntax Tree**\(^1\) is an easy-to-traverse
  - Detect high-level structures through a set of generic parsers - “detectors,”
  - **Detectors**: Recursively visit tree nodes to check if expected properties exist on a given subtree to “match” a component.

- Some can be detected by annotation; others are more complex structures
  - i.e., inheritance, dependencies, specific properties or methods
  - Java vs. C++ vs. Python vs. Go

---

\(^1\)Advancing Static Code Analysis With Language-Agnostic Component Identification, DOI: 10.1109/ACCESS.2022.3160485
2. RECOGNIZING HIGH-LEVEL STRUCTURES

Sample: Class | LAAST and pattern matching | Resulting structure

- Evaluated on **TrainTicket** (ISCE | Java) and **DeathStarBench** (APLOS | C++) system testbeds
- Component detection: Precision 96-100%, Recall 86-100%

^[Advancing Static Code Analysis With Language-Agnostic Component Identification, DOI: 10.1109/ACCESS.2022.3160485]
3. CONNECTING COI

Having each codebase IR in the form of a component-dependency graph, we can combine them into a holistic system IR.

Three ingredients

1. **Inter-service calls** detection – can be very precise
2. Parsing **deployment descriptors** (i.e., docker files)
3. Microservice overlaps (bounded context/domain driven dev.)
   - Detecting overlaps in **data entities**
SUMMARY OF THE PROCESS

1. Analyze each system part codebase
   ▪ produce LAAST (auxiliary IR to face heterogeneity)

2. Detect components and extract component graph
   ▪ system part IR – i.e., single microservice

3. Combine component graphs based on various strategies
   ▪ holistic system IR – as if the system was a virtual monolith

SO WHAT..?
SUMMARY OF THE PROCESS

1. **Analyze** each system-part heterogenous codebase -> LAAST
2. **Detect components** to extract a system-part intermediate representation
3. **Determine** holistic system IR based on various ingredients

**Outcome**: System intermediate representation

Based on a component dependency graph

**SO WHAT..(?)..CAN WE DO WITH THIS?**

*Static analysis tools in the era of cloud-native systems,*
DOI: 10.48550/arXiv.2205.08527
EXPERIMENTAL EVALUATION
WHAT PROVED TO WORK

We have applied our proposed approach to many problems

1. Software Architecture Reconstruction (SAR)
2. Visualization of microservice system architecture
3. Reasoning about access policy consistencies (JSR-375) in microservices
4. Detecting microservice bad smells
5. Reasoning about microservice semantic code clones
1. SOFTWARE ARCHITECTURE RECONSTRUCTION (SAR)

Show the decentralized architecture as the system centric perspective

Phases: Extraction | Construction | Manipulation | Reasoning (analysis)

Views: Domain view | Technology view | Service view | Operation view

Fig. 1. Merged Domain View from TrainTicket

Fig. 2. Service View from TrainTicket

Fig. 3. Operation View from TrainTicket

On Automatic Software Architecture Reconstruction of Microservice Applications April 2021 DOI: 10.1007/978-981-33-6385-4_21
2. VISUALIZATION OF MICROSERVICES

Using SAR to extract a visual view facilitates human-centered reasoning.

- **Common direction conventional models**
  - Problems:
    - two-dimensional space; no interaction
    - does not fit the volume of microservices

Reconstructing the Holistic Architecture of Microservice Systems using Static Analysis, DOI: 10.5220/0011032100003200
2. VISUALIZATION OF MICROSERVICES

More ambitions

- Something to fit the volume of microservices
- Three-dimensional space
- Augmented reality
- Interaction
- Microvision prototype

https://www.youtube.com/watch?v=7arB UbglEko

Microvision: Static analysis-based approach to visualizing microservices in augmented reality, SOSE 2022
2. VISUALIZATION OF MICROSERVICES

- Fall 2022 student research
3. REASONING: ACCESS POLICY CONSISTENCY

We can detect **endpoint access policy enforcements**

- Components are recognized with all properties
  - i.e., JSR-375 Role-based access control
- Check whether different endpoints apply equivalent access policy
- Perform across microservices (through inter-service communications)
  - Detect consistency errors across microservices

On automated RBAC assessment by constructing a centralized perspective for microservice mesh, DOI:10.7717/peerj-cs.376
3. ACCESS POLICY CONSISTENCY

We can detect enforced endpoint access policy

- Components recognized with all properties
  - i.e., JSR-375 Role-based access control
- Determine access policy equivalence across different endpoints
  - Perform across microservices
    - (through inter-service communications)
  - Detect consistency errors

On automated RBAC assessment by constructing a centralized perspective for microservice mesh, DOI:10.7717/peerj-cs.376
3. ACCESS POLICY CONSISTENCY

Detecting 5 violations

- Missing role violations
- Unknown access violations
- Entity access violations
- Conflicting hierarchy violations
- Unrelated access violations
4. MICROSERVICE BAD SMELLS

Having **System IR** and reading

“On the Definition of Microservice Bad Smells, DOI: 10.1109/MS.2018.2141031”

Pattern matching on the holistic **system IR**
- Detecting 11 bad smells.
- MSANose tool
5. SEMANTIC CODE CLONES

With too much development autonomy, or upon system integration.

- Certain features might coexist but are hidden in heterogeneity.

**Syntactic clone**

- Looks the same / does it do the same thing

- Approach using **system IR** considering components in control flow as a heuristic to narrow our similarity identification, then detecting which operations we perform with data and whether the data seem similar.
  - High Accuracy received on TrainTicket Benchmark
  - List of microservices/endpoints that are similar, ordered by similarity

Semantic Code Clone Detection Method for Distributed Enterprise Systems, DOI: 10.5220/0011032200003200
5. SEMANTIC CODE CLONES

Figure 1: Schema of the algorithm.

\[ \text{sim}(a_i, b_i) = \text{ctr}(a_i, b_i) + \text{rfc}(a_i, b_i) + \text{rp}(a_i, b_i) \]

<table>
<thead>
<tr>
<th>Nr</th>
<th>MS A</th>
<th>MS B</th>
<th>Sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ts-contact</td>
<td>admin-basic-info</td>
<td>37.5%</td>
</tr>
<tr>
<td>2</td>
<td>ts-config</td>
<td>ts-train</td>
<td>16.6%</td>
</tr>
<tr>
<td>3</td>
<td>ts-config</td>
<td>admin-basic-info</td>
<td>16.6%</td>
</tr>
<tr>
<td>4</td>
<td>ts-config</td>
<td>ts-travel2</td>
<td>33.3%</td>
</tr>
<tr>
<td>5</td>
<td>ts-config</td>
<td>ts-travel</td>
<td>33.3%</td>
</tr>
<tr>
<td>6</td>
<td>ts-order-other</td>
<td>ts-order</td>
<td>87.5%</td>
</tr>
<tr>
<td>7</td>
<td>ts-preserve</td>
<td>preserve-other</td>
<td>50.0%</td>
</tr>
<tr>
<td>8</td>
<td>ts-security</td>
<td>ts-train</td>
<td>50.0%</td>
</tr>
<tr>
<td>9</td>
<td>ts-security</td>
<td>ts-seat</td>
<td>16.6%</td>
</tr>
<tr>
<td>10</td>
<td>ts-train</td>
<td>ts-seat</td>
<td>16.6%</td>
</tr>
<tr>
<td>11</td>
<td>ts-train</td>
<td>ts-travel2</td>
<td>16.6%</td>
</tr>
<tr>
<td>12</td>
<td>ts-train</td>
<td>ts-travel</td>
<td>16.6%</td>
</tr>
<tr>
<td>13</td>
<td>ts-travel2</td>
<td>ts-travel</td>
<td>66.6%</td>
</tr>
</tbody>
</table>

Figure 2: Example of control-flow graph.
WHAT DID WE DEMONSTRATE?

- Static analysis can be beneficial to decentralized system analysis.
- It can do much more! I.E., help developers with codebase changes.
WHAT DID WE DEMONSTRATE?

- Robustness of our System IR to various tasks for Microservices
  - Targeting problems/gaps in the Microservices

- Possibly foundation to holistic static analysis of Microservices

- With such promising results, we can broaden our future research.
CONCLUSION

▪ Arguing why static analysis is not used in cloud-native systems
▪ Recognizing barriers to progress
▪ Introducing our experimental solution
  ▪ System IR based on component awareness
▪ Sharing promising evaluation
▪ Asking you to contribute!
  ▪ https://github.com/cloudhubs
FUTURE WORK

- Architectural Degradation and Technical Debt detection
- Improve component and component dependency parsing
  - Broaden language support beyond Java/C++/Go
  - Heterogenous system benchmark study
- Messaging integration to get a more comprehensive perspective
  - work in progress
- Integration with dynamic analysis
- Continuous restructuring of microservices
  - IEEE SOSE 2022 Best Paper Award for Soft K-means approach

\(^2\)Monolith to Microservices: VAE-Based GNN Approach with Duplication Consideration, SOSE 2022
WHAT DID WE MISS?

Questions?
Post your questions/remark to Tomas_Cerny@baylor.edu