ENHANCING RELIABILITY AND PERFORMANCE OF DATA-CENTRIC SYSTEMS WITH STATIC ANALYSIS

by

CHENGPENG WANG

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy
in Computer Science and Engineering

December 2023, Hong Kong

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

_____

CHENGPENG WANG

ENHANCING RELIABILITY AND PERFORMANCE OF DATA-CENTRIC SYSTEMS WITH STATIC ANALYSIS

by

CHENGPENG WANG

This is to certify that I have examined the above Ph.D. thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the thesis examination committee have been made.

Prof. Charles ZHANG, Thesis Supervisor

Prof. Xiaofang ZHOU, Head of Department

Department of Computer Science and Engineering

11 December 2023

DEDICATION

To My Family
and
My Friends

ACKNOWLEDGMENTS

First, I would like to express my deepest gratitude to my supervisor, Prof. Charles Zhang, for his unwavering support and guidance throughout my PhD journey. During my first year, when I encountered challenges with my initial research project, he always encouraged me with the saying: "stay hungry and stay humble." This phrase helped me stay calm and persevere through the challenging early stages of my research. Prof. Zhang's far-reaching vision has been instrumental in helping me find good research problems and pursue research with high impact.

I want to express my appreciation for the participation of the committee members of my thesis defense: Prof. Shing-Chi Cheung, Dr. Shuai Wang, and Prof. Xiangyu Zhang. Shing-Chi has been a constant support and inspiration throughout my PhD career. His passion for research and careful attention to detail have left a lasting impression on me. I am also grateful to him for providing a recommendation letter for my post-doc application. Shuai facilitated communication between research groups and drove deep discussions on software engineering and security. Finally, I would like to thank Xiangyu for serving as my defense committee member and offering me a post-doc position after graduation. His open and inclusive research attitude has encouraged me to explore new directions and produce interesting and practical work.

I am also grateful to several professors in the PL/SE group at HKUST: Dr. Jiasi Shen, Dr. Lionel Parreaux, and Dr. Amir Goharshady. Jiasi has been consistently supportive, providing insightful suggestions and comments on my ideas, especially those related to program synthesis and transformation. I will never forget her guidance on my post-doc applications and her efforts in organizing SEPL seminars. Lionel and Amir have broadened my perspective on diverse topics within the programming languages community through their courses, which has greatly benefited my research. I also want to show my appreciation to Prof Ke Yi and Dr Wei Wang, who served as the committee members of my qualifying examination, leading meaningful discussions on static analysis from both theoretical and practical perspectives.

I would like to express my gratitude to the members of our Prism group, includ-

ing Rongxin Wu, Gang Fan, Qingkai Shi, Peisen Yao, Wensheng Tang, Yongchao Wang, Heqing Huang, Yiyuan Guo, Yushan Zhang, Lingjie Huang, Anshunkang Zhou, Yuandao Cai, Maryam Masoudian, Jiajun Gong, Hao Ling, Kexin Ma, Hung Chun Chiu, Chengpeng Li, Xiangzhe Xu, Wei Chen, Bowen Zhang, Chengfeng Ye, Sixiang Peng, Yibo Jin, Shuhao Fu, Yuzhang Zhu, Yuheng Tang, Jiaxin Song, and Bangyan Du. I am honored to have worked with them, and their criticism and constructive suggestions in group discussions have helped me reshape and solidify my research ideas before paper submissions. I also want to express my respect and appreciation for more senior Prismers and the members of our start-up Sourcebrella Inc: Xiao Xiao, Jinguo Zhou, Xiaoheng Xie, and Wenyang Wang. Their efforts in implementing the static analysis infrastructure Pinpoint facilitated our trials of many early-stage ideas and ultimately led to the success of applying our research outcomes to real-world production.

Thanks also go to several friends in different research groups at HKUST, especially Huaijin Wang, Zhibo Liu, Yuanyuan Yuan, Zhenlan Ji, Zongjie Li, Dongwei Xiao, Jiarong Wu, Wuqi Zhang, Jialun Cao, Hengcheng Zhu, Xiao Li, Suyi Li, Jipeng Zhang, Runtao Liu, Shangyu Li, and Kai Wang. Particularly, I want to thank Kai Wang for his support and encouragement when I made little progress. His expertise in the database community also inspired me to form a holistic view of analyzing data-centric systems. Besides, Jipeng and Runtao led me to the world of large language models, offering valuable suggestions on applying the new advances in LLMs to static analysis problems.

Finally, I would like to thank my beloved parents, my uncle and aunt, my brother Qiran, and my friend Jianfeng. Their support has been invaluable in helping me persevere through difficult times. Without their understanding and tolerance, I would not have pursued my research dream and produced this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ENHANCING RELIABILITY AND PERFORMANCE OF DATA-CENTRIC SYSTEMS WITH STATIC ANALYSIS

by

CHENGPENG WANG

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

ABSTRACT

In the era of big data, data-centric systems have emerged as the fundamental infrastructure for processing, storing, and transmitting various forms of data, providing diverse services in our daily lives. The widespread adoption of data-centric systems highlights the critical need for enhancing their reliability and performance. Unreliable or inefficient data-centric systems can result in unanticipated economic losses and unnecessary consumption of computational resources, jeopardizing property safety and compromising the overall service experience.

This thesis provides a comprehensive analysis of data-centric systems using static analysis techniques. The research is centered around three critical components, namely the application, library, and database sides of the systems. By delving into data organization, propagation, manipulation, and validation, our techniques can successfully identify vulnerabilities and optimize computation, leading to enhanced reliability and performance of data-centric systems in a holistic manner.

The first part of our research focuses on ubiquitous data structures, called containers,

and improves the system performance by organizing data with efficient container types. We introduce CRES, a synthesizer designed to replace inefficient container types for a given program. CRES statically identifies container usage and selects methods with lower time complexity for each container method call, ultimately discovering a more efficient container type for each container object. CRES reduces execution time by 8.1% on average in our experimental subjects while theoretically preserving program behavior.

The second aspect of our research investigates the data propagation in the application code to improve the system reliability. The erroneous values through containers necessitate precise and efficient reasoning about container memory layout. To address this, we introduce ANCHOR, which utilizes memory orientation analysis to apply strong updates to container memory layouts and conducts reachability analysis. It is shown that ANCHOR detects 20 null pointer exceptions with only 9.1% as its false-positive ratio and finishes analyzing 5 MLoC within five hours. Its high precision and efficiency of bug detection demonstrate its potential to improve system reliability from the application side.

The third part of our research investigates the data manipulation conducted by library APIs. We propose DAINFER, an algorithm that identifies store-load APIs and derives API aliasing specifications from library documentation. Equipped with NLP models, DAINFER effectively interprets informal semantic information and achieves an efficient API aliasing specification inference with a precision of 79.78% and a recall of 82.29%. The inferred specifications can effectively benefit downstream analyses to derive fundamental program facts, such as value-flow and alias facts, further promoting bug detection and program optimization.

The final part of our research shifts to domain-specific programs, named data constraints, to investigate data validation upon databases. While data constraints play a crucial role in ensuring data correctness, the presence of equivalent ones can result in the waste of computational resources. To tackle this issue, we present EQDAC, an efficient decision procedure that utilizes two lightweight analyses to refute or prove the data constraint equivalence in polynomial time. It is demonstrated that EQDAC discovers 11,538 equivalent pairs from 30,801 data constraints in the Ant Group and uncovers 7,842 redundant data constraints. It successfully alleviates redundant data validation and reduces the CPU time by 15.48% from the database side.

CHAPTER 1

INTRODUCTION

In recent years, data-centric systems have gained significant popularity, particularly in industrial production settings. These systems serve as the backbone of modern computing infrastructure by enabling people to process, store, and transmit data. Common examples of data-centric systems include electronic health record (EHR) systems, geographic information systems (GIS), and financial technology (FinTech) systems. For example, Ant Group, a prominent FinTech company in China, has developed and deployed a wide range of FinTech systems to provide financial services to both individuals and small/medium-sized enterprises.

The widespread adoption of data-centric systems is considered a defining characteristic of the big data era. Unfortunately, it is far from trivial to develop a data-centric system in real-world scenarios. On the one hand, an unreliable data-centric system would cause data corruption and even yield non-estimable economic loss, for instance, in FinTech systems. On the other hand, an inefficient system would degrade the experience of users and also introduce a large amount of computation resource consumption. Thus, it is of critical importance to improve the reliability and performance of data-centric systems.

Notice that the instrumentation, deployment, and execution of data-centric systems are often fraught with substantial difficulties. In this context, static analysis emerges as a promising approach to achieving our goals. As an important program analysis methodology, static analysis has succeeded significantly in different problem domains, including static bug hunting [3, 4], program optimization [5, 6], and program refactoring [7, 8]. However, existing techniques are still inadequate for effectively analyzing data-centric systems. Specifically, there are several unique characteristics of the systems that are paid little attention to by existing static analysis techniques. For illustrative purposes, we present the main components of a data-centric system in Figure 1.1, which exemplifies the following characteristics.

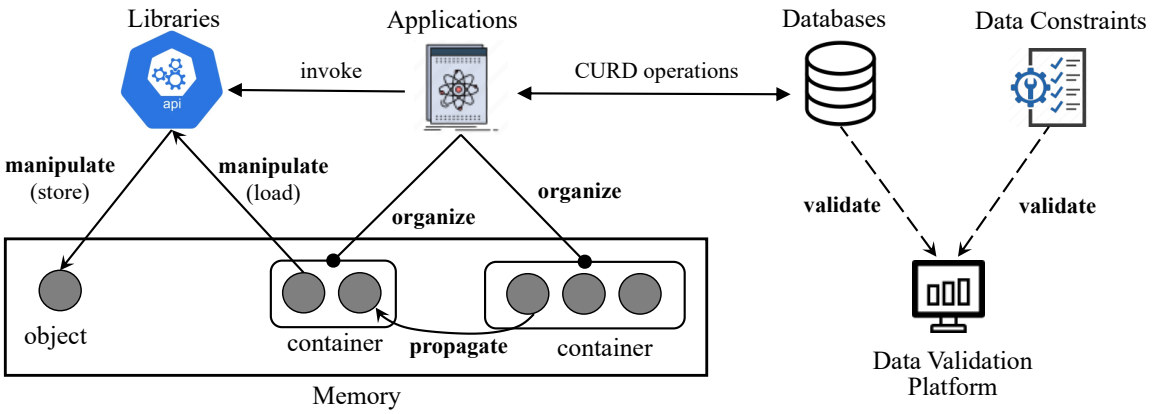


Figure 1.1: Main components of a data-centric system

- First, the application code of the systems can *organize* the data as memory objects with various data structures and *propagate* the data via the methods of data structures. Typically, after retrieving data records from databases or receiving inputs from users, the applications can store them in specific fields of user-defined classes and organize them with specific abstract data types, such as lists and other kinds of containers [9, 10, 11]. Any abnormal data propagation and inefficient data organization can introduce the reliability and performance issues of the systems, respectively.

- Second, data-centric applications can heavily rely on third-party libraries [12, 13]. Concretely, library APIs enable the developers to *manipulate* data upon memory with load and store operations. By storing and loading data at the same memory location, library APIs can make the application code propagate data within memory, which further affects the behavior of applications. Hence, it is crucial to investigate how library APIs manipulate data upon memory if we want to obtain fundamental program facts of applications to enhance reliability and performance.

- Third, data-centric systems often leverage domain-specific programs, called data constraints, to facilitate *data validation* to ensure data correctness [14, 15, 16]. These programs are typically executed on database tables, scrutinizing substantial data during the system runtime. However, the absence of practical techniques for managing data constraints can result in redundant computations on the database side, leading to increased resource consumption and potential impacts on system throughput.

To tackle the aforementioned characteristics of data-centric systems, we put forward a set of novel static analysis techniques that examine **data organization, propagation, manipulation, and validation** from the vantage points of **applications, libraries, and databases**. Through our research, we can effectively identify vulnerabilities and enhance the computational efficiency of these systems, ultimately resulting in data-centric systems that are more reliable and efficient.

1.1 Motivation

As stated earlier, a data-centric system can consist of various components such as applications, libraries, and databases. In the following sections, we present four observations related to data organization, propagation, manipulation, and validation within the systems. These observations serve as motivation for designing new static analysis techniques aimed at enhancing system reliability and performance.

1.1.1 Data Organization: Inefficient Container Types

A data-centric application often interacts with other applications or database engines to exchange data, and then organizes the data as objects in the memory with a specific kind of data structures, namely containers. There are a large number of libraries providing different implementations of containers, such as Java Collections Framework [10], Apache Commons Collections [11], Eclipse Collections [17], and Fastutil [18] in Java open-source community. Although different containers and their methods may have similar and even the same functionalities, they can differ greatly in terms of performance. Figure 1.2 shows a typical example of inefficient container usage in the project google-http-java-client. Obviously, the same functionality can be implemented by using LinkedList, which can avoid the overhead introduced by memory reallocation. As reported by previous study [19], replacing the ArrayList object with a LinkedList one achieves 46% speed-up against the same test set, showing the great potential of container type optimization in enhancing the performance of a data-centric system.

Inefficient container types widely exist in real-world data-centric applications. Developers are often not familiar with all the available container types and their methods, and

```
1 <T> List<T> getAsList(T value) {
2   if (value == null)
3     return null;
4   List<T> result = new ArrayList<T>();
5   result.add(value);
6   return result;
7 }
```

Figure 1.2: Inefficient container type usage in google-http-java-client

particularly, unaware of exchangeable container types and their performance difference, which can introduce inefficient container types in the development. To avoid inefficient container types, we need to propose a systematic solution to detect inefficient container types and optimize them with container replacement, which eventually yields a more efficient way of organizing memory objects. Concretely, we have to guarantee the behavioral equivalence of the programs during the container replacement and try to reduce the computation resource consumption of the program for large inputs.

1.1.2 Data Propagation: Indirect Value-Flows through Containers

As illustrated above, containers are widely utilized in data-centric application code, which convenience developers to implement the application logic without implementing low-level memory operations. More specifically, leveraging the methods offered by the containers, developers can propagate the stored objects across different functions and modules in the applications. Figure 1.3 shows a typical example extracted and simplified from the code in the project Hibernate-ORM. The method performList fetches a list from each SelectQueryPlan object and then inserts all the elements to the ArrayList results. Then, the method doList traverses the list returned by the method performList and collects a bounded number of non-redundant elements. The data in each collected element is stored in the list tmp. Actually, the code example demonstrates the common phenomenon that the values in the data-centric systems can be propagated by the container methods, such as the methods add and get of ArrayList. The objects retrieved from the query at line 4 are propagated across the functions and finally returned to the caller function of the method doList.

The indirect value flows induced by containers can cause vulnerabilities in the applications, demonstrating the necessity of understanding data propagation introduced by containers. In Figure 1.3, if there is a null element stored in the list at line 5, the invo-

```
1 public List<R> performList(Context ctx) {
2 List<R> results = new ArrayList<>();
3
4 for (SelectQueryPlan<R> plan : Plans) {
5   List<R> list = plan.getList(ctx);
6   int size = list.size();
7   if (size <= maxRowsJpa) {
8     for (int i = 0; i < size; i++) {
9       results.add(list.get(i));
10      }
11    }
12  }
13  return results;
14}
```

```
15 List<D> doList(Context ctx) {
16   List<R> list = performList(ctx);
17   List<D> tmp = new ArrayList<>();
18   Set<Object> distinct = new HashSet<>();
19   for (R r : list) {
20    if (distinct.add(result)) {
21      includedCount++;
22      if (includedCount > bound)
23        continue;
24      tmp.add(r.getData());
25    }
26  }
27  return tmp;
28}
```

Figure 1.3: An example of container usage in Hibernate-ORM

cation of the method getData at line 24 can introduce the null pointer exception (NPE). Meanwhile, the sensitive data stored in the elements of list at line 5 can be propagated to the returned list of the method doList, which may cause sensitive information leakage in the caller of the method doList. Therefore, Identifying value flows through containers is important for detecting a wide range of vulnerabilities, including but not limited to memory corruption and sensitive information leaks. However, it is stunningly challenging to reason such value flows with high precision and efficiency. Because the elements in the containers are stored in a sequence or as key-value pairs, reasoning value flows through containers requires precise modeling of container memory layouts, which can not be efficiently achieved by existing static analysis techniques [20, 21].

1.1.3 Data Manipulation: Store-Load Library APIs

Data-centric applications often rely heavily on various libraries to process the data with library APIs [12, 13]. Although library APIs significantly convenience the development process, their prevalence highlights the necessity of understanding library semantics in analyzing the application code. Notably, there exists a particular class of library APIs that store or load the data in the memory. When two specific methods conduct the load and store operations over an inner field of the data structure, they can introduce the aliasing relations between their parameters and return values [22, 23]. We show several examples in Figure 1.4. The first example is ArrayList in Java Collections Framework. If we invoke the add before the get upon the same ArrayList object, the return value of the latter may be aliased with the parameter of the former, which is shown in Figure 1.4(a). In more specific-

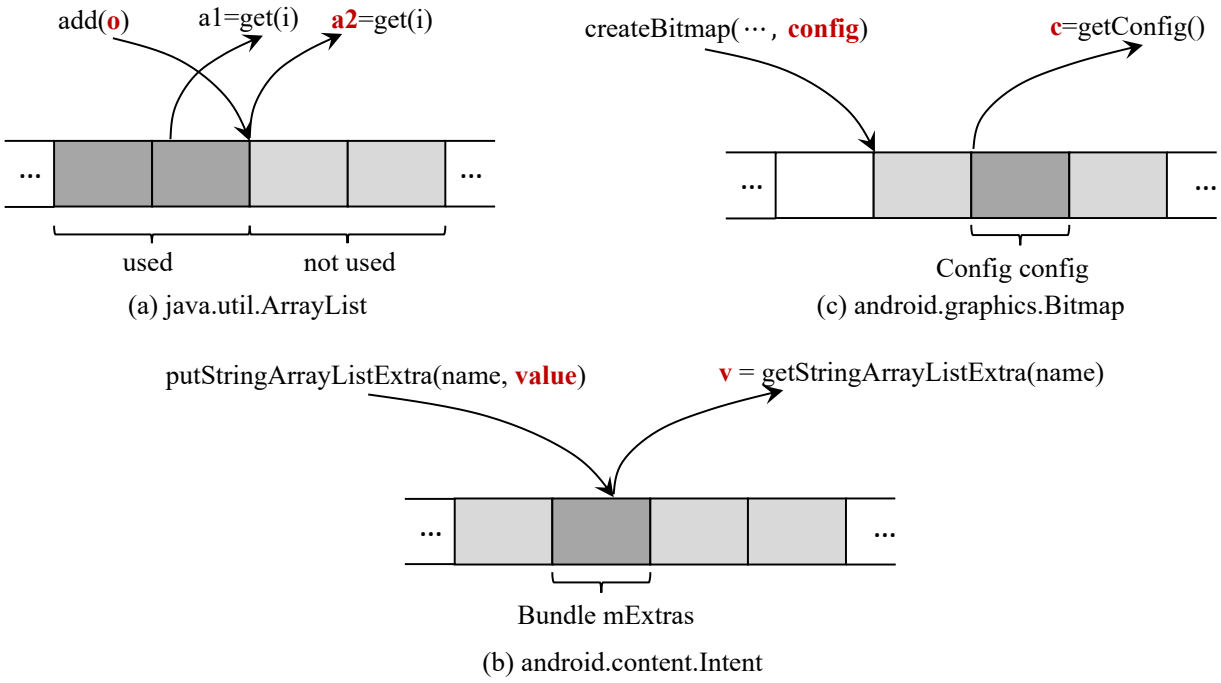


Figure 1.4: Examples of store/load library APIs

purpose data structures, such as the class Intent and Bitmap offered by Android platform [24], there are also many APIs conducting the memory store and load operations. In the class Intent, for instance, the API putStringArrayListExtra stores a pair of name and values into the Bundle object in the Intent object, while the API getStringArrayListExtra retrieves the value corresponding to a specific name. As shown in Figure 1.4(b), if their first parameters are aliased, the return value of the latter may be aliased with the second parameter of the former when they are invoked upon the same Intent object one after another. Another similar example is that the API getConfig can return the aliased value of the corresponding parameter of createBitmap in the class Bitmap, which is demonstrated in Figure 1.4(c).

With a similar motivation of analyzing indirect-value flows through containers, we have to identify the store-load library APIs and understand induced aliasing relations. Different from containers, such classes offered by various libraries cover a wide range of program domains, making it impractical to specify their specifications manually. To promote downstream clients upon data-centric applications using libraries, we need to propose an effective solution to infer their API aliasing specifications, which can eventually benefit bug detection [25, 3] and program optimization [26].

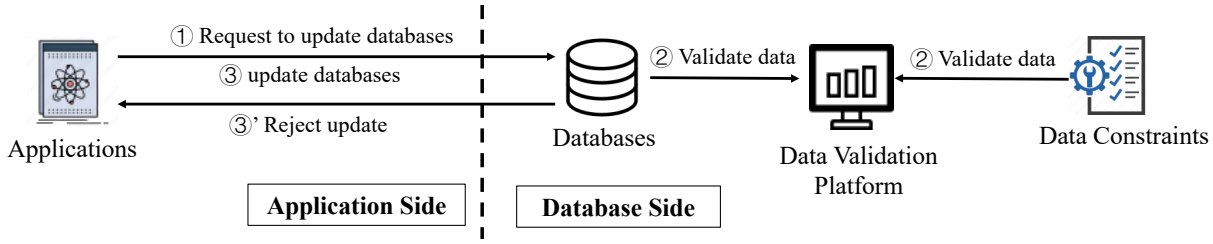


Figure 1.5: The workflow of data validation in the database side

1.1.4 Data Validation: Redundant Data Constraints

As security-critical systems, data correctness has always been a central concern of data-centric systems. In industrial settings, developers often define and execute domain-specific programs known as data constraints over databases to ensure data correctness during runtime [14, 15, 16]. Such data constraints can be specified in various languages, including SQL and other domain-specific languages. Figure 1.5 shows the workflow of data validation in the database side of a system. When an application attempts to update data in a database, a data validation platform validates whether the update violates any data constraint or not. If a data constraint is violated, the database update will be rejected. Otherwise, the update is permitted. It has been recognized that such workflow of data validation is an effective mechanism for ensuring data correctness in data-centric systems.

As a data-centric system evolves, the system specification becomes increasingly complex, leading developers to write numerous data constraints for data validation [16]. However, developers often face the difficulty of determining whether a new data constraint has already been specified. To mitigate the risk of data incorrectness, they tend to execute data constraints aggressively, even if they may be equivalent to existing ones. The redundancy in data constraints has become a form of "technical debt" within data-centric systems, resulting in wasteful consumption of computational resources during data validation.

To resolve equivalent data constraints, we need to introduce a new technique to support the developers reasoning the equivalence relation among a large number of data constraints in the system. To be more specific, we need to design an efficient decision procedure to determine the equivalence relation of data constraints, which can assist the developers in equivalence clustering and searching over data constraints. Benefiting from the decision procedure, the data constraint checking engine only needs to examine one

data constraint per equivalence cluster, and meanwhile, developers would be notified with equivalent data constraints upon data constraint submission, which can effectively reduce resource consumption and improve the performance of the system.

1.2 Contribution

This thesis targets the enhancement of reliability and performance by analyzing data-centric systems from four aspects, namely data organization, propagation, manipulation, and validation. Specifically, we make four technical contributions, which include complexity-guided container replacement synthesis, container-aware value-flow analysis, documentation-based API aliasing specification inference, and data constraint equivalence verification. In what follows, we elaborate our contributions in detail.

1.2.1 Complexity-Guided Container Replacement Synthesis

To optimize data organization in the application code, we present CRES, a container replacement synthesizer that automatically discovers inefficient container types and replaces them with efficient ones. Established upon a novel container semantic abstraction, CRES leverages pointer analysis to identify the container usage intentions, which guides CRES to narrow down container types and methods preserving program behavioral equivalence. Meanwhile, CRES employs complexity specifications of container methods to guide an enumerative search during the synthesis process, which makes the new programs more likely to be more efficient with large inputs. CRES has discovered over one hundred inefficient container usages in data-centric applications, and the synthesized replacements can improve the system performance by 8.1% on average.

1.2.2 Container-Aware Value-Flow Analysis

Reasoning container memory layouts is a crucial prerequisite for performing container-aware value-flow analysis, which uncovers the data propagation within data-centric applications. In this thesis, we introduce ANCHOR, a fast and precise value-flow analysis framework that incorporates precise container reasoning. Based on given container

semantic specifications, ANCHOR effectively models the semantics of container method calls and seizes the opportunity of applying strong updates to container memory layouts for precision enhancement. It supports various clients of value-flow analysis, including program slicing and detecting a wide range of value-flow bugs, significantly improving system reliability from the application side. Our evaluation demonstrates that ANCHOR successfully detects 20 null pointer exceptions with only 9.1% as its false-positive ratio and enables analysis of MLoC within a few hours. To the best of our knowledge, ANCHOR is the first container-aware value-flow analysis approach that achieves high precision, efficiency, and scalability simultaneously.

1.2.3 Documentation-based API Aliasing Specification Inference

To promote understanding of the behaviors of applications using libraries, we introduce DAINFER, an algorithm for inferring API aliasing specifications. DAINFER examines the data manipulation performed by library APIs and infers the aliasing relationships between API parameters and return values. Unlike existing studies, DAINFER leverages the library documentation for specification inference instead of any forms of programs. Established upon a tagging model and a large language model, DAINFER effectively interprets the informal semantic specification in the documentation. Besides, the class-hierarchy relation and API type signatures narrow down the aliasing pairs, enabling us to achieve high efficiency in the inference. It is shown that DAINFER infers the API aliasing specifications with a precision of 79.78% and a recall of 82.29%. The obtained aliasing specifications further facilitate alias analysis, revealing 80.05% more alias facts for API return values, and meanwhile, support taint analysis, identifying 85 more taint flows in 23 Android apps.

1.2.4 Data Constraint Equivalence Verification

We finally propose an efficient decision procedure EQDAC to verify the data constraint equivalence, which improves the system performance from the database side by eliminating redundant computation in the data validation. Although the equivalence verification problem is NP-hard, EQDAC can verify the equivalence for most real-world instances with two light-weighted analyses in polynomial time. Equipped with EQDAC, develop-

ers can efficiently conduct the equivalence clustering and searching upon data constraints. Notably, the soundness and completeness of EQDAC enable the system performance optimization without sacrificing the system's reliability. Our effort shows the great potential value of formal methods in optimizing data-centric systems.

1.3 Outline

The thesis is organized as follows. Chapter 2 introduces the background knowledge of data-centric systems and discusses the existing efforts, revealing the gap between real-world demands and existing techniques. Chapter 3 presents our synthesis framework CRES to identify and replace inefficient container usages, which improves the performance of the systems from the application side by optimizing data organization. Chapter 4 presents our value-flow analysis framework ANCHOR to precisely reason the value flows through containers, which supports value-flow bug detection to improve the reliability. Chapter 5 introduces our inference algorithm DAINFER that derives API aliasing specifications from library documentation, which promote the bug detection and program optimization from the perspective of library understanding. Chapter 6 demonstrates our decision procedure EQDAC that verifies the equivalence of data constraints efficiently, enabling the elimination of redundant data validation for performance enhancement. We finally conclude the thesis and discuss several promising future directions in Chapter 7.

The thesis summarizes the research papers published in the top-tier conferences and journals. The content of Chapter 3 has been published in [27]. The content of Chapter 4 has been published in [28]. The work demonstrated in Chapter 5 is currently under review. The content of Chapter 6 has been published in [29].

CHAPTER 2

BACKGROUND AND PREVIOUS STUDIES

This chapter presents the background of this thesis and discusses previous studies. First, Section 2.1 introduces data-centric systems, and especially discusses three important entities, namely containers, store-load API pairs, and data constraints, to demonstrate the data organization, propagation, manipulation, and validation from the application, library, and database sides. Particularly, we highlight the importance of reasoning their semantics of three entities in enhancing the system's reliability and performance. Second, Section 2.2 discusses four lines of existing effort, which provides meaningful inspiration on container replacement synthesis, container-aware value flow analysis, API aliasing specification inference, and data constraint equivalence verification.

2.1 Data-Centric System

Data-centric systems are a common category of systems in industry and play an essential role in our daily lives. As the backbone of our computing infrastructure, they support processing, storing, and transmitting various forms of data. Any abnormal or inefficient computation can degrade the reliability and performance of the systems, which has a significant impact on property safety, user experience, and resource consumption. In this thesis, we aim to ensure the reliability and improve the performance of data-centric systems. Specifically, we concentrate on three typical characteristics of data-centric systems, which distinguish them from other kinds of software systems. First, the developers utilize various data structures, especially containers, to organize the data in the memory and propagate the data across functions and modules by invoking the methods offered by data structures. Second, data-centric applications often heavily rely on various libraries to manipulate the memory, for example, conducting the store and load operations, which introduce indirect data flows in the application code. Third, data-centric systems often ensure data correctness by dynamically validating data before insertion into the database,

while the examined properties can be formulated as domain-specific programs, such as data constraints. To understand the behavior of the systems, we have to investigate the following three questions.

- (1) How do the developers organize and propagate the data in the applications?
- (2) How do library APIs manipulate the data in the memory?
- (3) How is the data verified to ensure correctness in the database?

In what follows, we introduce three important entities in the data-centric systems, namely containers, store-load API pairs, and data constraints, to demonstrate how they support organizing, propagating, manipulating, and verifying data, respectively. In particular, we highlight how they affect the system's reliability and performance, which motivates our research from the application, library, and database sides, respectively.

2.1.1 Container

Concept. Containers, as abstract data types, are widely used in data-centric application code. They enable developers to organize memory objects without implementing low-level memory operations. Apart form Java Collections Framework (JCF) [10], a variety of Java third-party libraries provide different container implementations, such as Apache Commons Collections (ACC) [11], Eclipse Collections (EC) [17], and Fastutil [18]. No matter how container types are defined, they can be formulated as the following structures from an abstract perspective.

Definition 2.1.1 *(Container) A container is a pair* (O, M)*, where*

- O *is a set of index-value pairs, which indicates the memory layout of inner storage.*
- M *is a set of methods. For each* $\mathrm{m} \in \mathrm{M}$*, the method call of* m*, denoted by* $\mathrm{r} = \mathrm{O}.\mathrm{m}(\mathbf{v})$*, induces an manipulation over* O *and yields* $(\mathrm{O}', \mathrm{r})$*, where* $\mathbf{v}$ *indicates the parameters,* O' *is a new memory layout, and* r *is the return value.*

Table 2.1: Examples of containers in Java

Containers	Frameworks	Layouts	Examples of Methods
ShortArrayList	High Performance Primitive Collections	list	short remove(int), void add(short)
LinkedList	Java Collections Framework	list	boolean offerLast(O)
HashSet	Java Collections Framework	set	boolean add(O), boolean contains(O)
TreeSet	Java Collections Framework	set	boolean add(O), boolean remove(O)
HashMap	Java Collections Framework	map	O put(O, O), boolean containsKey(O)
CollectionBag	Apache Commons Collections	map	O get(O), int getCount(O)

Definition 2.1.1 provides a uniform formulation for different container types. According to different forms of the indexes, the container types can be further categorized into three kinds, namely lists, sets, and maps. Specifically, we have

- A list takes non-negative integers as indexes showing the elements' positions.
- A set takes the stored elements as the values and their hash codes as the indexes.
- A map takes hashable objects as the indexes and the paired objects as the values.

Other container types can be regarded as instantiations of lists, sets, or maps. For example, a bag can be regarded as a specific form of a map, of which the indexes are the stored elements, and the values are the multiplicities of the elements. Besides, several data structures are essentially containers, even if their names do not indicate that they are lists, sets, or maps. For instance, the class HttpSession in the JavaEE framework is essentially a map as it organizes its elements as a key-value pair. From a concrete perspective, such data structures provide the methods to instantiate the set of methods M in Definition 2.1.1, which supports programmers in conducting desired operations over memory.

Example 2.1.1 *Table 2.1 shows the container examples in Java.* ShortArrayList *in the High Performance Primitive Collections (HPPC) library and* LinkedList *in the JCF store their elements in the lists and provide a series of operations over lists, such as* remove*,* add*, and* offerLast*. Besides, JCF also provides* HashSet*,* TreeSet*, and* HashMap *to store hashable objects. Moreover, the ACC library offers* CollectionBag*, which is essentially a map mapping its elements to the multiplicities.*

Relevance to System Reliability. As shown in Figure 1.3 in Section 1.1, data can be propagated through container objects, forming value flows in the application code. Any

erroneous value flow can cause the system to behave abnormally and even crash. For example, a null value flowing to the dereferenced operand can introduce a null pointer exception (NPE) in the application code. Other typical value-flow bugs include memory leaks [4], use-after-free [30], and taint vulnerabilities [25]. The existence of value-flow bugs greatly affects the reliability of the data-centric systems, introducing memory safety issues or the leak of sensitive information.

To detect value-flow bugs precisely and effectively, we need to consider the semantics of each container method and reason how each element is organized in a container. We will demonstrate the related works on different memory models of value-flow analysis in Section 2.2.2, which shows the research gap between existing efforts and the container-aware value-flow analysis we need in the presence of data-centric systems. Our solution will be demonstrated in Chapter 4.

Relevance to System Performance. A large number of third-party libraries provide a wide range of containers with similar functionalities, while their performance profiles can differ greatly. As demonstrated by Figure 1.2, choosing a proper container type may accelerate the execution by around 50% upon a workload in a test case. However, the developers are often unaware of alternative container types under their programming context and, thus, can hardly choose the optimal container type and their methods to organize and manipulate the objects in the memory efficiently, which slows down the execution of the application code in the system.

To optimize container usage for performance enhancement, we should replace inefficient containers without changing the program behavior. We will recap existing efforts on data structure synthesis in Section 2.2.1 and present our solution in Chapter 3.

2.1.2 Store-Load API Pair

Concepts. Libraries offer a variety of APIs for the developers of data-centric systems to implement their application modules, permitting them to concentrate on high-level logic only. Apart from the libraries providing different implementations of containers, typical libraries also include Android platform [24], Java EE framework [31], and various object-relational-mapping frameworks [32]. Among the library APIs with diverse functionalities,

```
// Store data in an Intent
Intent intent = new Intent();
intent.putExtra("name", null);
intent.putExtra("age", 25);
intent.putExtra("isStudent", true);

// Load data from the Intent
String name = intent.getStringExtra("name");
int age = intent.getIntExtra("age", 0);
boolean isStudent = intent.getBooleanExtra("isStudent", false);
```

Figure 2.1: An example program using the class android.content.Intent

there exist one particular class of library APIs that store its parameters to inner fields or load the value of inner fields as return values. As generalized store and load operations, such APIs can form store-load API pairs, which are formulated formally as follows.

Definition 2.1.2 *(Store-Load API Pair) Given a library class* $\mathtt{c}$*, two methods* $\mathtt{m}_1$ *and* $\mathtt{m}_2$ *offered by* $\mathtt{c}$ *form a store-load API pair* $(\mathtt{m}_1, \mathtt{m}_2)$ *if and only if the return value of* $\mathtt{m}_2$ *can be aliased with a parameter of* $\mathtt{m}_1$ *when* $\mathtt{m}_2$ *is invoked after* $\mathtt{m}_1$ *upon the same object of the class* $\mathtt{c}$*.*

Store-load API pairs are prevalent in the libraries written by modern programming languages, such as C++, Java, and Python. To protect the data stored in the inner fields, the classes may set the fields as non-public and only permit the access and modification via specific APIs. When two APIs store and access the same inner field respectively, they actually form the store-load API pairs, which can achieve the similar functionality to the store-load operations with the same pointers.

Example 2.1.2 *Consider the APIs shown in Table 2.1. The methods* put *and* get *of the classes* HashMap *and* CollectionBag *form a store-load API pair. Consider the program shown in Figure 2.1. The class* Intent *in Android platform offers four methods, namely* putExtra*,* getStringExtra*,* getIntExtra*, and* getBooleanExtra*. The method* putExtra *can form three store-load API pairs with the last three methods.*

Relevance to Reliability and Performance of Systems. With a similar motivation to container reasoning, we need to understand the data manipulation conducted by library APIs and discover the store-load API pairs. More specifically, we have to discover the possible aliasing relations between API parameters and return values, which enable us

to identify indirect value flows induced by library API calls in the application code. For example, the null value inserted by putExtra can be aliased the return value of getStringExtra in Figure 2.1, and thus, the variable name can be null. If we conduct any dereference upon name, the program can crash with the null pointer exception. Meanwhile, obtaining the aliasing facts induced by store-load API pairs support better understanding of application code, which has critical importance in the performance enhancement, such as program debloating [33] and data structure optimization [34]. We will summarize existing trials on library specification inference in Section 2.2.3 and demonstrate our API aliasing specification inference algorithm in Chapter 5.

2.1.3 Data Constraint

Concept. Data correctness has critical importance to the reliability of data-centric systems. For example, FinTech systems process sensitive financial data of their users. Any erroneous data manipulation would cause inestimable economic loss. To improve the system reliability, the developers of data-centric systems, especially database-backed systems, often specify data constraints and check them in the runtime. The violation of data constraints would prevent the applications from conducting abnormal data manipulations that trigger the system's alarms with error messages, which can guide the developers to system diagnosis. As important domain-specific programs for data validation, data constraints have been widely studied in recent years [15, 35, 36, 37, 14]. We can provide its formal definition as follows.

Definition 2.1.3 *(Data Constraint) Given a set of database tables $\mathcal{T}$, a data constraint is a first-order logic formula over the attributes of several tables $\mathtt{t} \in \mathcal{T}$, which should be satisfied during the system execution.*

A data constraint can be instantiated in different domain-specific languages. For example, developers can encode the negation of the first-order logic formula as a where clause in a SQL query and check the emptiness of the querying result. Besides, developers can also use several light-weighted languages, such as Google Aviator [38], or design new languages and evaluation engines to achieve high efficiency in the runtime checking.

```
s = 'IN';
if(contains(t.ty,s))
  assert(t.in > 0);
else
  assert(t.out > 0);
assert(t.amt > 0);
assert(t.oid != 0);
```

Figure 2.2: An example of a data constraint

Example 2.1.3 *Figure 2.2 shows an example of data constraints, which is a simplified data constraint in Ant Group. Similar to C/C++/Java programs, the assertions in the data constraints are expected to hold over the database table* t *in the runtime. Specifically, the attribute* ty *of the table* t *is a string attribute. When it contains 'IN' as its substring, the attribute* in *should be greater than 0, and otherwise, the attribute* out *should be greater than 0. Also, the attribute* amt *of the table* t *should be greater than 0, while the attribute* oid *cannot be 0.*

Relevance to System Reliability. Data constraints play a critical role in validating the data correctness in the database, which improves the reliability of data-centric systems. As long as developers formulate the system specification formally with data constraints, the checking engine would protect the data from being incorrectly modified if any data constraint was violated. Thus, the developers of a data-centric system configure the list of data constraints during the development cycle. When the system supports a new functionality, several new data constraints may need to be specified and examined. Any incomplete configuration of data constraints would threaten the system's reliability, degrading the effectiveness of the runtime checking.

Relevance to System Performance. In a very large database, the runtime checking of data constraints demands loading incredibly huge data. Also, a data-centric system can be equipped with thousands of data constraints. The two factors make the runtime checking of data constraints consume fairly large computation resources. Particularly, the response time of the system can be increased due to the checking of data constraints, affecting the system throughput in the deployment environment. To promote the system's performance, the developers leverage various caching strategies and deploy highly efficient machines in industrial scenarios.

Although specifying and examining data constraints has become a common practice in production, data-centric systems still suffer from performance issues caused by redundant

checking of data constraints. According to our experience, the developers often specify the data constraints equivalent to the ones submitted by others. The key reason is that they are unaware of whether there are any existing data constraints equivalent to their new ones. In this case, they prefer to specify as many data constraints as possible aggressively without sacrificing data correctness. Thus, equivalent data constraints can aggregate in the systems and introduce redundant data validation, gradually forming the technical debt of the systems.

To improve the performance without compromising reliability, we need an effective decision procedure to verify the data constraint equivalence. We will summarize previous studies on program equivalence verification in Section 2.2.4 and demonstrate our approach in Chapter 6.

2.2 Related Work

Starting from the four features of data-centric systems demonstrated in Section 1.1, we target four subproblems relating to containers, store-load API pairs, and data constraints:

- (1) *How to identify inefficient container types and synthesize efficient ones?*
- (2) *How to identify value flows through containers in the application code?*
- (3) *How to infer API aliasing specifications for the libraries used by application code?*
- (4) *How to identify equivalent data constraints to avoid redundant data validation?*

The effective solutions to the four problems would promote overall reliability and performance of the systems.

Notice that data-centric systems are often difficult to instrument, deploy, and execute with real-world inputs during the analysis phase. Hence, we choose static analysis as our technical scope to resolve the above problems. Specifically, we conduct a comprehensive study of previous program analysis techniques, especially static analysis, and summarize four relevant lines of literature, namely data structure synthesis, value-flow analysis, library specification inference, and program equivalence verification. Particularly, we will

highlight the research gap and the drawbacks of existing techniques, which further motivates our solutions in the subsequent chapters.

2.2.1 Data Structure Synthesis

There is an extensive body of literature on data structure synthesis, targeting optimizing data organization for performance enhancement. Specifically, existing studies can be divided into two categories. The first category conducts a conservative program transformation, which only replaces original data structures with alternative ones. Most of the works in this category share similar motivations to ours, aiming to reduce computation resource consumption via data structure replacement. For example, ARTEMIS [19] and SEEDS [39] search container types to minimize resource consumption when executing the program with the given test suite. However, they execute the program thousands of times over the test suite to search for the optimal selection, introducing a heavy time burden. BRAINY [40] and CHAMELEON [41] utilize dynamic profiling to obtain the heap information and predict the best container types with a prediction model, which is specified by expertise or obtained in the training process. However, the model can be restrictive when the expert knowledge is unavailable or the training data is not general enough. Different from the dynamic profiling-based approaches, CT+ [42, 43] attempts to reduce resource consumption by analyzing container usage statically, and replace container types based on the class hierarchy diagram. Unfortunately, CT+ can not synthesize more general replacements, such as replacing ArrayList with HashSet, because of the restrictive assumptions on interchangeable container types.

The second line of the works targets composing data structure designs out of existing data structures. Motivated by various scenarios, data structures are supposed to satisfy different constraints. For example, VOLT [7] is the latest data structure synthesizer, which aims to refine the data structure satisfying integrity constraints when introducing auxiliary fields. MASK [8] replaces outdated data structures by synthesizing their methods with the latest ones. The two synthesizers assure functional correctness while do not concern efficiency. Other works, such as COZY [44], RELC [45, 46], and DATA CALCULATOR [47], combine static cost models with operational semantics and synthesize data structures with an excellent performance to alleviate the human burden in designing data

structures. A recent work SOLIDARE refactors data types in real-world smart contracts to save the gas usage [6], which exhibits the closest motivation to ours.

According to the investigation of existing literature, we find that few studies focus on finding more efficient container types in real-world programs without changing program semantics. Although the second line of the works can synthesize more efficient data structures conforming to specific functional specifications, their solutions do not target the client programs using data structures, and instead, try to optimize the program from the data structure side. It should be noted that a variety of container types are provided by third-party libraries, which can be directly used in our program. Therefore, we want to optimize the application code by choosing better container types, and meanwhile, ensure the program behavioral equivalence during the optimization.

2.2.2 Value-Flow Analysis

Value-flow analysis resolves the program dependencies to improve the effectiveness of static analyses [4, 3, 25]. It was initially applied in program optimization and debugging by offering def-use relations. With the promotion of the research communities, it has become a fundamental technique of static program analysis and verification [5, 48, 49]. In our problem, it is actually a powerful technique to investigate the data organization and propagation in the application code.

A critical component of value-flow analysis is pointer analysis, which can greatly affect the precision and efficiency of the overall analysis. In order to achieve better precision with lower overhead, many value-flow analyses adopt on-demand pointer analyses so that unnecessary pointer facts are not calculated in the value-flow propagation [50, 51, 3]. To handle potentially unbounded heap memory, most of the value-flow analyses adopt a bounded abstract memory model [52], which summarizes the memory objects allocated by the same statement into an abstract object [53, 50, 51]. Thus, the number of memory objects is bounded by the program size. To support analyzing structural memory objects, such as arrays and containers, existing value-flow analyses do not conduct precise reasoning of how each element is stored in specific positions or corresponds to keys, and instead, just smash memory regions into the sets of objects [54, 55]. The strategy of memory smashing would introduce spurious program dependencies and conservatively identify

the value flows that cannot occur in actual program executions [54, 55], which can cause high false positives in static bug hunting [25].

In the community of program verification, a wide range of studies propose various memory abstractions for data structure verification. One common abstraction is based on generic predicates, merging the objects which satisfy certain predicates [56, 57, 58, 59, 60]. A typical instantiation is the three-valued logic analyzer (TVLA) [56, 57]. Equipped with a finite number of predicates, it establishes a bounded abstract domain and defines *focus* and *coerce* operators for semantic reduction [61], which can yield precise three-valued structures depicting the memory state. Another line of studies proposes a symbolic memory model and encodes memory states with logic formulas [62, 63]. Specifically, COMPASS depicts a container memory layout with a logical formula in the combined theory of linear integer arithmetic and uninterpreted functions [63]. Existing trials on the data structure verification may benefit the value-flow analysis in precise reasoning of containers, which further supports precise value-flow analysis for the programs using containers.

2.2.3 Library Specification Inference

The inference of library API specifications has always been a central topic in program analysis. Motivated by various downstream clients, existing specification inference techniques can target different forms of library specifications, including points-to [22], aliasing [23], taint [64], and commutativity specifications [65]. From the perspective of technical design, previous studies can be categorized into two major categories. The first line of studies analyzes the library implementations to abstract the semantics of library APIs. Typically, IFDS/IDE-based approaches summarize the data-flow facts of libraries using summary edges in exploded super graphs as their semantic abstractions [66, 67], which can be reused across various clients of data-flow analysis. Established upon a symbolic memory model, shape analysis computes the memory state for each statement of a library API as invariants, and derives the pre/post conditions of each library API as its specification [56, 57, 68]. While the inferred specification accurately depicts the semantics of the library API, the analysis suffers from scalability problems, especially in the presence of complex program structures [69].

The other line of existing techniques is mining-based approaches [22, 23, 64, 65]. Specif-

ically, they deduce the library API specifications from the program facts of the code using libraries, which can be obtained by static or dynamic analysis. For instance, ATLAS [22] enumerates the unit tests using library APIs via active learning and further infers the points-to specifications by examining whether the assertions hold or not. USPEC [23] deduces the potential aliasing facts induced by libraries from object manipulation traces, which are collected via static analysis techniques. Unlike the first line of studies, mining-based approaches do not rely on library implementations, which promotes their applicability when the source code is not available. However, the program facts of code using libraries may not be easily obtained in several scenarios. For example, the big code using libraries may not be accessible, and the unit tests of library APIs may not be easy to construct and execute.

In this thesis, we concentrate on the aliasing facts induced by library APIs and attempt to derive their aliasing specifications. To make our solution more applicable to data-centric systems, we need to tackle the drawbacks of existing studies and propose a new perspective to infer the specifications. We notice that the developers of data-centric systems often refer to library documentation, which offers critical knowledge of library APIs and effectively guides the development process. Compared with unit tests and other applications using libraries, library documentation is more likely to be available for analysis. Hence, we want to infer library API aliasing specifications for store-load API pairs from library documentation.

2.2.4 Program Equivalence Verification

Program equivalence verification is a crucial building block for many clients, such as translation validation [70, 71], program synthesis [72, 8, 73], and program optimization [74, 75]. To resolve redundant data validation on the database side, we prepare to leverage the equivalence verification technique to identify equivalent data constraints.

According to our investigation, there are two lines of studies focusing on program equivalence verification. One line of studies reduces equivalence checking to proving specific verification conditions, such as *relational verification* [76, 77, 78, 79, 80, 81]. Similar approaches include using symbolic execution for loop-free programs [82, 83, 8, 84, 85]. Another line of studies proves program equivalence via *term rewriting* [70, 86, 87, 88].

The effectiveness relies heavily on the quality of rewrite rules. First, they may sacrifice soundness or completeness if the rule set contains an incorrect rule or misses the right one [89]. Second, they may suffer from the phase ordering problem [90] in the presence of many rewrite rules. To obtain better complexity, [91] restricts the form of rewrite rules, and adopts tree isomorphism algorithms to check syntactic isomorphism, which ensures that the equivalence relation can be proved in polynomial time.

There has also been a wide range of literature concentrating on the equivalence verification of domain-specific programs. One typical line is SQL query equivalence verification, which is an essential topic in academia and industrial communities. The state-of-the-art approaches focus on specific forms of SQL queries [92] and apply either algebraic reasoning techniques [87, 93, 88] or symbolic reasoning [94, 75, 95] for equivalence verification. Typically, UDP [88] utilizes U-semiring to encode the bag semantics of SQL effectively and checks the isomorphism between two algebraic structures. However, it fails to handle advanced features, e.g., three-valued logic, and suffers from the inefficient chase procedure in the isomorphism checking [96]. In contrast, EQUITAS [75] encodes the semantics with a FOL formula and leverages a solver to determine the equivalence, handling more SQL features [97] than UDP.

Overall, existing efforts mainly explore the verification of the programs with more expressive program constructs, such as loops in imperative programs and three-valued logic in SQL queries, or try to verify large-scale programs with better scalability. However, in data-centric systems, we have to determine the equivalence relations upon thousands of data constraints. To resolve equivalence data constraints, we need to design a new decision procedure to achieve high efficiency in analyzing a large number of data constraints.

CHAPTER 3

COMPLEXITY-GUIDED CONTAINER REPLACEMENT SYNTHESIS

3.1 Introduction

General-purposed programming languages, including Java and C++, support a variety of containers, which creates great convenience of developing software systems with complex service logics, such as a variety of data-centric systems. Unfortunately, performance issues often emerge because of inefficient usage of container types. Programmers are often unaware of more efficient container types under their development context and tend to choose the container types that they are most familiar with. For example, in the program shown in Figure 3.1, the use of the container type ArrayList introduces unnecessary time overhead because the method, ArrayList.contains, performs linear searching. The same functionality can be supported efficiently by the class HashSet. It is quite surprising to find that 16% of execution time of the 3D design software, Raytrace, is introduced by inefficient container types [40], affecting the performance of ray tracing greatly. Moreover, there is abundant evidence that inefficient containers also increase other resource consumption, including memory [40, 19], energy [39, 98, 43], and CPU usage [19].

Goal and Challenge. Given a set of container types, our goal is to synthesize alternative container types and the associated methods at the container allocation sites and the container method call sites, respectively, such that the program after replacements preserves the original semantics and executes more efficiently for large inputs. In this way, our synthesis algorithm can optimize data organization to accelerate the execution of a given program. We also expect our synthesis algorithm to be general enough, supporting the program optimization to decrease other kinds of resource consumptions, such as the memory and CPU usage.

However, it is far from trivial to achieve the goal. First, we can not determine the container types to replace the original ones without violating the behavioral equivalence [99]

if we do not know how container objects are manipulated. Although several container types are interchangeable, e.g., ArrayList and LinkedList, the replacement patterns might be quite restrictive. Second, it is far from practical to derive a tight bound of the time complexity for a general container-manipulating program [100, 101, 102], especially data-centric applications in the wild, so we can not explicitly compare the complexity of the program before and after replacements to guide the synthesis.

Existing Effort. The existing works attempt to tackle the problem from two perspectives. One line of the existing approaches attempts to find optimal container usage by minimizing the resource consumption upon a given test suite [39, 19]. They mutate container types and evaluate the resource consumption via dynamic profiling until the minimal consumption is reached. The other line of the works selects better container types by performing a prediction task [41, 40, 103]. Based on the heap information and container usage patterns in a specific execution, they predict optimal container types by utilizing a prediction model, which is specified manually or obtained in a training process. Unfortunately, the existing works suffer from three drawbacks:

- *Huge time overhead*. They rely on the execution of the test suite, making the whole process quite time consuming [40, 39]. Particularly, the first line of the works executes the test suite iteratively to find the optimal selection and suffer the huge time overhead. For example, [19] takes 3.1 hours optimizing a project on average, which poses an enormous obstacle to large-scale adoption.

- *Unsoundness*. They can not guarantee the semantic equivalence of the program, as they can not discover how each container object is manipulated and determine the equivalent container types soundly. Although several approaches assume several container types are interchangeable [40, 19], the assumptions do not hold in certain cases, such as transforming LinkedHashMap to HashMap in the presence of map traversal.

- *Overfitting*. The effectiveness of the optimization can be degraded when the test suite or the training data does not provide general inputs. The program after replacements can execute slower when the inputs exercise the program along previously uncovered paths [104].

```
public List load(String[] a, int n) {
  List<String> u = new ArrayList<>();
  for (int i = 0; i < n; i++)
    if (!u.contains(a[i]))
      u.add(a[i]);
  return u;
}
```

```
public void check(String[] a, String s) {
  List v = load(a, a.length);
  if (v.contains(s)) {
    return true;
  }
  return false;
}
```

Figure 3.1: An efficient usage of ArrayList in the project IoTDB

Insight and Solution. We observe that container method calls reveal the intention of the programmers for which they use the containers. Specifically, programmers concern with specific *container properties*, such as the size, index or value-ownership, and index-value correlation. Container methods allow programmers to manipulate a container object by querying and modifying container properties. Our insight is that we can optimize a container-manipulating program if the intention can be achieved by other container types and methods with lower time complexity. In Figure 3.1, for example, the ArrayList object allocated in the method load is only manipulated by the methods ArrayList.add and ArrayList.contains. The programmers only wish to know whether an element is stored in the list, i.e., the value-ownership property of the ArrayList object. Thus, we can replace ArrayList with HashSet to avoid linear searching caused by ArrayList.contains, thereby improving program efficiency.

Based on the insight, we present CRES, a container replacement synthesizer to improve program efficiency. CRES synthesizes container replacements preserving program behavior and achieves the optimization for general inputs.

- To assure the behavioral equivalence, we propose the notion of *container behavioral equivalence* to determine the method candidates. Specifically, CRES analyzes container method calls to determine the concerned container properties. A method is a candidate of a container method call if it queries and modifies the concerned container properties in the same way as the original one.

- To achieve the optimization, we introduce the concept of *container complexity superiority* to constrain the complexity of container methods in the replacements. Specifically, CRES selects methods with low complexity from candidates so that the total complexity of the container method calls manipulating the object is lower than the one in the original program.

With the benefit of our insight, CRES can find the opportunity of achieving input-agnostic optimization and improve program efficiency significantly. To the best of our knowledge, CRES is the first work to guarantee the behavior equivalence without any assumption on interchangeable container types. Moreover, CRES escapes from the burden of huge overhead because it does rely on program execution and performs efficient static reasoning.

We evaluate CRES upon 12 real-world data-centric applications with intensive usage of containers in Java Collections Framework (JCF), of which the sizes range from 18.6 KLoC to 384.2 KLoC. CRES synthesizes 107 replacements covering six categories [105], such as replacing ArrayList with HashSet, replacing TreeMap with HashMap, etc. Particularly, 71 replacements in six projects have been confirmed by the developers. The time consumption of each project is decreased by 8.1% on average after replacements. Moreover, CRES finishes analyzing any project in 14 minutes, which distinguishes it from existing approaches suffering from the heavy time burden [39, 19]. We also prove its soundness theoretically to guarantee the behavioral equivalence. CRES has been integrated into the static analysis platform in the Ant Group, an international IT company providing the financial service for over 1 billion global users. In summary, we make the following main contributions:

- We propose a novel abstraction of containers and introduce two principled notions, namely *container behavioral equivalence* and *container complexity superiority*, to guide the synthesis.
- We establish an abstract domain and propose the *container property analysis* to guarantee the behavioral equivalence of the programs.
- We implement a synthesis framework CRES and evaluate it on data-centric applications, showing that it synthesizes replacements efficiently and significantly improves program efficiency.

The sections of this chapter are organized as follows. We first demonstrate the key idea of our synthesizer CRES with a motivating example in Section 3.2 and then provide a formal statement of our problem in Section 3.3. After illustrating the abstraction for container-manipulating program in Section 3.4, we detail the synthesis algorithm CRES to

```
1  public List getResources(String dir) {
2    List r = new ArrayList<File>(); //o2
3    for (int i = 0; i < RES_NUM; i++) {
4      File s = getFile(dir, i);
5      if (!r.contains(s))
6        r.add(s);
7    }
8    return r;
9  }
10 public List getPrivate(String dir) {
11   List p = new ArrayList<File>(); //o11
12   for (int i = 0; i < N_PRIVATE; i++)
13     p.add(getPrivateFile(dir, i));
14   return p;
15 }
16 public List getProtected(String dir) {
17   List q = new ArrayList<File>(); //o17
18   for (int i = 0; i < N_PROTECTED; i++)
19     q.add(getProtectedFile(dir, i));
20   return q;
21 }
22 public boolean invisible(ArrayList l) {
23   return l.contains(INVISIBLE_FILE);
24 }
25 public List getAllFiles(String dir) {
26   List f = new ArrayList<File>(); //o26
27   for (int i = 0; i < N_FILE; i++)
28     f.add(getFile(dir, i));
29   return f;
30 }
31 public void access(String dir, int token) {
32   List f = getAllFiles(dir);
33   List r = getResources("/OOPSLA");
34   List p = getPrivate("/Data");
35   List q = getProtected("/Data");
36   for (File file : f) {
37     if (!invisible(p)
38             && p.contains(file))
38       continue;
39     if (!invisible(q)
40             && q.contains(file)
41             && q.indexOf(file) > token)
42       continue;
43     if (r.contains(file))
44       System.out.println("Access");
45   }
46 }
```

Figure 3.2: A program accessing the available and visible files in a specific directory

generate container replacement for program efficiency improvement in Section 3.5. Section 3.6 and Section 3.7 show the details of the implementation and evaluation results. Finally, we summarize CRES in Section 3.8.

3.2 Cres in a Nutshell

In this section, we present a motivating example to state the importance of replacing inefficient containers for program efficiency improvement (Section 3.2.1), and illustrate the key idea of our approach to solving the problem of complexity-guided container replacement synthesis (Section 3.2.2).

3.2.1 Motivating Example

Figure 3.2 presents the example extracted and simplified from the projects iotdb and google-http-java-client, which are two popular open-source data-centric applications. The container objects o_2, o_{11}, o_{17}, and o_{26} are allocated by the allocation statements at lines 2, 11, 17, and 26, respectively. The methods getAllFiles and getResources collect the files in the directories named dir and OOPSLA, and store them in o_{26} and o_2, respectively.

The methods getPrivate and getProtected collect the files demanding two different access privileges and store them in two ArrayList objects o_{11} and o_{17}, respectively. We can obtain three observations as follows.

- The ArrayList object o_2 is manipulated by ArrayList.add and ArrayList.contains, so the programmers only wish to know whether an element is stored in o_2, i.e., the value-ownership of o_2. Notice that a HashSet object also supports the value-ownership checking and returns the same result. Besides, the method HashSet.contains works with amortized constant time. Therefore, the program will be more efficient if we replace ArrayList with HashSet at line 2.

- The ArrayList object o_{26} is manipulated by the insertions and traversal. The methods of LinkedList also support the same functionalities. Besides, the method LinkedList.add runs in constant time complexity while the method ArrayList.add works with amortized constant time because of memory reallocation. Thus, we can replace ArrayList with LinkedList to reduce time consumption.

- The ArrayList object o_{11} is created for the value-ownership checking. However, o_{11} and o_{17} are provided as the parameters of the method invisible. The programmers are concerned about the index-value correlation of o_{17} in the invocation of the method ArrayList.indexOf. If we replace the type of o_{11} with HashSet, the code cleanliness can be degraded, as the method invisible should be inlined at two call sites. Thus, we only leverage LinkedList to avoid memory reallocation.

The replacements can bring a significant improvement in program efficiency. For example, the total execution time of the corresponding test cases in google-http-java-client can be reduced by 27.1% if we replace an ArrayList object with a LinkedList object. A large body of literature also reveals that inefficient container types can introduce unnecessary time consumption and even increase time complexity [41, 40, 19]. Thus, it is meaningful to synthesize container replacements to improve program efficiency.

3.2.2 Synthesizing Replacement

The synthesized container replacements should preserve the program behavior and achieve the optimization for large inputs. Unfortunately, it is non-trivial to find the replacements satisfying the two constraints. First, we can not determine which container types can guarantee the behavioral equivalence after replacements if we do not know how container objects are manipulated. Even if several container types are interchangeable in any usage context, such as LinkedList and ArrayList, more general replacements, such as transforming the type of o_2 to HashSet, can not be discovered [41, 19, 43]. Second, it is far from practical to derive a tight bound of the time complexity for a real-world program [100, 101, 102]. We need an effective and computable measure to guide the synthesis such that the synthesis can achieve the input-agnostic optimization for large inputs.

The key idea of our approach comes from the observation about the intention of container usage. We realize that the purpose of programmers is to utilize specific facts about containers, which we call *container properties*. Programmers can query and update the container properties by invoking container methods. In Figure 3.2, for example, the programmers are concerned about the value-ownership of o_2, i.e., the fact that whether an object is stored in o_2. The methods ArrayList.contains and ArrayList.add query and update the value-ownership, respectively. When the concerned properties can be updated and queried by more efficient methods in the same way, we can replace the types and methods to improve program efficiency. Specifically, we propose two concepts to address the challenges:

- We introduce *container behavioral equivalence* to determine the methods that query and update the concerned properties in the same way as the original ones. For instance, only the value-ownership of o_2 is concerned in Figure 3.2, and the methods of HashSet guarantee the container behavioral equivalence. Thus, replacing it with a HashSet object preserves the behavioral equivalence.

- We propose *container complexity superiority* to measure whether the methods manipulating a container object are more efficient after replacements. In Figure 3.2, HashSet.contains has much lower time complexity than ArrayList.contains, and HashSet.add and ArrayList.add do not have significant difference in complexity. After

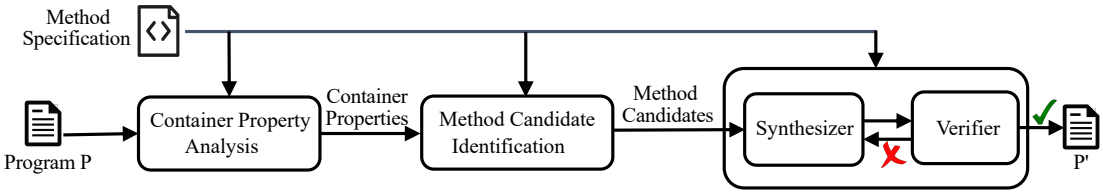


Figure 3.3: Schematic overview of our approach

transforming the type from ArrayList to HashSet, the new program has container complexity superiority.

Based on our insight, we can improve the program efficiency for general inputs if the replacements guarantee the container behavioral equivalence and the container complexity superiority simultaneously. Figure 3.3 shows the workflow of our approach, which consists of three stages.

- In the first stage, the container property analysis identifies which container properties are queried and how they are updated upon each container object in the program. For example, it can discover that only the value-ownership is queried upon o_2 and o_{11} in Figure 3.2.
- In the second stage, the methods are identified as the method candidates if they the preserve container behavioral equivalence. For instance, the methods HashSet.contains and HashSet.add are the candidates of the container method calls at line 5, 6, and 43, as they query and update the value-ownership in the same way as the methods ArrayList.contains and ArrayList.add.
- In the third stage, we instantiate a CEGIS paradigm [106, 107, 108] to synthesize container types and methods. A synthesizer selects efficient method candidates and resolves the counterexamples in the consequent rounds if type checking fails in the verification. For instance, the actual parameters of the method invisible are inconsistent at lines 37 and 39 if we replace o_{11} with a HashSet object, so the synthesizer refines the type of o_{11} by selecting another type LinkedList in a consequent round.

Specifically, the generation and selection of method candidates both rely on sound reasoning about the queried container properties and how they are updated in the program.

Technically, we establish an abstract domain to abstract the container-property queries in the program, which guide the generation of method candidates and further guarantee the container behavioral equivalence.

With the benefits of our insight, our approach stands out due to the following three perspectives.

- *The low overhead introduced by the synthesis.* The synthesis algorithm does not rely on any input and execution of the program, and static reasoning of container properties is sufficient to identify candidates with quite low overhead.
- *Sound and various replacements.* The algorithm analyzes the concerned container properties to guide method candidate identification, which not only guarantees the behavioral equivalence but also discovers the replacements uncovered by existing approaches, such as replacing ArrayList with HashSet, and replacing LinkedHashMap with HashMap.
- *Input-agnostic optimization.* The algorithm utilizes the complexity specification of container methods to guide the synthesis so that the replacements are insensitive to the program inputs, and the time complexity of the program is more likely to be decreased.

3.3 Problem Formulation

In this section, we first present the language used in this paper and its concrete state (Section 3.3.1). We then define the behavioral equivalence (Section 3.3.2) to constrain the program behavior after container replacements. Finally, we formalize the problem of complexity-guided container replacement synthesis (Section 3.3.3).

3.3.1 Program Syntax and Concrete State

Let $\mathcal{C}$ and $\mathcal{M}$ denote the family of the container types and their methods, respectively. Also, we let method denote the function mapping a container type τ to the set of container methods supported by τ. Figure 3.4 shows the syntax of the language. The expressions

$$
\begin{aligned}
&\textit{Program}\ \ P := F+ \\
&\textit{Container method}\ \ F_C := f_{\mathcal{C}}(v_1, \ldots, v_m) \\
&\textit{Function}\ \ F := f(v_1, \ldots, v_n)\{S;\ \ \textbf{return}\ e\} \\
&\textit{Statement}\ \ S := v = \textbf{new}\ \ \tau \mid v = e \mid S_1; S_2 \mid \textbf{return}\ e \\
&\qquad\qquad \mid \textbf{if}\ (e)\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \mid \textbf{while}\ \ (e)\ \ \textbf{do}\ \ S\ \ \textbf{od} \\
&\qquad\qquad \mid v = c.f_{\mathcal{C}}(v_1, \ldots, v_m) \mid v = f(v_1, \ldots, v_n) \\
&\textit{Expression}\ \ e := a \mid v \mid u_1 \oplus u_2 \mid \otimes u \\
&\textit{Variable}\ \ v := c \mid u \\
&\textit{Operator}\ \ \oplus := \wedge \mid \vee \mid + \mid - \mid = \mid \cdots \quad \otimes := \neg \mid - \mid \cdots
\end{aligned}
$$

Figure 3.4: The syntax of the language.

include literals, variable expressions, and unary/binary expressions. A statement can be an allocation statement, an assignment, a sequencing, a branch, a loop, a function call, or a return statement. Particularly, a function call is either an invocation of a user-defined function f or a container method $f_{\mathcal{C}}$ with the receiver container c. A program has a unique function as its entry, which has a unique return statement.

We denote the sets of program variables and values by Var and $\mathsf{Val} := \mathsf{Addr} \cup \mathsf{OVal}$, respectively. Specifically, Addr is a set of addresses of objects, OVal is a set of non-address values, and $\mathsf{Idx} \subseteq \mathsf{Val}$ includes the index values of the containers. Formally, we can define the **concrete state** as follows.

Definition 3.3.1 (Concrete State) *A concrete state* $s \in \mathit{State}$ *is a 3-tuple* $(\varepsilon, \mu, \beta)$*, where*

- *An* environment $\varepsilon \in \mathit{Env} := \mathit{Var} \to \mathit{Val}$ *maps a set of variables* Var *to a set of values* Val*.*
- *A* memory $\mu \in \mathit{Mem} := (\mathit{Addr}, \mathit{Idx}) \to \mathit{Val}$ *maps a pair of an address and an index to a value, which is the value stored at the index of a container object.*
- *A* base $\beta \in \mathit{Base} := \mathit{Addr} \to \mathit{U}$*, where* $\mathit{U} := \{((A, \preceq_1), \ldots, (A, \preceq_k)) \mid A \subseteq \mathit{Idx}\}$*, maps an address to a* k*-tuple, of which the entry is a partially ordered set. Each partial order* $\preceq_i$ *determines a specific order of the indexes of the container object stored at the address.*

The concrete state supports the semantics of container methods with various features. In JCF, for instance, TreeMap supports accessing the value associated to the largest key,

```
public boolean foo1(String a) {
  List<String> l = new ArrayList<>();
  l.put("PL");  l.put("SE");
  boolean b1 = l.contains(a);
  return b1;
}
```

```
public boolean foo2(String a) {
  Set<String> s = new HashSet<>();
  s.add("PL");  s.add("SE");
  boolean b2 = s.contains(a);
  return b2;
}
```

Figure 3.5: Two behaviorally equivalent programs

and LinkedHashMap supports iterating according to the insertion order. These advanced features can be expressed by specific partial orders in the base.

Example 3.3.1 *Consider the function foo1 in Figure 3.5. At the exit of the function, we have* $\varepsilon(l) = a_l$, $\mu(a_l, 0) =$ "PL" *and* $\mu(a_l, 1) =$ "SE", *where* a_l *is the address where the ArrayList object is allocated. Particularly, we enforce* $\beta(a_l)$ *equal to* $\varnothing$, *as its semantics does not rely on any order of the indexes.*

3.3.2 Behavioral Equivalence

The program after the replacements should preserve the semantic equivalence. Based on the concrete state, we define the **behavioral equivalence** [99] for two programs to constrain the input-output relationship, which is a specific form of program behavior.

Definition 3.3.2 (Behavioral Equivalence) P *is behaviorally equivalent to* P', *denoted by* $P \simeq P'$, *if and only if for any input, the expressions* e *and* e' *in the return statements of the entry functions evaluate to the same value, i.e.,* $[\![e]\!](s) = [\![e']\!](s')$. $[\![e]\!]$ *is the function mapping a concrete state to the value of the expression.* s *and* s' *are the concrete states of* P *and* P' *at the exits, respectively.*

Example 3.3.2 *Suppose that* $s_1 = (\varepsilon_1, \mu_1, \beta_1)$ *and* $s_2 = (\varepsilon_2, \mu_2, \beta_2)$ *are the concrete states at the exits of the functions foo1 and foo2 in Figure 3.5, respectively. We have* $\varepsilon_1(b1) = T$ *and* $\varepsilon_2(b2) = T$ *iff* a *is equal to "PL" or "SE", i.e.,* $[\![b1]\!](s_1) = [\![b2]\!](s_2)$, *indicating that they are behaviorally equivalent.*

Behavioral equivalence defines an equivalence relation between two programs based on the input-output relationship, which is a program behavior concerned in many scenar-

ios [99]. The program after replacements should be behaviorally equivalent to the original program, preserving the semantics we are concern about.

3.3.3 Problem Statement

To synthesize container replacements, we identify the allocation statements of container objects and container method calls as the skeleton of synthesis. Given a program $\mathtt{P}$, we denote the set of the two kinds of statements by $\mathcal{S}$. In what follows, we let $\mathcal{S}_a \subseteq \mathcal{S}$ and $\mathcal{S}_c \subseteq \mathcal{S}$ denote the sets of container allocation statements and container method calls, respectively. We state the problem of complexity-guided container replacement synthesis as follows.

Definition 3.3.3 (Complexity-Guided Container Replacement) *Given a program* $\mathtt{P}$*, synthesize the replacement mappings* $\psi_c : \mathcal{S}_c \to \mathcal{M}$ *and* $\psi_a : \mathcal{S}_a \to \mathcal{C}$*. For* $st_c \in \mathcal{S}_c$ *and* $st_a \in \mathcal{S}_a$*, we replace the container method* $f_{\mathcal{C}}$ *invoked by* st_c *with* $\psi_c(st_c)$*, and replace the container type* τ *used in* st_a *with* $\psi_a(st_a)$*, which should satisfy: (1) Behavioral equivalence:* $\mathtt{P}'$ *and* $\mathtt{P}$ *are behavioral equivalent; (2) Complexity superiority:* $\mathtt{P}'$ *consumes no more time than* $\mathtt{P}$ *for any large input.*

Intuitively, the behavioral equivalence and the complexity superiority formulate our expectations on the new program after replacements. To solve the problem, we establish the abstraction for containers in Section 3.4 and design the synthesis algorithm to synthesize the container replacement mappings in Section 3.5.

Remark. We only concentrate on the statement-wise replacements synthesis in our problem. A major advantage of performing such a form of replacements is that the program structure is not affected by the replacements. If we conduct more aggressive changes to the code, e.g., defining two functions to replace the invocation of the function invisible at lines 37 and 39 in Figure 3.2, the program after replacements can have a big difference from the original one, which degrades the code cleanliness and increases the difficulty of the maintenance.

3.4 Program Abstraction

In this section, we first introduce the notion of the container-property query and establish the abstract states (Section 3.4.1). We then propose the method semantic specification and define the concept of the container behavioral equivalence to guarantee the behavioral equivalence (Section 3.4.2). Finally, we define the notion of the container complexity superiority as the heuristic guidance to synthesize replacements satisfying the complexity superiority (Section 3.4.3).

3.4.1 Container Property Abstraction

As explained in Section 3.2.2, we can represent the intention of utilizing containers by the concerned container properties, which are specific forms of facts about containers. To show the intention of container usage, we define the container-property query and establish an abstract domain to abstract the concerned container properties.

Container-Property Query. We introduce the concept of the *container property* to indicate the intention of container usage. A container property is essentially a numeric quantity or a predicate upon the indexes and the values of a container object. Intuitively, it is a specific form of facts about a container object. Table 3.1 shows the typical container properties of commonly-used container types in JCF, which depict the following facts.

- size shows the size of a container object, i.e., the number of the values in the container.
- $\mathsf{isIdx}(\lambda)$ and $\mathsf{isVal}(\nu)$ indicate the index-ownership and value-ownership, respectively. λ or ν is an index or a value of the container iff $\mathsf{isIdx}(\lambda) = \mathsf{T}$ or $\mathsf{isVal}(\nu) = \mathsf{T}$.
- $\mathsf{isCor}(\lambda, \nu)$ indicates the index-value correlation. The index λ is paired with the value ν iff $\mathsf{isCor}(\lambda, \nu) = \mathsf{T}$.
- $\mathsf{InsOrd}(\lambda_1, \lambda_2)$ indicates the insertion order of indexes. λ_1 is inserted before λ_2 iff $\mathsf{InsOrd}(\lambda_1, \lambda_2) = \mathsf{T}$.
- $\mathsf{KeyOrd}(\lambda_1, \lambda_2)$ indicates the order of keys. λ_1 is larger than λ_2 iff $\mathsf{KeyOrd}(\lambda_1, \lambda_2) = \mathsf{T}$.

Table 3.1: Examples of container properties

	size	isIdx(λ)	isVal(v)	isCor(λ, v)	InsOrd(λ_1, λ_2)	KeyOrd(λ_1, λ_2)
ArrayList	✓		✓	✓		
LinkedList	✓		✓	✓		
HashSet	✓	✓	✓			
TreeSet	✓	✓	✓			✓
LinkedHashSet	✓	✓	✓		✓	
HashMap	✓	✓	✓	✓		
TreeMap	✓	✓	✓	✓		✓
LinkedHashMap	✓	✓	✓	✓	✓	

Let Property denote the family of the container properties. Based on the concept of the container property, we define the **container-property query** to formalize which container property is utilized by a container method call.

Definition 3.4.1 (Container-Property Query) *A container-property query is a function* q *mapping a pair of a concrete state and a container variable to a container property* p*, i.e.,*

$$\mathrm{q} : \mathit{State} \times \mathit{Var} \to \mathit{Property}$$

$$(\mathrm{s}, \mathrm{c}) \mapsto \mathrm{p}$$

Furthermore, we can construct a family of container-property queries $\mathcal{Q}$ to represent all the possible container-property queries induced by container methods.

Example 3.4.1 *Consider the container type* LinkedHashMap *as an example. The methods* LinkedHashMap.containsKey *and* LinkedHashMap.containsValue *induce the queries of the container properties* isIdx(λ) *and* isVal(v)*, respectively. Besides, the method* LinkedHashMap.get *queries the container property* isCor(λ, v)*. Its iterator also queries the container property* InsOrd(λ_1, λ_2)*, as its semantics relies on the insertion order.*

Abstract State. Based on container-property queries, we can establish an abstraction of concrete states in Section 3.3.1. To assure the boundedness of the abstract domain, we adopt the allocation site-based memory abstraction [52], and introduce an abstract object to summarize the memory objects allocated by the same allocation statement. Formally, we define the **abstract state** for container-manipulating programs as follows.

Definition 3.4.2 (Abstract State) *$\mathcal{V}_c$ is the set of container variables and $\mathcal{O}_c$ is the set of abstract container objects. An abstract state is $\widetilde{s} = (\widetilde{\varepsilon}, \widetilde{\rho})$, where*

- *$\widetilde{\varepsilon} : \mathcal{V}_c \to 2^{\mathcal{O}_c}$ indicates the points-to fact of container variables. For each container variable $c \in \mathcal{V}_c$, $\widetilde{\varepsilon}(c)$ is the set of abstract container objects which c may point to.*
- *$\widetilde{\rho} : \mathcal{O}_c \to 2^{\mathcal{Q}}$ indicates the property-query fact of container objects. For each container object $o \in \mathcal{O}_c$, $\widetilde{\rho}(o)$ contains the container-property queries occurring upon the object o.*

Example 3.4.2 *Consider the function foo1 in Figure 3.5. We have $\mathcal{O}_c = \{o_2\}$, where o_2 is the ArrayList object allocated at line 2. The container object is only created for the value-ownership checking. Before the return statement, the abstract state is $(\widetilde{\varepsilon}, \widetilde{\rho})$, where $\widetilde{\varepsilon}(l) = \{o_2\}$, $\widetilde{\rho}(o_2) = \{q\}$, and $q(s, c)$ is isVal(v), indicating that only the value-ownership of o_2 is concerned in the program.*

Intuitively, an abstract state abstracts away the facts irrelevant to container variables and objects. Based on the abstract state, we can determine how a container object is utilized in the program by identifying (1) which container objects are manipulated and (2) which container properties are concerned. The abstract state provides sufficient information about the intention of container usage and enables us to examine the behavioral equivalence.

3.4.2 Behavior Constraint

Based on our insight, the behavioral equivalence must hold if the container properties are queried and updated in the same way as the original program. To formulate the criteria explicitly, we first introduce the notion of the container-property modifier and provide a novel representation of method semantic specification. We then propose the container behavioral equivalence to specify the constraints that guarantee the behavioral equivalence.

Method Semantic Specification

To maintain the container content for further queries in the program, each container method updates the memory μ and base β in the concrete state and modifies the container properties. To depict the effect of a container method, we define the **container-property modifier** formally as follows.

Definition 3.4.3 (Container-Property Modifier) *A container-property modifier is a function* t *mapping a 4-tuple, which consists of a container variable, a tuple of parameter variables, a concrete state, and a container property, to a container property, i.e.,*

$$t : \mathit{Var} \times \mathit{Var}^* \times \mathit{State} \times \mathit{Property} \to \mathit{Property}$$

$$(c, args, s, p) \mapsto p'$$

where p *and* p' *indicate the container properties before and after applying the modifier, respectively.*

Similar to container-property queries, we can construct a family of container-property modifiers $\mathcal{T}$ to enumerate all the possible effects of container methods.

Example 3.4.3 *Suppose that the language only supports the usage of* LinkedHashMap*. A container-property modifier can be one of the following forms: (1) Increasing or decreasing the size by at most one; (2) Inserting or removing an index or a value; (3) Inserting or removing a pair; (4) Inserting or removing an element from the partially ordered set, which preserves the insertion order.*

Notice that a container-property modifier can affect several container properties. To show the effect explicitly, we establish a function $\omega_{\mathcal{T}} : \mathcal{T} \to 2^{\mathcal{Q}}$ mapping a container-property modifier to the set of the container-property queries affected by it. For instance, when t inserts or removes an element in the i-th partially ordered set of the base, it can affect the query of i-th partial order.

Given a container method, its semantics is essentially a combination of two orthogonal parts, namely specific container-property queries and container-property modifiers. Formally, we can define the **method semantic specification** as follows.

Definition 3.4.4 (Method Semantic Specification) *The method semantic specification is a function* $\alpha_{\mathcal{M}} : \mathcal{M} \to 2^{\mathcal{Q}} \times 2^{\mathcal{T}}$*. For a given* $f_{\mathcal{C}} \in \mathcal{M}$*,* $(Q, T) := \alpha_{\mathcal{M}}(f_{\mathcal{C}})$ *indicates the container-property queries and the container-property modifiers induced by* $f_{\mathcal{C}}$*, respectively.*

Example 3.4.4 *The method semantic specification maps* LinkedHashMap.get *to* $(\{q\}, \varnothing)$*, where* $q(s, c) =$ *isCor*(λ, v)*. Similarly,* LinkedHashMap.put *is mapped to* $(\varnothing, \{t_1, t_2, t_3, t_4, t_5\})$*, where* t_1 *increases the size by at most one, and* t_i $(2 \leqslant i \leqslant 5)$ *insert an element or a pair to update* *isIdx*(λ)*,* *isVal*(v)*,* *isCor*(λ, v)*, and* *InsOrd*(λ_1, λ_2)*, respectively.*

The method semantic specification describes the semantics of container methods in a compact way, blurring the details of how the memory is updated and container property is computed. Using the abstraction, we propose the container property analysis in Section 3.5.1 to compute the abstract states, which maintain the concerned container properties of each container object. The properties provide sufficient guidance to guarantee the behavioral equivalence in the container replacements.

Container Behavioral Equivalence

To guarantee the behavioral equivalence in the container replacements, the container methods in the new program P' should query and modify the concerned container properties in the same way as the ones in the program P. Besides, we need to constrain the types of container objects manipulating by the same container method call, which should be equal to assure that P' is well-typed. To provide the criteria of the behavioral equivalence for our problem explicitly, we define the **container behavioral equivalence** formally as follows.

Definition 3.4.5 (Container Behavioral Equivalence) *Given two programs* P *and* P'*, where* P' *is obtained by applying* ψ_a *and* ψ_c *to* P *to perform container replacements.* P *and* P' *have the container behavioral equivalence relation, denoted by* $P \simeq_{\mathcal{C}} P'$*, if and only if for any* $st_c \in \mathcal{S}_c$ *in the form of* $v = c.f_{\mathcal{C}}(v_1, \ldots, v_m)$*,* ψ_a *and* ψ_c *satisfy*

$$Q = Q' \tag{3.1}$$

$$\forall o \in \widetilde{\varepsilon}_{st_c}(c)\ \forall q \in \widetilde{\rho}_e(o),\ \eta(T, q) = \eta(T', q) \tag{3.2}$$

$$\forall o_1 \in \widetilde{\varepsilon}_{st_c}(c)\ \forall o_2 \in \widetilde{\varepsilon}_{st_c}(c),\ \mathit{alloc}(o_1, st_{a1}) \wedge \mathit{alloc}(o_2, st_{a2}) \rightarrow \psi_a(st_{a1}) = \psi_a(st_{a2}) \tag{3.3}$$

where $(Q, T) := \alpha_{\mathcal{M}}(f_{\mathcal{C}})$ *and* $(Q', T') := \alpha_{\mathcal{M}}(\psi_c(st_c))$*.* $(\widetilde{\rho}_e, \widetilde{\varepsilon}_e)$ *and* $(\widetilde{\rho}_{st_c}, \widetilde{\varepsilon}_{st_c})$ *are the program states at the exit of* P *and before* st_c*, respectively. The predicate* $\mathit{alloc}(o, st_a)$ *indicates the relation that* o *is allocated by* st_a*.* η *is defined as follows:*

$$\eta(T, q) = \{t \mid q \in \omega_{\mathcal{T}}(t), t \in T\} \tag{3.4}$$

The intuition behind Definition 3.4.5 is straightforward. The constraints in Equations 3.1 and 3.2 assure that the returned value of a container method call in P' is always

the same as the one in P, as the methods in P' query and modify the concerned container properties in the same way as the ones in P. Meanwhile, Equation 3.3 constrains the types of container objects manipulated by the same container method call, assuring the program P' is well-typed.

Obviously, we can explicitly examine the equations based on the abstract states to determine the methods and types in the replacements. Specifically, we utilize Equations 3.1 and 3.2 to identify possible methods for container method replacements (Section 3.5.2), and leverage Equation 3.3 to refine the replacements in the synthesis (Section 3.5.3). Formally, we state Theorem 3.4.1 to justify that container replacements assuring container behavioral equivalence finally guarantee behavioral equivalence.

Theorem 3.4.1 *Container behavioral equivalence relation is a behavioral equivalence relation, i.e.,*

$$P \simeq_{\mathcal{C}} P' \Rightarrow P \simeq P'$$

Proof. According to Definition 3.4.5, P' only differs from P in terms of the container allocation statements and container method calls. Therefore, we only need to prove that for each container method call $v = c.f_{\mathcal{C}}(v_1, \ldots, v_m)$ in P and the corresponding call $v' = c.f'_{\mathcal{C}}(v_1, \ldots, v_m)$ in P', we have the equality relation $[\![v]\!](s) = [\![v']\!](s')$, where $f'_{\mathcal{C}} := \psi_c(st_c)$. s and s' are the program states after the container method calls in P and P', respectively.

If not, we can find a control flow path l in P and l' in P' containing $st_c := v = c.f_{\mathcal{C}}(v_1, \ldots, v_m)$ and $st'_c := v' = c.f'_{\mathcal{C}}(v_1, \ldots, v_m)$, respectively, which are the first container method calls in l and l' violating the equality relation. According to Equation 3.1, we have $Q_1 = Q'_1$, where $(Q_1, T_1) := \alpha_{\mathcal{M}}(f_{\mathcal{C}})$ and $(Q'_1, T'_1) := \alpha_{\mathcal{M}}(\psi_c(st_c))$. Obviously, there exists a pair of container method calls located not after st_c and st'_c in l and l', respectively, which induce different modifiers upon a container property in Q. Denote the two method calls by st_p and st'_p, which invoke $g_{\mathcal{C}}$ and $g'_{\mathcal{C}}$, respectively. Assume that $(Q_2, T_2) := \alpha_{\mathcal{M}}(g_{\mathcal{C}})$ and $(Q'_2, T'_2) := \alpha_{\mathcal{M}}(g'_{\mathcal{C}})$. Then, we have

$$\exists q^* \in Q,\ \eta(T_2, q^*) \neq \eta(T'_2, q^*)$$

The method calls manipulate the container object o, of which the properties in Q are queried by st_c and st'_c afterwards. According to the definition of $\widetilde{\rho}_e$, we have $Q \subseteq \widetilde{\rho}_e(o)$,

thus we have

$$\exists q^* \in \widetilde{\rho}_e(o),\ \eta(T_2, q^*) \neq \eta(T_2', q^*)$$

This contradicts with Equation 3.2. Q.E.D.

Theorem 3.4.1 enables us to guarantee the behavioral equivalence by examining the container behavioral equivalence. For each container method call, we can identify its method candidates which satisfy the constraints in Definition 3.4.5. Finally, we can select efficient container candidates so that the replacements are likely to satisfy the complexity superiority.

3.4.3 Complexity Guidance

To achieve the optimization, we expect the new program to satisfy the complexity superiority. Specifically, the synthesis algorithm should be aware of the time complexity of each container method. To this end, we propose the method complexity specification (Section 3.4.3) and then define container complexity superiority to provide the effective guidance for the synthesis (Section 3.4.3).

Method Complexity Specification

To depict time complexity of a container method in a fine-grained manner, we define a family of time complexity functions $\mathcal{TC}$ to represent different time complexities, which include (amortized) constant time complexity, (amortized) linear time complexity, etc. The functions of amortized time complexity are introduced as symbols to distinguish them from constant time complexity, linear time complexity, etc. Meanwhile, there exists an order between several container methods even if they have the same time complexity function. For example, the method LinkedHashMap.put has to maintain the indexes in a linked list to preserve the insertion order, and consumes more time than the method HashMap.put. Based on $\mathcal{TC}$, we can formalize the **method complexity specification**.

Definition 3.4.6 (Method Complexity Specification) *The method complexity specification is a function* CS *mapping a container method* $f_{\mathcal{C}}$ *to its complexity score* $\theta \cdot tc(n)$*, indicating its time complexity and a constant factor.*

Example 3.4.5 *The methods* HashMap.put *and* LinkedHashMap.put *are mapped to* $\theta_1 \cdot tc(n)$ *and* $\theta_2 \cdot tc(n)$*, respectively, where* $\theta_1 < \theta_2$*.* $tc(n)$ *is the function of amortized constant time complexity.* $\theta_1 < \theta_2$ *indicates that the method* LinkedHashMap.put *consumes more time than the method* HashMap.put*.*

Based on Definition 3.4.6, we can measure the total time complexity score of container method calls by a function of n. Then we can naturally compare the order of the complexity scores of container method calls in two programs by comparing the coefficients of $tc_i(n)$, where $tc_i(n)$ is the time complexity function occurring in the complexity scores.

Container Complexity Superiority

Although the method complexity specification provides an abstraction of the efficiency of each container method, we are still unaware of the frequency of container method calls, and estimating the time complexity for a general program is far from practical. To establish effective guidance for synthesis, we define the **container complexity superiority** formally as the heuristic criteria of the complexity superiority.

Definition 3.4.7 (Container Complexity Superiority) *Let* P' *be the program obtained by applying* ψ_a *and* ψ_c *to* P*.* P' *has the container complexity superiority over* P *iff for any* $st_a \in \mathcal{S}_a$ *and* o *allocated by* st_a*, we have*

$$\sum_{st_c \in \mathcal{S}_c(o)} CS(\psi_c(st_c)) \leqslant \sum_{st_c \in \mathcal{S}_c(o)} CS(f_{\mathcal{C}}) \tag{3.5}$$

where $f_{\mathcal{C}}$ *is the container method in* st_c*, and* $\mathcal{S}_c(o)$ *contains the container method calls that manipulate* o *in an execution of* P*, i.e.,*

$$\mathcal{S}_c(o) = \{st_c \mid st_c := v = c.f_{\mathcal{C}}(v_1, \ldots, v_m) \in \mathcal{S}_c, o \in \widetilde{\varepsilon}_{st_c}(c)\} \tag{3.6}$$

$\widetilde{\varepsilon}_{st_c}$ *indicates the points-to facts before the statement* st_c*.*

Example 3.4.6 *Assume that we have specified the method complexity specifications as follows:*

$$CS(\textit{ArrayList.add}) = tc(n) \qquad CS(\textit{ArrayList.contains}) = n$$

$$CS(\textit{HashSet.add}) = 2 \cdot tc(n) \qquad CS(\textit{HashSet.contains}) = 1$$

$\mathtt{tc}(n)$ *is the time complexity function of amortized constant complexity. In Figure 3.2, the total time complexity score of the container method calls manipulating* o_2 *is* $2n + \mathtt{tc}(n)$*, After replacing it with a* HashSet *object, the score is* $2 + 2 \cdot \mathtt{tc}(n) < 2n + 2 \cdot \mathtt{tc}(n)$*, showing the container complexity superiority of the program after the replacements.*

Checking the complexity superiority requires precise reasoning of program complexity. However, deriving a tight bound of program complexity is stunningly difficult [102] and far from practical for a real-world program [101, 109], especially for programs involving sophisticated manipulations of data structures [100, 110, 111, 112]. Although the container complexity superiority does not imply the complexity superiority, it provides the effective guidance to find the opportunity of synthesizing the replacements to improve program efficiency, as evidenced by our evaluation in Section 3.7.

3.5 Synthesis Algorithm

This section presents our synthesis algorithm that achieves the goals described in Section 3.4.2 and Section 3.4.3. It takes as inputs the source code of a program P and the container method specifications. The algorithm finally computes the container replacement mappings ψ_a and ψ_c, based on which we can obtain a new program P'. As shown in Section 3.4.2 and Section 3.4.3, the container behavioral equivalence and the container complexity superiority pose sophisticated constraints for the container replacement mappings ψ_a and ψ_c. To satisfy all the constraints, our synthesis algorithm works with three stages as follows:

- To understand the intention of container usage, we present the container property analysis to determine which container-property queries occur upon a container object (Section 3.5.1).
- To assure P' queries and modifies container properties in the same way as P, we identify the method candidates for a container method call based on Equations 3.1 and 3.2 (Section 3.5.2).
- A synthesizer selects the methods from the method candidates with lowest time complexity to guarantee the container complexity superiority. A verifier performs

$$\frac{\widetilde{\rho} \vdash S_1 \rightsquigarrow \widetilde{\rho}_1 \quad \widetilde{\rho}_1 \vdash S_2 \rightsquigarrow \widetilde{\rho}'}{\widetilde{\rho} \vdash S_1; S_2 \rightsquigarrow \widetilde{\rho}'}$$

(SEQUENCING)

$$\frac{\widetilde{\rho} \vdash S \rightsquigarrow \widetilde{\rho}' \quad \widetilde{\rho} = \widetilde{\rho}'}{\widetilde{\rho} \vdash \mathtt{fix}(S) \rightsquigarrow \widetilde{\rho}}$$

(FIX-I)

$$\frac{\begin{array}{c}\widetilde{\rho} \vdash S_1 \rightsquigarrow \widetilde{\rho}_1 \quad \widetilde{\rho} \vdash S_2 \rightsquigarrow \widetilde{\rho}_2 \\ \widetilde{\rho}' = \widetilde{\rho}_1[o \mapsto \widetilde{\rho}_1(o) \cup \widetilde{\rho}_2(o) \mid o \in \mathcal{O}_c]\end{array}}{\widetilde{\rho} \vdash \textbf{if } (e) \textbf{ then } S_1 \textbf{ else } S_2 \rightsquigarrow \widetilde{\rho}'}$$

(BRANCH)

$$\frac{\begin{array}{c}\widetilde{\rho} \vdash S \rightsquigarrow \widetilde{\rho}_1 \quad \widetilde{\rho} \neq \widetilde{\rho}_1 \\ \widetilde{\rho}_2 = \widetilde{\rho}_1[o \mapsto \widetilde{\rho}(o) \cup \widetilde{\rho}_1(o) \mid o \in \mathcal{O}_c] \\ \widetilde{\rho}_2 \vdash \mathtt{fix}(S) \rightsquigarrow \widetilde{\rho}'\end{array}}{\widetilde{\rho} \vdash \mathtt{fix}(S) \rightsquigarrow \widetilde{\rho}'}$$

(FIX-II)

$$\frac{\begin{array}{c}(Q, T) = \alpha_{\mathcal{M}}(f_{\mathcal{C}}) \\ \widetilde{\rho}' = \widetilde{\rho}\,[o \mapsto \widetilde{\rho}(o) \cup Q \mid o \in \widetilde{\varepsilon}(c)]\end{array}}{\widetilde{\rho} \vdash v = c.f_{\mathcal{C}}(v_1, \ldots, v_m) \rightsquigarrow \widetilde{\rho}'}$$

(CONTAINERCALL)

$$\frac{\widetilde{\rho} \vdash \mathtt{fix}(S) \rightsquigarrow \widetilde{\rho}'}{\widetilde{\rho} \vdash \textbf{while } (e) \textbf{ do } S \textbf{ od} \rightsquigarrow \widetilde{\rho}'}$$

(LOOP)

Figure 3.6: Abstract transformers in the container property analysis.

type checking by examining whether Equation 3.3 holds to assure the container behavioral equivalence. If the type checking fails, the synthesizer refines the synthesized types and methods in the consequent rounds (Section 3.5.3).

We also state the soundness and complexity of the synthesis theoretically (Section 3.5.4). The soundness theorem guarantees that the new program P' must be behaviorally equivalent to P. For clarity, we use the program in Figure 3.2 to explain each stage of our approach throughout this section.

3.5.1 Container Property Analysis

According to Definition 3.4.1, we can compute container-property queries to reveal the intention of container usage. Suppose we have obtained points-to fact $\widetilde{\varepsilon}$ at each program location based on an off-the-shelf points-to analysis. Using the method semantic specification, we can easily compute the property-query fact $\widetilde{\rho}$ at each program location. Finally, we obtain the property-query fact $\widetilde{\rho}_e$ at the exit of the program, indicating all the container-property queries occurring in the program.

Figure 3.6 defines the abstract transformers of program statements. Specifically, we

should handle four program constructs, including a sequencing, a branch, a container method call, and a loop.

- The rule of sequencing is simple, in which the transformer is exactly the composition of the transformers of its parts.
- For a branch, the transformer merges the container-property queries occurring upon a container object along two paths.
- The rule CONTAINERCALL relies on the points-to fact $\widetilde{\varepsilon}$ before the statement to identify the container object o manipulated by the container method call. Q indicates the container-property queries induced by $f_{\mathcal{C}}$. To update $\widetilde{\rho}$, we merge $\widetilde{\rho}(o)$ with Q directly to show that the container-property queries in Q occur upon o.
- To compute the container-property queries in a loop, we need to calculate the fixed point by applying the transformer of the loop body iteratively. Due to the finite sizes of $\mathcal{Q}$ and $\mathcal{O}_c$, the fixed point must be reached after applying the rule FIX-II finite times.

Example 3.5.1 *Consider the* ArrayList *object* o_{11} *in Figure 3.2. According to the method semantic specification, we have* $\widetilde{\varepsilon}(r) = \{o_{11}\}$ *and* $\widetilde{\rho}(o_{11}) = \varnothing$ *before line 23. The method* ArrayList.contains *queries the container property* isVal(v)*, i.e., the value-ownership of* o_{11}*. By applying the rule* CONTAINERCALL*, we have* $\widetilde{\rho}'(o_{11}) = \{q\}$*, where* $q(s, c) =$ isVal(v)*, indicating that the value-ownership query has occurred upon* o_{11} *after line 23. Similarly,* o_{11} *is also manipulated by the container method call at line 38, and we can obtain* $\widetilde{\rho}_e(o_{11}) = \{q\}$ *at the exit of the program, which means that only the value-ownership query occurs upon* o_{11}*.*

It is worth noting that the container property analysis collects the queried container properties interprocedurally. For each function call, the analysis merges the queried container properties of its parameters and return value, such that the container properties queried upon a single container object in different functions can be aggregated. Eventually, we can understand the usage intention of each container object according to all the queried container properties. Besides, the container property analysis reasons how container objects are utilized in a flow-sensitive manner. Crucially, $\widetilde{\rho}_e$ over-approximates

Algorithm 1: Identifying method candidates.

Input: P: A container-manipulating program; $\alpha_{\mathcal{M}}$: Method semantic specification;
Output: $\widehat{\psi}_c$: Method candidate mapping;

```
1 S_a, S_c ← getSkeleton(P);
2 ρ̃_e ← getQueryFact(P);
3 ψ̂_c ← [st_c ↦ ∅ | st_c ∈ S_c];
4 foreach st_c := v = c.f_C(v_1, ..., v_m) ∈ S_c do
5     ε̃_{st_c} ← getPTFact(P, st_c);
6     foreach f'_C ∈ M do
7         if isEquivalent (f_C, f'_C, ρ̃_e, ε̃_{st_c}, α_M) then
8             ψ̂_c(st_c) ← ψ̂_c(st_c) ∪ f'_C;
9 return ψ̂_c;
```

the container-property queries occurring upon container objects and provides sufficient guidance for method candidate identification to guarantee the container behavioral equivalence. Notably, pointer analysis affects the precision of the container property analysis. When the points-to facts are imprecise, the container property analysis can discover that a container object o is manipulated by a container method call, while o is not pointed by c in any concrete execution. Therefore, $\widetilde{\rho}_e$ can contain the container-property queries which do not occur in any execution. We will quantify the effect of pointer analysis in the evaluation to show that its imprecision degrades the effectiveness of the replacements.

3.5.2 Method Candidate Identification

To guarantee the behavioral equivalence, we have to determine the container methods preserving the container behavioral equivalence. Specifically, the constraints in Equations 3.1 and 3.2 should be satisfied so that the concerned container properties can be queried and modified int the same way as the original program. Formally, we define the **method candidate** as follows.

Definition 3.5.1 (Method Candidate) *Given a container method call* $st_c \in \mathcal{S}_c$*, a container method* $f'_{\mathcal{C}} \in \mathcal{M}$ *is a method candidate of* st_c *if and only if it satisfies Equations 3.1 and 3.2.*

Essentially, we should compute the method candidate mapping $\widehat{\psi}_c : \mathcal{S}_c \rightarrow 2^{\mathcal{M}}$ to indicate the method candidates of a container method call. At a high level, we can leverage

the method semantic specification $\alpha_{\mathcal{M}}$ and the property-query fact $\widetilde{\rho}_e$ at the exit to identify the method candidates.

Algorithm 1 shows the procedure of identifying method candidates. It first utilizes the points-to fact $\widetilde{\varepsilon}_{st_c}$ to identify the container objects manipulated by the container method call st_c. getQueryFact returns the property-query fact $\widetilde{\rho}_e$ at the exit of P, and isEquivalent checks whether Equations 3.1 and 3.2 hold for a container method call st_c. Utilizing the method semantic specification $\alpha_{\mathcal{M}}$, isEquivalent enumerates each container object o manipulated by st_c and checks whether the method $f'_{\mathcal{C}}$ queries and modifies the concerned container properties of o in the same way as the original method $f_{\mathcal{C}}$ in P. Finally, Algorithm 1 collects the method candidates for each container method call.

Example 3.5.2 *Assume that* $\mathcal{C} = \{$*ArrayList, LinkedList, HashSet*$\}$*. Consider the object* o_{11} *in Figure 3.2. Utilizing the container property analysis, we obtain* $\widetilde{\rho}_e(o_{11}) = q$*, where* $q(s, c) =$ *isVal*(v)*. The container method calls* $st_c@l_{23}$ *and* $st_c@l_{38}$ *manipulate* o_{11} *at lines 23 and 38, respectively. The methods HashSet.contains and ArrayList.contains both induce the value-ownership query and do not induce any container-property modifier, so Equations 3.1 and 3.2 both hold. Similarly, we have*

$$\widehat{\psi}_c(st_c@l_{23}) = \widehat{\psi}_c(st_c@l_{38}) = \{\textit{ArrayList.contains, LinkedList.contains, HashSet.contains}\}$$

Theorem 3.4.1 states that container behavioral equivalence implies the behavioral equivalence. According to Algorithm 1, Equations 3.1 and 3.2 must hold if we select the method for a container method call st_c from its method candidate set $\widehat{\psi}_c(st_c)$. Next, we can assure the container behavioral equivalence as long as the replacements satisfy Equation 3.3, i.e., the new program P' is well-typed. Therefore, we can obtain a well-typed program P' after the container replacements, which is behaviorally equivalent to P.

3.5.3 Container Replacement Synthesis

To improve program efficiency, we should select the methods from $\widehat{\psi}_c(st_c)$ for each container method call st_c to satisfy the container complexity superiority in Definition 3.4.7. Besides, we have to conduct type checking by examining Equation 3.3 to assure the container behavioral equivalence.

Algorithm 2: Container replacement synthesis.

Input: P: A program; $\widehat{\psi}_c$: Method candidate mapping; CS: Method complexity specification;
Output: ψ_a, ψ_c: Container replacement mappings;

$\mathcal{S}_a, \mathcal{S}_c \leftarrow \texttt{getSkeleton}(P)$;
$\varphi_a, \varphi_c \leftarrow \texttt{getOriginalUsage}(P)$;
$\psi_a \leftarrow [st_a \mapsto \bot \mid st_a \in \mathcal{S}_a]$; $\psi_c \leftarrow [st_c \mapsto \bot \mid st_c \in \mathcal{S}_c]$;
$\sigma \leftarrow [st_a \mapsto \varnothing \mid st_a \in \mathcal{S}_a]$; $\widehat{\psi}_a \leftarrow [st_a \mapsto \mathcal{C} \mid st_a \in \mathcal{S}_a]$;
foreach $st_a \in \mathcal{S}_a$ **do**
    /* Synthesizer: Guess replacements */
    $min \leftarrow MAX\_CS$;
    $\mathcal{S}_a \leftarrow \mathcal{S}_a \setminus \{st_a\}$;
    foreach $\tau' \in \widehat{\psi}_a(st_a)$ **do**
        $\psi'_c \leftarrow \psi_c$;
        foreach $st_c \in \texttt{callSites}(st_a)$ **do**
            $\psi'_c(st_c) \leftarrow \texttt{getMinCS}(\widehat{\psi}_c(st_c) \cap \texttt{method}(\tau'), CS)$;
            if $\psi'_c(st_c) = \top$ **then**
                $\widehat{\psi}_a(st_a) \leftarrow \widehat{\psi}_a(st_a) \setminus \{\tau'\}$;
        $cur \leftarrow \texttt{getCSSum}(\texttt{callSites}(st_a), \psi'_c, CS)$;
        if $cur < min$ **then**
            $min \leftarrow cur$;
            $\psi_c \leftarrow \psi'_c$;
            $\psi_a(st_a) \leftarrow \tau'$;

    /* Verifier: Type checking */
    foreach $st'_a \in \mathcal{S}_a$ **do**
        if $\texttt{callSites}(st_a) \cap \texttt{callSites}(st'_a) \neq \varnothing$ **then**
            $\sigma(st'_a) \leftarrow \sigma(st'_a) \cup \{st_a\}$;
    $CE \leftarrow \{st'_a \mid st'_a \in \sigma(st_a),\ \psi_a(st'_a) \neq \psi_a(st_a),\ \psi_a(st'_a) \neq \bot\} \cup \{st_a\}$;
    if $|CE| > 1$ **then**
        foreach $st'_a \in CE$ **do**
            $\mathcal{S}_a \leftarrow \mathcal{S}_a \cup \{st'_a\}$;
            $\widehat{\psi}_a(st'_a) \leftarrow \bigcap_{st''_a \in CE} \widehat{\psi}_a(st''_a)$;
            $\psi_a(st'_a) \leftarrow \bot$;
            $\psi_c \leftarrow [st_c \mapsto \bot \mid st_c \in \texttt{callSites}(st'_a)]$;
return ψ_a, ψ_c;

To meet the two requirements, we instantiate a counterexample-guided inductive synthesis (CEGIS) paradigm [107, 106, 113, 114]. Algorithm 2 shows the procedure of container replacement synthesis. At a high level, it processes a container allocation statement st_a in a round and finally synthesizes the container replacement mappings ψ_a and ψ_c. Specifically, each round contains the following two steps:

- *Guess replacements*: The synthesizer selects the most efficient methods from the candidates for the container method calls manipulating o, where o is allocated by st_a.
- *Type checking*: The verifier performs type checking by examining Equation 3.3. The container allocation statements are reprocessed in the consequent rounds if they violate Equation 3.3.

Initially, Algorithm 2 sets the types and methods to $\perp$ in ψ_a and ψ_c to indicate undefined types and methods, respectively. Besides, it introduces the mapping $\widehat{\psi}_a$ to maintain feasible types for container allocation statements, and all the types are regarded as feasible initially. Each round of Algorithm 2 synthesizes the replacements for the container object o allocated by st_a. For clarity, we introduce the function callSites to obtain the container method calls manipulating o.

Next, to illustrate each step, we use the object o_{11} in Figure 3.2 as an example. Suppose that $st_a@l_{17}$ has been processed before $st_a@l_{11}$ in the CEGIS loop, where $st_a@l_{17}$ and $st_a@l_{11}$ allocate o_{17} and o_{11}, respectively. At the beginning of the round, we have $\psi_a(st_a@l_{17}) = \text{LinkedList}$ and $\text{HashSet} \notin \widehat{\psi}_a(st_a@l_{17})$, as the value-ownership of o_{17} is necessary in the program, and LinkedList supports more efficient insertions than ArrayList by avoiding memory reallocation.

Guess Replacements. The synthesizer enumerates $\tau' \in \widehat{\psi}_a(st_a)$ and utilizes getMinCS to find the method candidate supported by τ' with the lowest complexity (lines 11-14). If τ' does not support any method candidate, getMinCS returns a symbolic method $\top$ with MAX_CS as its time complexity score, and τ' is removed from $\widehat{\psi}_a(st_a)$, indicating that τ' is not the feasible type of st_a. Finally, the synthesizer selects the container type with the smallest sum of time complexity scores (lines 15-19).

Example 3.5.3 *The synthesizer selects the methods* HashSet.contains *and* HashSet.add *to manipulate* o_{11}. *Because the sum of their complexity scores is smaller than that of any other selection, the synthesizer enforces* $\psi_a(st_a@l_{11}) =$ HashSet.

Type Checking. The verifier performs type checking by examining whether Equation 3.3 holds (lines 25-26). If type checking fails, it adds the allocation statements to the set of counterexamples CE, which are refined by being reprocessed in the consequent rounds (lines 27-31). Moreover, we constrain that the counterexamples have the same set of feasible container types (line 29), pruning off the type selections causing the failure of type checking in the consequent rounds.

Example 3.5.4 *Before type checking, we have* $\psi_a(st_a@l_{17}) =$ LinkedList *and* $\psi_a(st_a@l_{11}) =$ HashSet. *The container objects* o_{11} *and* o_{17} *are both manipulated by the container method call at line 23, violating the constraint in Equation 3.3, so they are refined and reprocessed in the consequent rounds. At the end of this round, we have* HashSet $\notin \widehat{\psi}_a(st_a@l_{11})$, *as* HashSet *is not the feasible container type for* $st_a@l_{17}$. *Furthermore, their feasible type* LinkedList *is selected in the consequent rounds, finally passing type checking.*

Particularly, the verifier updates a mapping σ to show the relation between st_a and st'_a that the two allocated objects can be manipulated by the same container method call (lines 22-24). Intuitively, σ maintains the constraints for type checking, which are refined and utilized inductively in each round. To improve the efficiency of the synthesis, we use several data structures to cache the relationships frequently utilized in the synthesis. For example, we memorize the set of container method calls manipulating the container object allocated by a specific allocation statement so that we can get the value of callSites at lines 15, 23, and 31 without unnecessary recomputation.

Algorithm 2 synthesizes the container replacements inductively to guarantee the container behavioral equivalence and the container complexity superiority. Specifically, the counterexample-guided refinement assures that container behavioral equivalence must hold in the synthesis. Besides, the selected candidates have the lowest complexity among the method candidates, which assures the container complexity superiority. Even if type checking fails, the trivial selection, i.e., setting all the types and methods to the original ones, is still permissive in the consequent rounds, so the sum of the time complexity

scores can not be increased. Obviously, the method complexity specifications determine the complexity guidance and further affect the synthesized replacements. We will configure different specifications to quantify the influence and demonstrate the advantages of the form of our method complexity specifications in Definition 3.4.6.

3.5.4 Summary

Based on the sound points-to facts, our approach synthesizes the container replacements efficiently, which do not change the program semantics. We formulate two theorems to state the soundness and the complexity of Algorithm 2.

Theorem 3.5.1 (Soundness Theorem) *ψ_a and ψ_c provide sound container replacements, i.e., the program P' obtained by applying ψ_a and ψ_c for replacements has behavioral equivalence relation with the original program P.*

Proof. Based on Theorem 3.4.1, we only need to prove that for any $st_c \in \mathcal{S}_c$ and $f'_{\mathcal{C}} \in \widehat{\psi}_c(st_c)$, $f'_{\mathcal{C}}$ and $f_{\mathcal{C}}$ satisfy the three equations in Definition 3.4.5, where $f_{\mathcal{C}}$ is the container method invoked in st_c. In Algorithm 1, isEquivalent checks whether Equations 3.1 and 3.2 are satisfied. In Algorithm 2, the verifier performs type checking and examines whether Equation 3.3 holds. Given sound points-to facts, Equation 3.3 must hold for the synthesized container replacement mappings. Thus, the soundness of the synthesis totally relies on the soundness of the container property analysis.

The off-the-shelf points-to analysis provides a sound result $\widetilde{\varepsilon}$ for the abstract transformers in Figure 3.6. Consider an arbitrary container method call $v = c.f_{\mathcal{C}}(v_1, \ldots, v_m)$. For any concrete execution of the program, the container object manipulated by the method call can be abstracted by an abstract container object $o \in \widetilde{\varepsilon}(c)$. We use a set Q to denote the set of the container-property queries induced by the call, i.e., $(Q, T) := \alpha_{\mathcal{M}}(f_{\mathcal{C}})$.

The rule ContainerCall adds all the container-property queries in Q to $\widetilde{\rho}(o)$, which is a subset of $\widetilde{\rho}_e(o)$. Thus, the container-property queries occurring on the concrete container object must be included by $\widetilde{\rho}_e(o)$, which means the rule ContainerCall defines a sound abstract transformer for container method calls. Similarly, we can prove the other

three rules, i.e., the rules SEQUENCING, BRANCH and LOOP, define sound abstract transformers. Finally, the soundness of container property analysis assures the soundness of container replacements. Q.E.D.

Theorem 3.5.2 (Complexity of Synthesis) *Assume* $|\mathcal{S}_a| < |\mathcal{S}_c|$. *The time complexity of Algorithm 2 is* $O(|\mathcal{C}|^2 \cdot |\mathcal{M}| \cdot |\mathcal{S}_a| \cdot |\mathcal{S}_c|)$.

Proof. First, consider the guessing process, which corresponds to the steps from line 9 to line 19. The upper bound of the iteration count from line 9 to line 19 is

$$\sup_{st_a \in \mathcal{S}_a} |\widehat{\psi}_a(st_a)| = O(|\mathcal{C}|)$$

Similarly, the upper bound of the iteration count from line 11 to line 14 is $O(|\mathcal{S}_c|)$, as $\mathsf{callSites}(st_a) \subseteq \mathcal{S}_c$. Notice that the function getMinCS has to find the minimal value from at most $|\mathcal{M}|$ unordered elements, so it runs in $O(|\mathcal{M}|)$. The function getCSSum at line 15 also runs in $O(|\mathcal{S}_c|)$. In each round, the synthesizer guesses the replacements in

$$O(|\mathcal{C}| \cdot (|\mathcal{S}_c| \cdot |\mathcal{M}| + |\mathcal{S}_c|)) = O(|\mathcal{C}| \cdot |\mathcal{S}_c| \cdot |\mathcal{M}|)$$

Second, consider the counterexample generation in the type checking, which corresponds to the steps from line 22 to line 25. The upper bound of the iteration count is $|\mathcal{S}_a|$. Meanwhile, the disjointness checking at line 23 can be preprocessed in $O(|\mathcal{S}_a| \cdot |\mathcal{S}_c|)$ before the synthesis. By looking up the memorization, the step at line 23 can be achieved in constant time. Also, the construction of CE at line 25 runs in $O(|\mathcal{S}_a|)$. Therefore, the counterexamples are generated in $O(|\mathcal{S}_a|)$.

Third, consider the second loop in the type checking, which correspond to the steps from line 27 to line 31. The upper bound of the iteration count is $|CE| = O(|\mathcal{S}_a|)$. The computation at line 29 can be hoisted out of the loop, which runs in $O(|\mathcal{S}_a|)$. The result can be cached and reused in each iteration in $O(1)$. Therefore, the loop runs in $O(|\mathcal{S}_a|)$.

According to the above results, we can conclude that each round of the synthesis runs in

$$O(|\mathcal{C}| \cdot |\mathcal{M}| \cdot |\mathcal{S}_c| + |\mathcal{S}_a| + |\mathcal{S}_a|) = O(|\mathcal{C}| \cdot |\mathcal{M}| \cdot |\mathcal{S}_c| + 2|\mathcal{S}_a|)$$

Finally, consider the upper bound of the number of the rounds in the synthesis. According to the step at line 29, $|\widehat{\psi}_a(st'_a)|$ must decrease by at least one, where st'_a will

be resolved in the consequent round. On the one hand, $|\widehat{\psi}_a(st'_a)|$ is bounded by $|\mathcal{C}|$, as $\widehat{\psi}_a(st'_a) \subseteq \mathcal{C}$. On the other hand, $|\widehat{\psi}_a(st'_a)|$ must be larger than 0 at the end of the synthesis. Therefore, the number of the rounds is bounded by $|\mathcal{S}_a| \cdot |\mathcal{C}|$. Assume $|\mathcal{S}_a| < |\mathcal{S}_c|$. The time complexity of Algorithm 2 is

$$\begin{aligned}
&O(|\mathcal{C}| \cdot |\mathcal{M}| \cdot |\mathcal{S}_c| + 2|\mathcal{S}_a|) \cdot O(|\mathcal{S}_a| \cdot |\mathcal{C}|) + O(|\mathcal{S}_a| \cdot |\mathcal{S}_c|) \\
=&O(|\mathcal{C}|^2 \cdot |\mathcal{M}| \cdot |\mathcal{S}_a| \cdot |\mathcal{S}_c| + 2|\mathcal{C}| \cdot |\mathcal{S}_a|^2 + |\mathcal{S}_a| \cdot |\mathcal{S}_c|) \\
=&O(|\mathcal{C}|^2 \cdot |\mathcal{M}| \cdot |\mathcal{S}_a| \cdot |\mathcal{S}_c|)
\end{aligned}$$

Particularly, the assumption is introduced to simplify the estimated complexity of Algorithm 2. In general, the container object allocated by a container allocation statement is often manipulated by more than one container method call, and a container method call often only manipulates a single container object. Thus, the assumption holds in almost all the programs. Q.E.D.

It is worth mentioning that Theorem 3.5.2 does not provide a tight upper bound of the complexity. Actually, the overhead depends on the multiple aspects of the container usage. For example, the way of manipulating container objects affects the result of the container property analysis, and further determines the size of $\mathsf{callSites}(st_a)$. Besides, it is more likely to trigger the refinements if a large number of container objects are manipulated by the same container method calls. In practice, the synthesis performs with almost linear scalability, which is evidenced by our experiments.

3.6 Implementation

We have implemented our approach as a tool named CRES. The inputs of CRES are the source code and the method specifications, including the method semantic specification and the method complexity specification. CRES itself is implemented based on PINPOINT [3, 115], the static analysis platform in the Ant Group. When analyzing a Java program, the frontend of PINPOINT transforms the class files to LLVM IR [116], and then CRES identifies the container allocation statements and container method calls to obtain the skeleton. In what follows, we discuss more key configurations and designs for the synthesis algorithm.

Method Specifications. In the implementation, we focus on the containers in JCF. We specify the method semantic specification in the configuration file by assigning a pair of container-property queries and modifiers to each container method. Particularly, we adopt the insertion order and the key order as the partial orders to describe the container properties of LinkedHashMap and TreeMap, respectively. Besides, we provide the method complexity specification in a fine-grained manner. Specifically, we specify the constant factor θ along with time complexity to show the difference between the methods with the same time complexity. For example, the factor θ of LinkedList.add is smaller than that of LinkedHashSet.add, indicating that the latter consumes more time than the former due to the extra maintenance of a linked list, although they both run in amortized constant time.

On-demand Points-to Analysis. To support our container property analysis, we utilize PINPOINT to perform flow and context-sensitive pointer analysis. We are only concerned about the points-to facts of container variables, as the points-to facts of other variables do not affect the result of the container property analysis. Therefore, we query the points-to facts on demand to avoid unnecessary overhead in the container property analysis. Besides, the points-to facts at each program location are the prerequisite of examining the constraints to achieve the container behavioral equivalence and the container complexity superiority. To avoid redundant points-to query, we memorize the points-to facts and obtain the grace performance of the synthesis. Moreover, the points-to facts can also be obtained from other off-the-shelf pointer analyses [117, 50, 118], which means that CRES can be implemented easily based on other static analysis platforms [25, 54, 3].

3.7 Evaluation

We evaluate the effectiveness and efficiency of CRES by investigating the following three research questions:

- **RQ1:** What is the improvement CRES achieves for real-world programs?
- **RQ2:** Which kinds of container replacements does CRES synthesize?
- **RQ3:** What is the time and space overhead of CRES?

Result Highlights. In summary, CRES is unusually effective and efficient.

- *Significant efficiency improvement of experimental subjects*: The execution time is reduced by 8.1% on average. Particularly, CRES reduces the time consumption of the project google-http-java-client by 27.1%.
- *Various replacement patterns and many confirmations*: CRES discovers 107 replacements in six patterns, and 71 replacements have been confirmed by the developers. Several patterns, such as replacing ArrayList with HashSet, are uncovered by previous works.
- *Ability to scale to large-scale programs*: CRES finishes analyzing the project IoTDB with 384.2 KLoC in 14 minutes within 10 GB peek memory. The memory and time overhead is almost linear with the size of the project.

We also design a group of ablation studies to quantify the influence of the method complexity specifications and the precision of the pointer analysis. At the end of the evaluation, we discuss the quality of the replacements, the limitations of CRES, and several future directions.

3.7.1 Experimental Setup

Subjects. We evaluate CRES on 12 real-world data-centric applications, which are shown in Table 3.2. The projects are actively maintained and widely used in both academia and industry, covering different sizes (ranging from 18.6 KLoC to 384.2 KLoC) and diverse categories (such as microservice platforms, RPC frameworks, data management systems, etc.) Besides, the projects contain intensive usage of containers with various types, which provides more opportunities for CRES to find different patterns of container replacements.

Experimental Setting. For each project, we perform a whole-program analysis to obtain the points-to facts of container variables. Because we can not obtain the inputs for the projects in the real-world scenario, we follow the existing works and utilize the test suites of the projects to measure their time consumption [19]. To make the measurement more convincing, we repeat the execution of the test suite of each project 100 times and

Table 3.2: The medium ratio of reduced and original execution time and 95% confidence interval of the ratio.

Project	Description	Size (KLoC)	Medium (%)	95% CI (%)
bootique	Microservice platform	18.6	4.5	[4.4, 4.6]
mapper	Server application	22.4	7.3	[7.0, 7.6]
incubator-eventmesh	Eventing infrastructure	24.9	4.1	[3.9, 4.3]
google-http-java-client	Web client	25.2	27.1	[25.9, 28.3]
light-4j	Microservice platform	44.3	5.2	[5.0, 5.4]
roller	Server application	54.4	9.5	[9.2, 9.8]
IginX	Data management system	68.1	3.5	[3.4, 3.6]
sofa-rpc	RPC framework	76.4	3.7	[3.4, 4.0]
Glowstone	Server application	85.6	13.1	[12.9, 13.3]
dolphinscheduler	Eventing infrastructure	89.5	5.3	[5.1, 5.5]
dubbo	RPC framework	196.5	7.5	[7.2, 7.8]
IoTDB	Data management system	384.2	6.3	[6.2, 6.4]
			8.1	[7.8, 8.4]

perform Mann-Whitney U test to examine whether the improvement is statistically significant [119, 120]. We conduct all the experiments on a 64-bit machine with 40 Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz and 512GB of physical memory.

3.7.2 Answers to Research Questions

CRES aims to reduce the execution time of all the evaluated subjects. We quantify the effectiveness and efficiency of CRES by answering the three research questions.

Study of RQ1

We utilize the test suite of each project to measure the execution time of the test tasks affected by the replacements. Specifically, we compute the medium value and 95% confidence interval of the ratio between the reduced time consumption and the original one.

Table 3.2 shows the ratio of reduced time consumption and the original one for each project. The lower bound of 95% confidence interval is positive in each project, which means that CRES can improve the efficiency of all the projects statistically significantly. On average, the medium of reduced time cost ratio reaches 8.1%, and the 95% confidence

interval is [7.8, 8.4] . This demonstrates the effectiveness of CRES in improving the efficiency of real-world projects.

Particularly, the medium of reduced time ratio reaches 27.1% for the project google-http-java-client, and its 95% confidence interval is [25.9, 28.3]. The project is the HTTP client library for Java, supporting the access of the resource on the web via HTTP. Any project which depends on the library can benefit from the improvement of efficiency, which shows the significant impact of CRES. Another example is that the medium of the ratio reaches 13.1% for the project Glowstone. It is a customizable server for the game Minecraft, and its efficiency improvement promotes the performance of the service, shortening the response time of the interactions in the game. Generally, the improvement can benefit the applications depending on these projects, showing the great impact of CRES on the performance optimization of real-world programs.

Answer to RQ1: CRES improves the efficiency of all the subjects significantly, and the medium of reduced time ratio reaches 8.1% on average.

Study of RQ2

Table 3.3 displays the patterns of container type replacements synthesized by CRES. There are 107 replacements covering six different categories, and 71 replacements in six projects have been confirmed by the developers. If CRES replaces the container type of a container object, the methods in the container method calls manipulating the object are also replaced, e.g., the invoked container method at line 5 in Figure 3.2 is the method HashSet.contains in the replacements. However, the names of the methods are the same before and after the replacements in almost all the cases, so we do not discuss how the methods are replaced in detail.

Case Studies. We show two examples of typical replacement patterns as follows.

Transform ArrayList to HashSet. Figure 3.7(a) shows the container replacement in the project light-4j. Based on the result of the container property analysis, we find that the ArrayList object pointed by EXCLUSIONS and exclusions is only created for querying the value-ownership, as the method calls manipulating the object are the insertions or check-

Table 3.3: The counts of different replacements.

Project	#Conf/#Total	#R1	#R2	#R3	#R4	#R5	#R6
bootique	0/4			4			
mapper	0/6		5		1		
incubator-eventmesh	19/19	1	16	2			
google-http-java-client	0/4		4				
light-4j	0/5		2	3			
roller	0/6			5	1		
IginX	11/11		9		1		1
sofa-rpc	12/12		5			2	5
Glowstone	0/11		6	3	1		1
dolphinscheduler	7/7		6	1			
dubbo	12/12	1	3	1		2	5
IoTDB	10/10	2	1	6			1
	71/107	4	57	25	4	4	13

R1: LinkedList⇒ArrayList
R2: ArrayList⇒LinkedList
R3: ArrayList⇒HashSet
R4: TreeMap⇒HashMap
R5: LinkedHashMap⇒HashMap
R6: LinkedHashSet⇒HashSet

```
1 public boolean isExcluded(String s) {
2   List exclusions = new ArrayList<String>();
3   if (EXCLUSIONS != null)
4     exclusions = EXCLUSIONS;
5   return MANAGEMENT.equals(s)
6     || SCALABLE_CONFIG.equals(s)
7     || exclusionList.contains(s);
8 }
```

(a) An inefficient usage of ArrayList in light-4j

```
1 public T getPath(T p, T q, T r) {
2   List v = new ArrayList<>();
3   for (T c = p; c != q; c = c.pre())
4     v.add(c.pre().post().indexOf(c));
5   for (Integer i : v)
6     r= r.post().get(i);
7   return r;
8 }
```

(b) An inefficient usage of ArrayList in mapper

Figure 3.7: Examples of inefficient usage of containers.

ing whether an object is stored in the list. CRES synthesizes the replacement in which the types of EXCLUSIONS and exclusions are changed to HashSet safely, and the time complexity of querying the value-ownership can be reduced from linear complexity to amortized constant complexity.

Transform ArrayList to LinkedList. Figure 3.7(b) shows the transformation from ArrayList to LinkedList in the project mapper. The ArrayList object allocated at line 2 is manipulated by the method ArrayList.add and its iterator in the iteration. According to the documentation, we find that the method ArrayList.add has amortized constant time complexity due to the memory reallocation, while the method LinkedList.add has constant time complexity, and no memory reallocation occurs in the insertions. Meanwhile, there is no difference

in time complexity between the iterations over these two types of containers. Therefore, CRES synthesizes the replacement in which ArrayList is replaced with LinkedList to reduce the time cost of insertions.

Compared with the existing approaches [41, 19, 42], CRES can discover more general optimization patterns. As shown by the example in Figure 3.2, it can replace an ArrayList object with a HashSet object if only the method ArrayList.contains is invoked after its insertions. Although we can replace inefficient container types that conform to specific patterns, the way of replacing container types is not general. On the contrary, CRES offers a general framework for discovering various replacement patterns according to container method specifications and container usage intention. Meanwhile, we remark that CRES only aims to discover the replacements that can make the difference in time complexity and does not consider the environmental factors, such as the microarchitecture, and specific memory management strategies, such as garbage collection (GC) in JVM, which can also affect the execution time. Therefore, CRES might miss the opportunity of performing environment-dependent optimizations. However, Table 3.2 has shown that CRES is effective enough to improve the efficiency.

Answer to RQ2: CRES synthesizes 107 container replacements in various patterns without changing the program behavior, 71 of which have been confirmed.

Study of RQ3

We measure the time and memory overhead of CRES in the synthesis, which is shown in Figure 3.8. Overall, CRES finishes the analysis of the program with 384.2 KLoC in 14 minutes with 9.36 GB peak memory consumption. We adapt the regression analysis to study the observed complexity of CRES. The R-squared value for time and memory cost are 0.9796 and 0.9786, respectively, which are pretty close to 1. It indicates that the overhead of CRES grows nearly linearly at a gentle rate, permitting CRES to efficiently analyze large-scale programs manipulating containers. Compared with the existing works, CRES features with its efficiency and only needs 2.5 minutes to analyze per project. However, ARTEMIS executes the benchmark iteratively to obtain the optimal solution by genetic algorithm [19], and it spends 3.1 hours optimizing a project on average.

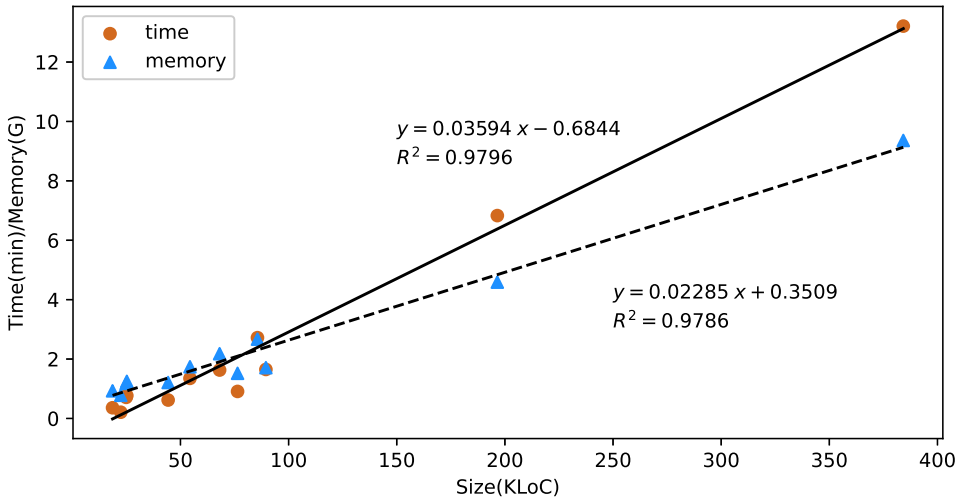


Figure 3.8: Time and memory overheads of CRES

Answer to RQ3: CRES features linear scalability and finishes the analysis in 14 minutes with 9.36 GB peak memory for the program with 384.2 KLoC.

3.7.3 Ablation Study

CRES leverages an off-the-shelf pointer analysis to identify the manipulated container objects, so the precision of the pointer analysis can affect the result of the container property analysis and the synthesized container replacements further. Besides, the method complexity specifications are specified manually, and the subjectivity of the specifications might also affect the replacements synthesized by CRES. Therefore, we set up the following ablations to investigate the influence of the pointer analysis and the method complexity specifications.

Ablation Study Setting

We propose two groups of the ablation studies to quantify the impact of pointer analysis and the method complexity, respectively.

- We use CRES-NS and CRES-RS to synthesize the replacements with different method complexity specifications. In CRES-NS, the constant factors are all equal to 1. In

Table 3.4: The counts of different replacements synthesized by the ablations

Project	#R1	#R2	#R3	#R4	#R5	#R6
bootique			(4, 4, 3)			
mapper		(5, 5, 3)		(1, 1, 0)		
incubator-eventmesh	(1, 1, 0)	(16, 16, 11)	(2, 2, 0)			
google-http-java-client		(4, 4, 1)				
light-4j		(2, 2, 2)	(3, 3, 1)			
roller			(5, 5, 4)	(1, 1, 0)		
IginX		(9, 9, 7)		(1, 1, 1)		(1, 0, 1)
sofa-rpc		(5, 5, 4)			(2, 0, 2)	(5, 0, 3)
Glowstone		(6, 6, 6)	(3, 3, 2)	(1, 1, 1)		(1, 0, 1)
dolphinscheduler		(6, 6, 5)	(1, 1, 1)			
dubbo	(1, 1, 1)	(3, 3, 2)	(1, 1, 1)		(2, 0, 1)	(5, 0, 4)
IoTDB	(2, 2, 1)	(1, 1, 1)	(6, 6, 3)			(1, 0, 1)

CRES-RS, the constant factors are randomly generated and conform to the specific order. For example, the constant factor of the method LinkedHashMap.put is larger than the one of the method HashMap.put.

- CRES-P leverages a flow and context-insensitive pointer analysis [117] to perform the container property analysis. Other modules and configurations are not changed.

We evaluate the three ablations upon the projects in Table 3.2. It is worth mentioning that the complexity function of each container method can be derived from the documentations, not inducing any bias in the manual configuration. Thus, we only quantify the influence of the constant factors in the method complexity specification.

Ablation Study Result

Table 3.4 shows the numbers of the container replacements synthesized by the three ablations [1], based on which we can obtain the following findings:

- CRES-NS can discover all the replacements synthesized by CRES except for the ones belonging to R5 and R6. Without the constant factors, CRES-NS can not distinguish the container methods with the same complexity function, such as the methods LinkedHashMap.put and HashMap.put.

[1] A triple shows the numbers of the replacements synthesized by CRES-NS, CRES-RS, and CRES-P.

- The replacements synthesized by CRES-RS are the same as the ones synthesized by CRES. The methods of the selected types have lower time complexity than the original ones in the first four patterns. Besides, all the methods of HashMap and HashSet have smaller constant factors than the ones of LinkedHashMap and LinkedHashSet, respectively.

- CRES-P synthesizes 74 container replacements out of 107 replacements synthesized by CRES. The imprecise points-to information yields the spurious container-property queries of several container objects, which prevents the synthesis algorithm from seizing the opportunity of optimizing the usage of the objects.

As we can see, the effectiveness of CRES does not largely depend on the manual configuration but relies on a precise pointer analysis. It is shown that CRES is easy to be configured by the users, and the documentations of the container methods provide the sufficient knowledge to specify the specifications, which can support the effective replacement synthesis for CRES.

We also apply the replacements synthesized by the ablations for each project and measure the improvement of program efficiency. It is shown that the 95% confidence intervals of the reduce time ratio are [7.0%, 7.6%] and [5.3%, 5.7%] on average for CRES-NS and CRES-P, respectively. We also measure the overhead of the synthesis for the three ablations. Specifically, CRES-NS and CRES-RS take the similar overhead to CRES, as the method complexity specifications only affect the selected method candidates and the types in Algorithm 2. CRES-P is more efficient with the benefit of the light-weighted pointer analysis, e.g., it finishes analyzing the project IoTDB in 10.3 minutes with 7.1 GB peek memory. However, CRES is practical enough for the real-world programs, as it does not suffer from the heavy overhead and synthesizes more replacements than CRES-P.

3.7.4 Discussion

Quality of Container Replacement. As shown by the answer to RQ2, CRES finds 107 replacements covering six categories. Notably, the developers greatly appreciated our efforts. For example, a developer of the project IoTDB commented that "*Since we often pay less attention to these details, if a tool can be used to do this work, it will be great!*" A developer of

```
public Point retrieveValidLastPoint(int n) {
  List<IChunkMetadata> seqDataList = new LinkedList<>();
  for (int i = 0; i < n; i++)
    seqDataList.add(getDataFromDevice());
  for (int i = seqDataList.size() - 1; i >= 0; i--) {
    Point lastPoint = getChunkLastPoint(seqDataList.get(i));
    if (lastPoint.getValue() != null)
      return lastPoint;
  }
  return null;
}
```

Figure 3.9: An example in which CRES fails to synthesize the optimal replacements

the project sofa-rpc was even inquired about CRES with comments like "*Where can I get the tool?*" Particularly, CRES has been integrated into the CI process in the Ant Group, which is a FinTech company with over one billion global users. The synthesized replacements can be forwarded to the developers as suggestions during the development cycle, making the applications deployed and executed economically.

Limitations. Although CRES is shown to be effective and efficient, it also comes with several limitations. First, the container complexity superiority formulates the complexity superiority in a heuristic manner. Generally, it is hard to derive a tight bound of the time complexity for a given program [100, 101, 102]. Meanwhile, our method complexity specification can not capture dynamic features dependent on the architectures [40] and might be imprecise for small inputs. However, it is almost infeasible to quantify the time cost of a container method for any input and execution environment. Fortunately, the experiential results have shown the effectiveness of the guidance provided by method complexity specification and container complexity superiority. Second, CRES can not always find optimal replacements. Consider the program extracted from the IoTDB in Figure 3.9. CRES can not discover that the random access in the second loop is merely used for traversing the container, so it replaces the LinkedList object with an ArrayList object, while the optimal solution should be replacing the random access in the second loop by an iterator.

Future Work. The insight underlying CRES is applicable to reduce other resource consumption and support other kinds of replacements. We only need to provide the method specifications in which the resource consumption of each container method is specified. For example, we can extend the method complexity specification to model the energy cost [98], and then CRES can optimize energy consumption seamlessly. Also, dynamic pro-

filing can be integrated to capture the runtime data [121] so that CRES can benefit from the static complexity model and the runtime overhead model simultaneously. Introducing annotations in container implementation can also be a good way of encoding the cost model to support the optimization. Also, when the libraries are updated, we need to update the specifications of new versions of container methods. Meanwhile, we can empower CRES with more container properties, such as boundedness. If only a finite number of insertions occur upon a container object, we can safely replace it with an array to avoid the memory bloat [33, 34]. Lastly, CRES is language-agnostic. It can easily extended to reduce inefficient container types in other program languages, such as C++ programs [122]. We believe that CRES provides a general framework to support the container replacements, reducing different kinds of resource consumption of a program.

3.8 Conclusion

We have introduced CRES, a novel synthesizer that automatically replaces inefficient container usage, optimizing data organization in data-centric application code. It analyzes the concerned container properties and finds more efficient container methods that preserve the behavioral equivalence to improve program efficiency. CRES is highly effective and efficient in analyzing real-world data-centric applications. It synthesizes 107 instances of container replacements covering six categories, which reduce the execution time by 8.1% on average. CRES also stands out with its excellent scalability, and finishes analyzing the project with 384.2 KLoC in 14 minutes. We hope the insight underlying CRES can be extended to reduce other resource consumption, such as memory, energy, and CPU usage.

CHAPTER 4

CONTAINER-AWARE VALUE-FLOW ANALYSIS VIA MEMORY ORIENTATION

4.1 Introduction

A container, e.g., *list*, *set*, or *map*, is an abstract data type that supports manipulating a collection of objects by its methods. General-purpose programming languages provide many implementations, such as the C++ STL containers [9], the Java Collections Framework (JCF) [10], and the specific classes in the Java EE framework (Java EE) [31]. In a data-centric application, they are pervasively utilized to manage the memory objects. Suppose we want to understand the data propagation in the systems to support various tasks, such as program understanding [123, 124, 125], bug detection [126, 127, 128, 129, 130], and debugging [131, 132]. In that case, we have to reason the semantics of containers and identify the value flows through containers.

Goal and Challenges. Our goal is to establish a fast and precise reasoning of container memory layouts for value-flow analysis. Unfortunately, the problem is always one of the "Achilles' heels" of static analysis [69]. Note that a container modification changes both which objects are stored, i.e., the ownership, and which indexes are associated with the objects, i.e., the index-value correlation. The precise reasoning of containers requires the strong updates upon container memory layouts, involving the program facts in multiple domains. First, we require a precise pointer analysis [133, 134, 135] to identify the manipulated objects, including both the containers and the elements. Second, applying strong updates to a list-like container relies on numeric analyses [136, 137, 138] to determine the relational positions of the manipulations. Third, the indexes of map-like containers are general comparable objects, and its strong updates often depend on complex relational properties of strings [139, 140, 141, 142] and user-defined data structures [58, 59]. More importantly, the prerequisites are closely intertwined, demanding a solution to address

```
Queue<Pr> ps = new LinkedList<Pr>();
Pr p = Space.getProject(dir);
ps.offer(p);
ps.offer(new Proj(newDir, arg));
executeProject(ps.peek());
```

(a) Example code in Hibernate-ORM

```
HttpSession<String,Object> s = new HttpSession<>();
s.put(Support.FIND_BLOCK, Boolean.FALSE);
s.put(Support.FIND_WHAT, searchWord);
String word = (String) s.get(Support.FIND_WHAT);
addToHistory(new EditorFindSupport(block, word));
```

(b) Example code in NetBeans

```
Stack<State> s = new Stack<>();
s.addElement(otherStateElem);
s.push(new State(cursor));
State state = includeStack.pop();
```

(c) Example code in Struts

```
Dictionary<String, String> config = new Hashtable<>();
config.put(Factory.CLASS, Driver.getName());
config.put(Factory.NAME, "iotdb");
String name = config.get(Factory.NAME);
```

(d) Example code in IoTDB

Figure 4.1: Examples of a programming idiom

them simultaneously. The overall quality of the results would collapse if any one of these analyses became imprecise.

Existing Effort. Reasoning container memory layouts is theoretically an undecidable problem [143]. Existing approaches mainly adopt two different strategies to achieve the over-approximation. One line of the techniques smashes a container and only reasons about the ownership without analyzing the indexes [144, 25, 51, 55, 12]. Although the analyses scale to large programs, the spurious value flows plague the analysis results such that 75.2% of the client analysis are false positives [13]. The other line of the techniques encodes the program values by logical formulae and applies strong updates by enforcing the container axioms [62, 63]. Despite the high precision, the exhaustive symbolic reasoning introduces a significant number of case analyses, causing the disjunctive explosion problem [145] and degrading the scalability significantly. For example, COMPASS only scales to 128 KLoC even when the solving procedure is optimized [146, 63].

Insight. Although analyzing generic containers is pie in the sky, there exists a particular class of containers, of which the memory layouts can be precisely tracked by deterministic indexes. As shown in Figure 4.1, for example, the modifications of the container objects always occur at the end or use constant keys, which can be discovered by tracking all possible modifications upon container objects. The stored objects can be identified by the deterministic indexes, which enables strong updates upon container memory layouts. Specifically, we formulate the idiom by the notion of *anchored containers*. A container is an anchored container at a program location if all the preceding modifications have deterministic indexes. Anchored containers widely exist in real-world programs, e.g., 75.6%

Java EE containers in the top 10 cases searched on GitHub conform to the programming idiom[1]. They establish the "anchors" for memory objects, which can be used to identify precise value flows through containers.

Solution. We introduce the *memory orientation analysis* to identify anchored containers and compute their precise index-value correlations, which further support a fast and precise value-flow analysis. Specifically, we establish a combined abstract domain embodied with path constraints to track multi-domain properties precisely, such as points-to facts and deterministic indexes. At a high level, our approach works in two stages:

- The memory orientation analysis identifies anchored containers and applies the strong updates to their memory layouts. A non-anchored container is smashed without analyzing its index-value correlation. Based on container memory layouts, the memory orientation analysis enables the construction of a precise value-flow graph. For example, the container objects o_1 and o_3 in Figure 4.2 are anchored containers. It index-value correlation implies that `p` is `null` and `r` is not `null` at line 21, and the analysis constructs the precise value-flow graph with the solid edges in Figure 4.3.
- We conduct a demand-driven reachability analysis to solve an instance of the value-flow problem. It collects the value-flow facts of interest when traversing the value-flow graph. The constraints are collected and solved to determine the reachability if necessary. For example, the null pointer exception (NPE) detector traverses the value-flow graph in Figure 4.3 from *null* values to dereferenced pointers, and reports an NPE with no false positive.

Note that it is non-trivial to identify and apply strong updates to anchored containers in the first stage, involving the accumulative effects of the modifications along control flow paths. For example, the container object o_2 are modified at lines 6, 8, and 9, and the last two modifications have non-deterministic indexes, so o_2 is not an anchored container after line 8. Particularly, we establish a subdomain to maintain the accumulative effects of modifications upon each container object, which explicitly indicates whether it is an anchored container. When transforming each subdomain simultaneously, we instantiate

[1]The empirical data is listed online: `https://containeranalyzer.github.io/empirical.pdf`.

[2]The program is simplified from Hibernate-ORM, IoTDB, and Struts.

```
1 void foo(String s) {
2   HttpSession hs;//o1
3   Map m = new HashMap<String,String>();//o2
4   hs.setAttribute("id", "a");
5   hs.setAttribute("age", null);
6   m.put("id", "b");
7   String i = hs.getAttribute("id");
8   if (c) {m.put(s, i);}
9   else {m.put(s, null);}
10  Stack<String> ids = new Stack<>();//o3
11  String j = m.get("id");
12  ids.add(i);
13  if (c) {ids.add(j);}
14  bar(hs, ids);
15 }
16 void bar(HttpSession hs, Stack ids) {
17  String p = hs.getAttribute("age");
18  String q = ids.peek();
19  String r = hs.getAttribute("id");
20  if (c)
21    out(p.length()+q.length()+r.length());
22 }
```

Figure 4.2: A motivating program[2]

a semantic reduction operator [61] to track the interleaving among multiple subdomains, and apply strong updates to anchored containers.

Highlight. The memory orientation analysis benefits value-flow analysis with three characteristics as follows:

- The memory orientation analysis applies strong updates to anchored containers in the combined domain rather than enforcing container axioms by logics exhaustively, making the analysis more precise than [25, 12] and less vulnerable to disjunctive explosion [145].
- Although the memory orientation analysis only computes the precise index-value correlations of anchored containers, it amplifies the precision benefit and obtains the more precise ownership information of other containers, no matter whether they are anchored containers or not.
- The memory orientation analysis divorces analyzing container semantics from value-flow analysis and delays reasoning about the feasibility of value-flow paths until client analyses, effectively alleviating the burden of constraint solving.

We implement ANCHOR and evaluate it upon several real-world data-centric applications by choosing thin slicing [123] and value-flow bug detection [30] as the clients. It is shown that ANCHOR enables a satisfactory improvement in thin slicing, reporting 17.1% fewer statements than the smashing-based slicer for the real-world programs. Moreover, it discovers all the taint flows through containers in the OWASP benchmark projects [147] with no false positive, and detects 20 NPEs in the real-world projects with 9.1% (2/22) as

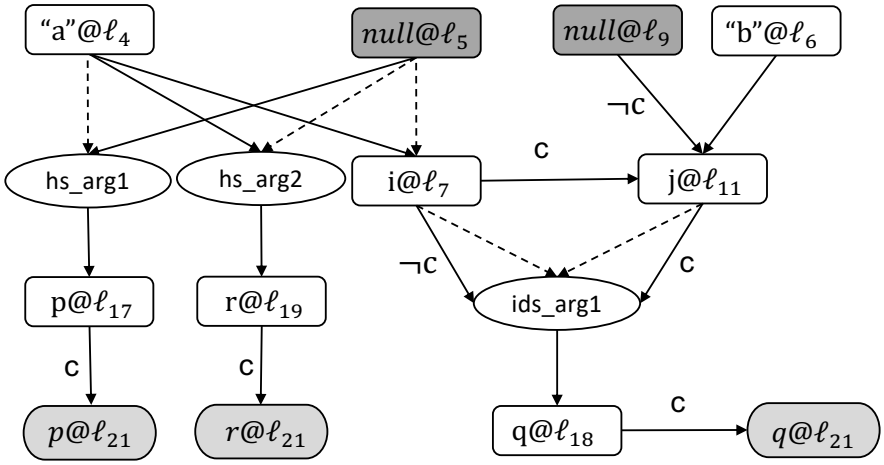


Figure 4.3: The value-flow graph (VFG)[3]of Figure 4.2. A node represents a value at a program location, and an edge from $a@\ell_1$ to $b@\ell_2$ indicates that the value a flows to the value b between the program locations ℓ_1 and ℓ_2. The nodes hs_arg1, ids_arg1, and hs_arg2 represent the auxiliary parameters [1, 2, 3], which indicate the elements accessed at lines 17, 18, and 19, respectively.

its false-positive ratio. In contrast, the smashing-based detector reports 31.0% spurious taint flows and 66.7% false positives of NPEs, respectively. Remarkably, ANCHOR features graceful scalability and finishes analyzing the program with 5.12 MLoC in 5 hours. Upon the submission, there had been 12 true positives of NPEs confirmed by the developers [148]. We also prove the soundness of ANCHOR theoretically. ANCHOR has been integrated into the static analysis platform PINPOINT in the Ant Group, an international FinTech company with over 1 billion global users. The deployment of ANCHOR demonstrates the impact of our approach in improving system reliability by detecting value-flow bugs in application code. In summary, we make the following main contributions:

- We introduce the notion of *anchored containers* and establish a combined abstract domain to identify them automatically.
- We propose the *memory orientation analysis* to apply strong updates to anchored containers, which promotes further value-flow analysis.
- We implement and evaluate ANCHOR by thin slicing and value-flow bug detection, showing its high precision and linear scalability in real-world scenarios.

[3]For simplicity, we omit the constraint ϕ in Figure 4.3 if $\phi = \mathsf{T}$.

- ANCHOR has been deployed in Ant Group to examine data propagation within data-centric applications. It has reported hundreds of bugs in the CI process. We have published the list of the bugs detected by ANCHOR online [148], along with the detailed empirical data of anchored containers.

The rest of the chapter is organized as follows. Section 4.2 shows the motivating example and the outline of our approach. Section 4.3 presents the preliminary background, and Section 4.4 formulates the problem we focus on. Section 4.5 defines the abstract memory, and Section 4.6 presents the details of the memory orientation analysis. Section 4.7 discusses the demand-driven reachability analysis and presents two typical clients of value-flow analysis, including thin slicing and value-flow bug detection. Sections 4.8 and 4.9 demonstrate the details of the implementation and the evaluation, respectively. Section 4.10 summarizes our work on container-aware value-flow analysis.

4.2 Overview

The section presents the container categorization, illustrates a motivating example, and finally outlines our overall idea.

4.2.1 Category of Containers

We adopt the classification in [63] and categorize the containers into two types, namely *position-dependent containers* and *value-dependent containers*. In a position-dependent container, e.g., ArrayList and Stack in JCF, each element has a position, representing the location where it is stored. A value-dependent container, e.g., the HashMap in JCF and HttpSession in Java EE, stores its elements based on their values. Particularly, the Set is also a value-dependent container. Generally, a container is manipulated by an method call at an *index*. An index is a non-negative integer in a position-dependent container, which denotes the position where the method manipulates the container, or a key in a value-dependent container, which is a comparable object, such as a string and other user-defined types.

4.2.2 Motivating Example

This section presents a motivating example program and discusses the limitations of existing efforts. Finally, we demonstrate our aim at the end of the section.

Program Description. Figure 4.2 shows a container-manipulating program, which typically appears in data-centric application code. Specifically, it contains two functions, i.e., foo and bar, and there are three container objects allocated in the function foo, namely the HttpSession object o_1, the HashMap object o_2, and the Stack object o_3. After inserting several elements into the container objects, the function foo invokes the function bar and passes o_1 and o_3 as the actual parameters. In the function bar, three elements are retrieved from the two container objects, and finally dereferenced at line 21. To simplify the example, we introduce a boolean variable c as the branch conditions, where c does not always evaluate to *true* or *false*.

Following existing efforts on value-flow analysis [54, 3], we leverage the value-flow graph (VFG) to demonstrate how each value propagates in the program, which is shown in Figure 4.3. Particularly, an edge $(v_1@\ell_i,\ v_2@\ell_j)$ labeled a constraint ϕ in the VFG indicates that the value can flow between v_1 at ℓ_i and v_2 at ℓ_j when ϕ holds, where ℓ_i and ℓ_j denote the positions in the control flow graph. Based on the VFG, we can further perform a variety of value-flow clients. In the NPE detection, a feasible path from *null* value to a dereferenced pointer indicates a possible NPE in the program. For example, the feasible path from $\mathtt{null}@\ell_5$ to $\mathtt{p}@\ell_{21}$ indicates that the dereference of p causes an NPE at line 21. Unfortunately, it is non-trivial to establish the VFG for a container-manipulating program, as it is required to analyze container memory layouts, which involves multiple sophisticated analyses, such as points-to analysis, numeric analysis, etc.

Limitations of Existing Approaches. Existing static approaches mainly analyze container memory layouts in two ways, which are not precise or scalable enough for real-world programs. One line of the recent efforts smashes each container into a set of objects and does not reason the indexes [144, 25, 51, 55, 12]. Although they often obtain gentle scalability, spurious value flows can plague the results of the clients. For example, a smashing-based analysis can introduce five dash edges in the VFG shown in Figure 4.3, which indicate the spurious value flows. Specifically, the dash edge from $\mathtt{null}@\ell_5$ to $\mathtt{i}@\ell_7$

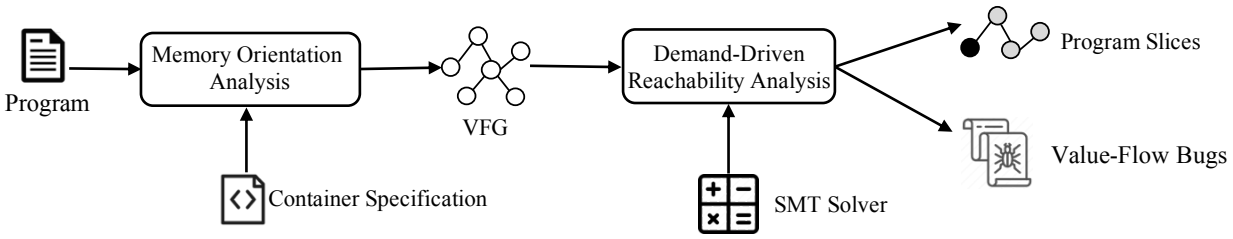


Figure 4.4: Schematic overview of our approach

shows the spurious fact that i can be *null* after line 7, as the smashing-based analysis does not analyze the index-value correlation of o_1. Similarly, it discovers that $q@\ell_{21}$ and $r@\ell_{21}$ can be *null* and finally yields two false positives.

The other line leverages symbolic analysis to compute the program values by logical formulae and applies strong updates to memory layouts by enforcing container axioms [62, 63]. Container memory layouts, including index-value correlations, are abstracted by logical formulae explicitly and exhaustively, and case analysis makes the number of the disjunctions exponentially large. For instance, s in Figure 4.2 does not have a deterministic value, which introduces two disjunctions for o_2 after line 8, corresponding to the cases that s is equal or not equal to "id", respectively. Thus, the analyses have to handle the verbose constraints, preventing its scalability significantly.

Our Aim. We aim to establish fast and precise reasoning of container memory layouts for value-flow analysis to investigate data propagation within data-centric applications. Specifically, we expect the analysis to obtain precise value flows through containers so that it could remove more spurious value flows than smashing-based approaches. Besides, the analysis is required to exhibit high efficiency and gentle scalability when analyzing large-scaled programs. It would have a great impact on many static clients, including thin slicing and value-flow bug detection, and thus, promote the understanding and improve the reliability of data-centric applications with heavy usage of containers.

4.2.3 Our Approach

Our work balances the tension between precision and scalability by utilizing a programming idiom. In many cases, the modifications of a position-dependent container occur at

its beginning or end, while the keys of a value-dependent container are constant. Given a container memory layout, the modified layout is unique if the index of the modification is deterministic. Moreover, if all the modifications upon a container have deterministic indexes before the program location ℓ, its memory layout can be precisely determined at ℓ by tracking the modifications before ℓ along control flow paths. We define the class of containers as *anchored containers*, which enable strong updates and further promote a precise value-flow analysis.

Utilizing anchored containers as our sweet spot, we propose the *memory orientation analysis* to analyze container memory layouts precisely. Figure 4.4 shows the schematic overview of our approach. In the high level, our approach consists of two phases, namely the memory orientation analysis and the demand-driven reachability analysis. In the first phase, the memory orientation analysis tracks multi-domain properties simultaneously, such as points-to facts and deterministic indexes, which support identifying anchored containers for strong updates. Based on container memory layouts, it constructs a precise value-flow graph, e.g. constructing the graph with the solid edges in Figure 4.3 for the program in Figure 4.2. In the second phase, we conduct a demand-driven reachability analysis by the graph traversal to solve an instance of the value-flow problem. Note that we do not invoke SMT solvers in the memory orientation analysis but store the constraints compactly in the graph, which are only collected and checked on demand in the traversal to avoid unnecessary overhead.

Benefit. The memory orientation analysis unleashes the strength of anchored containers with twofold benefits:

- Precise memory layouts of anchored containers support discovering more precise value flows and promotes a chain of further analyses in the clients. For example, the anchored container o_1 in Figure 4.2 finally enables the analysis to avoid the false positives in the NPE detection.

- The precision benefit can be further amplified, which promotes the reasoning the memory layouts of other containers. For example, the non-anchored container o_2 exclude the *null* value defined at line 5, as the anchored container o_1 makes the *null* not reach i at line 7.

$$
\begin{aligned}
&\textit{Program} && P := F+ \\
&\textit{Function} && F := f(v_1, v_2, \ldots)\{S;\} \\
&\textit{Statement} && S := v = \textbf{new}\ \ \tau \mid v = u \mid v = a \\
& && \mid c.\textbf{insert}(u, v) \mid c.\textbf{remove}(u) \mid v = c.\textbf{access}(u) \\
& && \mid S_1; S_2 \mid \textbf{if}\ (v)\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \mid r = \textbf{call}\ f(v_1, v_2, \ldots) \\
& && \mid \textbf{foreach}\ \ (u, v)\ \ \textbf{in}\ \ c\ \ \textbf{do}\ \ S\ \ \textbf{od} \mid \textbf{return}\ v
\end{aligned}
$$

Figure 4.5: The syntax of the language

The motivating example illustrates the workflow of ANCHOR. From the example, we find it is crucial but non-trivial to identify and utilize anchored containers. In Section 4.4, we formulate our problem and formally define anchored containers, following the outline of other sections.

4.3 Preliminaries

The section presents several preliminary concepts, including program syntax, concrete memory, concrete semantics, and value-flow graph.

4.3.1 Program Syntax

We borrow the language syntax in [63] and formalize our analysis with a simple Java-like language in Figure 4.5. A program is in the static single assignment form [149]. Statements include allocation statements, assignments, container method calls, sequencing, branches, function calls, container traversals, and return statements. Particularly, the right-hand side of an assignment can be a variable or a literal.

We analyze two types of containers, namely position-dependent containers and value-dependent containers. Container method calls have three cases, namely inserting, accessing, and removing elements. Each container method can manipulate a container at a specific index. Particularly, we add an artificial index τ_e to represent the end of a position-dependent container so that the language supports adding an element at the end more flexibly. Besides, a loop can traverse a container, and all the elements are accessed once

exactly. We assume that a loop in the program is memoryless [150], i.e., the order of the iterations does not affect the semantics of the loop. The language also supports the nesting of containers because v in the insertion can point to another container.

Remark. Our work mainly focuses on the semantic analysis of container manipulations. We do not discuss how to handle the fields of memory objects, although we achieve the field sensitivity based on existing techniques [3, 115]. The language syntax in Figure 4.5 also omits several program constructs, such as reflective method calls, which are not the major concerns of our work. With the benefit of the formulation of the above syntax, our analysis can ensure the soundness in analyzing all the program constructs shown in Figure 6.4, which is proven in Section 4.6.7.

4.3.2 Concrete Memory and Concrete Semantics

Given a program P, a program location ℓ is the position in the control flow graph. We regard program memory as a collection of values $\nu \in \mathcal{V}$ bound with addresses $\alpha \in \mathcal{D} \subseteq \mathcal{V}$ and indexes $\delta \in \Delta \subseteq \mathcal{V}$. Specifically, a memory location is denoted by a pair $(\alpha, \delta) \in \mathcal{D} \times \Delta$. An index δ can be an address or a literal in the program. We introduce two sets $\mathcal{D}_p$ and $\mathcal{D}_v$ to denote the set of the addresses where the position-dependent and value-dependent containers are stored, respectively.

Definition 4.3.1 (Concrete Memory State) *A concrete memory state* M *at the program location* ℓ *is* (E, L)*, where*

- *The* environment E *is a function mapping a program variable* $u \in \mathcal{X}$ *to a value* $\nu \in \mathcal{V}$*, indicating the value of the variable. Particularly,* $\mathsf{E}(u)$ *is the address where the object pointed by* u *is stored if* $\mathsf{E}(u) \in \mathcal{D}$*.*
- *The* layout L *maps a memory location* $(\alpha, \delta) \in \mathcal{D} \times \Delta$ *to the pair of an index and a value* $(\delta, \nu) \in \Delta \times \mathcal{V}$*. Particularly,* $\perp \in \mathcal{V}$ *is introduced to show that there does not exist any value stored at the index.*

Based on this concrete memory, we can define the operational semantics of container methods straightforwardly, which is similar to the definitions in [63]. To simplify the

$$\textsf{Ins}\ \frac{\begin{array}{c}E(c) = \alpha_c \quad E(u) = \nu_u \quad E(v) = \nu_v \quad L'(\alpha_c,\ \nu_u) = \nu_v \\ L' = L[(\alpha_c,\ j) \to (j,\ sec(L)(\alpha_c,\ j-1)) \mid j > \nu_u + 1] \ \text{ if } \ \alpha_c \in \mathcal{D}_p \\ L' = L[(\alpha_c,\ \nu_u) \to (\nu_u,\ \nu_v)] \text{ if } \alpha_c \in \mathcal{D}_v\end{array}}{(E,\ L) \vdash c.\textbf{insert}(u, v) : (E,\ L')}$$

$$\textsf{Rem}\ \frac{\begin{array}{c}E(c) = \alpha_c \quad E(u) = \nu_u \\ L' = L[(\alpha_c,\ j) \to (j,\ sec(L)(\alpha_c,\ j+1)) \mid j \geqslant \nu_u] \ \text{ if } \ \alpha_c \in \mathcal{D}_p \\ L' = L[(\alpha_c,\ \nu_u) \to (\nu_u,\ \bot)] \ \text{ if } \ \alpha_c \in \mathcal{D}_v\end{array}}{(E,\ L) \vdash c.\textbf{remove}(u) : (E,\ L')}$$

$$\textsf{Acc}\ \frac{\begin{array}{c}E(c) = \alpha_c \quad E(u) = \nu_u \\ L(\alpha_c, \nu_u) = (\nu_u, \nu') \quad E' = E[v \to \nu']\end{array}}{(E,\ L) \vdash v = c.\textbf{access}(u) : (E',\ L)}$$

$$\textsf{Proc-I}\ \frac{P = \varnothing}{(E,\ L) \vdash \mathrm{proc}(u, v, P, S) : (E,\ L)} \qquad \textsf{Proc-II}\ \frac{\begin{array}{c}P = P' \cup \{(\delta',\ \nu')\} \\ E_1 = E[u \to \delta',\ v \to \nu'] \\ (E_1,\ L) \vdash S : (E_2,\ L_2) \\ (E_2,\ L_2) \vdash \mathrm{proc}(u, v, P', S) : (E',\ L')\end{array}}{(E,\ L) \vdash \mathrm{proc}(u, v, P, S) : (E',\ L')}$$

$$\textsf{Loop}\ \frac{\begin{array}{c}E(c) = \alpha_c \quad P = \{(\delta_i,\ \nu_i) \mid (\delta_i,\ \nu_i) \in L(\alpha_c,\ \delta_i),\ \nu_i \neq \bot\} \\ (E,\ L) \vdash \mathrm{proc}(u, v, P, S) : (E',\ L')\end{array}}{(E,\ L) \vdash \textbf{foreach}\ \ (u, v)\ \ \textbf{in}\ \ c\ \ \textbf{do}\ \ S\ \ \textbf{od} : (E',\ L')}$$

Figure 4.6: Concrete semantics of container-manipulating programs

notation, we introduce the mapping $sec(L)$ to get the value stored at a specific index in a container, i.e., $L(\alpha, \delta) = (\delta,\ sec(L)(\alpha, \delta))$. Figure 4.6 shows the definitions of concrete semantics.

- The rules Ins and Rem define the concrete operational semantics of the insertion and removal, respectively. Due to the difference between position and value-dependent containers, the two rules conduct the case analysis when modifying the layout in the concrete memory.
- For the access method call, the rule Acc obtains the value ν' stored at the index $E(u)$ and enforce the $E'(v)$ equal to the value ν'.
- A loop traversing a container takes each index-value pair in the container to execute the statement S in each iteration. Particularly, the helper rule Proc is defined inductively to exercise the traversal.

Example 4.3.1 *In Figure 4.2, the method* setAttribute *adds two key-value pairs in the container*

object pointed by hs *at lines 4 and 5. After line 5, we have*

$$E(hs) = \alpha,\ L(\alpha, \text{“id”}) = (\text{“id”}, \text{“a”}),\ L(\alpha, \text{“age”}) = (\text{“age”}, null)$$

4.3.3 Value-Flow Graph

A value q flows to p if q is assigned to p directly (via an assignment, such as $p = q$) or indirectly (via container method calls, such as $c.\textbf{insert}(0, p); q = c.\textbf{access}(0)$). We can construct a graph to abstract how the value reaches a program location from another by an edge labeled with a constraint. Formally, we define the value-flow graph as follows.

Definition 4.3.2 (Value-Flow Graph) *A value-flow graph (VFG) is a directed graph* $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Theta)$*, where* $\mathcal{N}$*,* $\mathcal{E}$ *and* Θ *are defined as follows:*

- $\mathcal{N}$ *is a set of nodes, each of which is denoted by* $v@\ell$*, indicating* v *is defined or used at a program location* ℓ*.*
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ *is a set of edges.* $(v_1@\ell_1, v_2@\ell_2) \in \mathcal{E}$ *means that the value* $v_1@\ell_1$ *flows to* $v_2@\ell_2$*.*
- Θ *maps each edge to a constraint* ϕ*, meaning that the value-flow relation holds only when* ϕ *is satisfied.*

Example 4.3.2 *In Figure 4.2, we can find that* “a” *is associated with the index* “id” *before line 7, so we add the edge from* “a”$@\ell_4$ *to* i$@\ell_7$ *to the VFG in Figure 4.3, indicating the value flow induced by container method calls at lines 4 and 7.*

Following the previous studies [54, 3, 30], we formulate value-flow analysis as a reachability problem over a VFG. We can track the value-flow facts for various clients, such as thin slicing [123], value-flow bug detection [30], etc. For example, we collect all feasible paths from null to the dereferenced values in the NPE detection.

4.4 Container-Aware Value-Flow Problem

To obtain precise value-flow paths, we need to identify the indirect flows induced by container method calls. Concretely, we need to determine the objects accessed by each

access method call, and the reachability relation should perceive the induced value flows. We call this *container-aware value-flow problem*, which is a fundamental concern in static analysis, especially for the investigation of data propagation in data-centric applications.

Based on the semantics of container method calls, container memory layouts, i.e., the ownership and index-value correlations, determine the value flow through containers. First, an object is never accessed if it is not stored in the container. Second, an object can be accessed by the access method call at the index δ if associated with δ. Now we formalize the two properties and provide the formal definition of the container memory layout.

Definition 4.4.1 (Container Memory Layout) *Assume that a container object* o *is stored at the address* $\alpha \in \mathcal{D}$ *in the concrete memory* $M = (E, L)$. *The memory layout of the container object* o *consists of the following two properties:*

- *Ownership: It indicates whether there exists* $\delta \in \Delta$ *for a value* v *such that* $L(\alpha, \delta) = (\delta, v)$. *For a value-dependent container object* o, *it also indicates whether there exists* $v' \in \mathcal{V}$ *for a value* v *such that* $L(\alpha, v) = (v, v')$.
- *Index-value correlation: For any pair of an index and a value* (δ, v), *the index-value correlations indicates whether* v *is paired with* δ *at the address* α, *i.e.,* $L(\alpha, \delta) = (\delta, v)$.

It is necessary to analyze container memory layouts precisely and efficiently to support analyzing real-world programs using containers. However, precise reasoning of container memory layouts, as other non-trivial semantic properties, is an undecidable problem for general programs [143]. Fortunately, as explained in Section 4.2.3, there is a typical class of container objects of which modifications occur at deterministic indexes. Their memory layouts are deterministic after the modifications, which enables precise reasoning without case analysis. Formally, we define the notion of *anchored container* as follows.

Definition 4.4.2 (Anchored Container) *A container object* o *is an anchored container at the program location* ℓ *if an arbitrary modification method call* st *before* ℓ *has the constant index* δ. *Particularly,* τ_e *is a constant index.*

Intuitively, we can identify anchored containers and analyze their memory layouts precisely. Meanwhile, the precision enhancement introduced by anchored containers also

benefits analyzing memory layouts of other containers, even if the containers are not anchored containers.

Example 4.4.1 *In Figure 4.2,* setAttribute *modifies* o_1 *upon constant keys at lines 4 and 5, so* o_1 *is an anchored container after line 5. Therefore,* i *must be equal to "*a*" at line 7. Meanwhile,* o_2 *is not an anchored container, as* s *is not constant at lines 8 and 9. It is worth noting that we still obtain more precise ownership of* o_2 *that the* null *at line 5 is excluded, showing that the precision enhancement can even propagate to non-anchored containers.*

Roadmap. To solve the container-aware value-flow problem, we propose the memory orientation analysis, which is our main technical contribution, to identify and apply strong updates to anchored containers. The combined effects of container semantics are analyzed without sophisticated reasoning of indexes, and finally encoded in the VFG. However, as explained in Section 4.1, it is non-trivial to enable the identification, involving analyzing the facts in multiple domains simultaneously. Specifically, we present a novel memory abstraction to maintain the multi-domain program facts (Section 4.5), based on which the memory orientation analysis reasons container semantics by applying abstract transformers (Section 4.6.1~Section 4.6.5) and constructs a precise VFG (Section 4.6.6). For a specific client, we conduct a demand-driven reachability analysis by traversing the VFG (Section 4.7), which benefits from the precise enhancement provided by the memory orientation analysis.

4.5 Abstract Memory

The section presents the abstract memory used in this work (Section 4.5.1~Section 4.5.3), and highlights the technical challenges of memory orientation analysis that rests on the abstraction (Section 4.5.4).

4.5.1 Abstract Memory State

We abstract the memory based on allocation sites [52] and form a finite set of abstract objects $\mathbf{O} := \mathbf{O}_p \cup \mathbf{O}_v \cup \mathbf{O}_s$, where $\mathbf{O}_p$, $\mathbf{O}_v$, and $\mathbf{O}_s$ are the sets of position-dependent

container objects, value-dependent container objects, and non-container-typed objects, respectively. Besides, we construct a finite set of literals $\mathbf{O}_c \subseteq \mathbf{O}_s$, where $\hat{\tau}_e \in \mathbf{O}_c$ represents the end position of a position-dependent container. $\mathbf{X}$ is a set of program variables, and Φ is a set of path constraints. Formally, we define abstract memory state as follows.

Definition 4.5.1 (Abstract Memory State) *An abstract memory state* $\mathbf{M}$ *at the program location* ℓ *is a 4-tuple* $(\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U})$*. Here,* $\mathbf{E}, \mathbf{L}, \mathbf{C}$*, and* $\mathbf{U}$ *are defined as follows.*

- *Abstract environment* $\mathbf{E}$ *maps a program variable* v *to a set of abstract memory object (*o*) and constraint (*ϕ*) pairs, indicating* v *points to* o *when* ϕ *holds.*
- *Abstract layout* $\mathbf{L} := (\mathbf{B}, \mathbf{R})$ *contains two subdomains:*
  - $\mathbf{B} := \mathbf{O}_p \cup \mathbf{O}_v \to \mathcal{B}$ *abstracts the ownership of each container object, where*

    $$\mathcal{B} := \mathcal{P}(\mathbf{O} \times \Phi) \times \mathcal{P}(\mathbf{O} \times \Phi)$$

    Basically, we set the first entry of $\mathbf{B}(o_p)$ *to* $\{(o, \mathsf{T}) \mid o \in \mathbf{O}\}$ *for* $o_p \in \mathbf{O}_p$*, indicating that we only concern the stored objects without considering their positions in a position-dependent containers.*
  - $\mathbf{R} := \mathbf{O}_p \cup \mathbf{O}_v \to \mathcal{R}_p \cup \mathcal{R}_v$ *abstracts the index-value correlations of the container objects, where*

    $$\mathcal{R}_p := \mathcal{P}(\cup_{k=0}^{N} \mathbf{O}^k \times \Phi), \quad \mathcal{R}_v := \mathcal{P}(\mathbf{O}_c \times \mathbf{O} \times \Phi)$$

    N *is the number of insertions upon position-dependent containers, bounding the sizes of the container objects.*
- *Constant domain* $\mathbf{C}$ *is a function of program variables.* $\mathbf{C}(v) \in \mathbf{O}_c$ *when* v *must point to a literal, and* $\mathbf{C}(v) = \bot$ *when* v *is not initialized. Otherwise,* $\mathbf{C}(v) = \top$*.*
- *Uniqueness domain* $\mathbf{U}$ *maps* $o \in \mathbf{O}_p \cup \mathbf{O}_v$ *to 1 if* o *is always modified upon deterministic indexes. Otherwise,* $\mathbf{U}(o) = 0$*.*

Essentially, an abstract environment $\mathbf{E}$ over-approximates the points-to facts, i.e., v may point to a memory object o in a concrete execution if ϕ is satisfied, where $(o, \phi) \in$

$\mathbf{E}(v)$. Similarly, the ownership and the index-value correlation of each container are over-approximated by the separate subdomains of $\mathbf{L}$. Meanwhile, a constant domain $\mathbf{C}$ under-approximates whether a variable points to a literal. Lastly, $\mathbf{U}$ stores the accumulative effects of the modifications upon each container object, and thus, serves as the criteria of identifying anchored containers.

Definition 4.5.2 (Criteria of Anchored Container) *An abstract container object* o *is anchored at the program location* ℓ *if* $\mathbf{U}(o) = 1$ *where* $\mathbf{M} = (\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U})$ *is the abstract memory at* ℓ*.*

Example 4.5.1 *Consider* $o_1 \in \mathbf{O}_v$ *pointed by* $\mathtt{hs}$ *in Figure 4.2.* $\mathtt{ind}_1$ *and* $\mathtt{ind}_2$ *are the indexes of the insertions at lines 4 and 5, respectively. Obviously, we have* $\mathbf{C}(\mathtt{ind}_1) =$ "$\mathtt{id}$" *and* $\mathbf{C}(\mathtt{ind}_2) =$ "$\mathtt{age}$"*. Therefore,* o_1 *is always modified upon deterministic indexes after line 5. Concretely, we have*

$$\mathbf{B}(o_1) = (\{(\text{“}\mathtt{id}\text{”}, \mathsf{T}), (\text{“}\mathtt{age}\text{”}, \mathsf{T})\}, \{(\text{“}\mathtt{a}\text{”}, \mathsf{T}), (\mathtt{null}, \mathsf{T})\})$$

$$\mathbf{R}(o_1) = \{((\text{“}\mathtt{id}\text{”}, \text{“}\mathtt{a}\text{”}), \mathsf{T}), ((\text{“}\mathtt{age}\text{”}, \mathtt{null}), \mathsf{T})\}, \;\; \mathbf{U}(o_1) = 1$$

Particularly, $\mathbf{U}(o_1) = 1$ *indicates that* o_1 *is an anchored container after line 5.*

4.5.2 Join Operator and Partial Order

Given the definitions of $\mathbf{O}$, Φ, and $\mathbf{M}$, we introduce the join operator and the partial order of the combined domain.

Definition 4.5.3 (Join Operator of M) $\sqcup_{\mathbf{M}}$ *is the join operator of* $\mathbf{M}$*, i.e.,* $\mathbf{M} = \mathbf{M}_1 \sqcup_{\mathbf{M}} \mathbf{M}_2$*, where* $\mathbf{M}_1 = (\mathbf{E}_1, \mathbf{L}_1, \mathbf{C}_1, \mathbf{U}_1)$ *and* $\mathbf{M}_2 = (\mathbf{E}_2, \mathbf{L}_2, \mathbf{C}_2, \mathbf{U}_2)$*.* $\mathbf{M} = (\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U})$ *is defined as follows.*

$$\mathbf{E}(v) := \mathbf{E}_1(v) \sqcup_{\mathbf{E}} \mathbf{E}_2(v) \quad \mathbf{B}(o) := \mathbf{B}_1(o) \sqcup_{\mathbf{B}} \mathbf{B}_2(o) \quad \mathbf{R}(o) := \mathbf{R}_1(o) \sqcup_{\mathbf{R}} \mathbf{R}_2(o)$$

$$\mathbf{C}(v) := \mathbf{C}_1(v) \sqcup_{\mathbf{C}} \mathbf{C}_2(v) \quad \mathbf{U}(o) := \mathbf{U}_1(o) \sqcup_{\mathbf{U}} \mathbf{U}_2(o)$$

Particularly, we define $\sqcup_{\mathbf{E}}, \sqcup_{\mathbf{C}}$*, and* $\sqcup_{\mathbf{U}}$ *as follows:*

$$\mathbf{E}_1(v) \sqcup_{\mathbf{E}} \mathbf{E}_2(v) := \{(o, \; \phi_1 \vee \phi_2) \mid (o, \phi_1) \in \mathbf{E}_1(v), \; (o, \phi_2) \in \mathbf{E}_2(v)\}$$

$$\mathbf{B}_1(o) \sqcup_{\mathbf{B}} \mathbf{B}_2(o) := \{((o_k, \phi_k^1 \vee \phi_k^2), (o_v, \phi_v^1 \vee \phi_v^2)) \mid ((o_k, \phi_k^i), (o_v, \phi_v^i)) \in \mathbf{B}_i(o), \; i \in \{1, 2\}\}$$

$$\mathbf{R}_1(o) \sqcup_{\mathbf{R}} \mathbf{R}_2(o) := \{(t,\ \phi_1 \vee \phi_2) \mid (t, \phi_1) \in \mathbf{R}_1(o),\ (t, \phi_2) \in \mathbf{R}_2(o),\ t \in \cup_{k=0}^{N} \mathbf{O}^k\}$$
$$\cup \{(t,\ \phi_1 \vee \phi_2) \mid (t, \phi_1) \in \mathbf{R}_1(o),\ (t, \phi_2) \in \mathbf{R}_2(o),\ t \in \mathbf{O}_c \times \mathbf{O}\}$$

$$\mathbf{C}_1(v) \sqcup_{\mathbf{C}} \mathbf{C}_2(v) := \textit{ite}(\mathbf{C}_1(v) = \mathbf{C}_2(v), \mathbf{C}_1(v), \top)$$

$$\mathbf{U}_1(o) \sqcup_{\mathbf{U}} \mathbf{U}_2(o) := \textit{ite}(\mathbf{U}_1(o) = \mathbf{U}_2(o), \mathbf{U}_1(o), 0)$$

Specifically, ***ite*** *is the shorthand of* if-then-else *expression. For clarity, we assume that* $(o, F) \in \mathbf{E}(v)$ *if there does not exist* ϕ *such that* $(o, \phi) \in \mathbf{E}(v)$*. For other subdomains, we also use the similar assumptions to make the definitions compact.*

Definition 4.5.4 (Partial Order of M) $\mathbf{M}_1 \sqsubseteq_{\mathbf{M}} \mathbf{M}_2$ *if and only if there exists* $\mathbf{M}'$ *such that* $\mathbf{M}_1 \sqcup_{\mathbf{M}} \mathbf{M}' = \mathbf{M}_2$*.* $\sqsubseteq_{\mathbf{M}}$ *also naturally exports the definitions of* $\sqsubseteq_{\mathbf{E}}$*,* $\sqsubseteq_{\mathbf{L}}$*,* $\sqsubseteq_{\mathbf{C}}$*, and* $\sqsubseteq_{\mathbf{U}}$*.*

Intuitively, a join operator is a set union with a disjunction of path constraints. The complexity of a join operator of each subdomain is $O(|D_1| + |D_2|)$, because we can maintain D_1 and D_2 by sorted lists, and a join operator takes linear time. Further, we naturally extend the join operators $\sqcup_{\mathbf{R}}$ and $\sqcup_{\mathbf{E}}$ to the batch join operators $\widetilde{\sqcup}_{\mathbf{R}}$ and $\widetilde{\sqcup}_{\mathbf{E}}$ in order to join multiple elements, which are applied in the rules in Section 4.6.3 and Section 4.6.5.

4.5.3 Layout Operator for Strong Update

According to the definition of **L**, its second subdomain **R** abstracts the index-value correlations of container objects, i.e., how objects are associated with the indexes. To support strong updates, we define the layout operators to describe the semantics of inserting, accessing, and removing the objects, which are applied to update the abstract memory in Section 4.6.

Definition 4.5.5 (Layout Operator) *Given* $o_l \in \mathbf{O}_p$*,* $o_m \in \mathbf{O}_v$*,* $o_i, o_k \in \mathbf{O}_c$ *and* $o_v \in \mathbf{O}$*, a layout operator has one of the following forms.*

- $\mu(\mathbf{R}, o_l, o_i, o_v)$ *and* $\mu(\mathbf{R}, o_m, o_k, o_v)$ *insert an single object and a pair of objects to* o_l *and* o_m *at a deterministic position* o_i *and key* o_k*, respectively, producing the new container memory layouts after the insertions.

- $\pi(\mathbf{R}, o_l, o_i)$ *and* $\pi(\mathbf{R}, o_m, o_k)$ *collect the object at a deterministic position* o_i *and key* o_k*, respectively, returning a set of abstract objects paired with constraints, which indicate accessed elements and conditions.*

- $\omega(\mathbf{R}, o_l, o_i)$ *and* $\omega(\mathbf{R}, o_m, o_k)$ *remove the single object and the pair of objects at a deterministic position* o_i *and key* o_k *from* o_l *and* o_m*, respectively, producing the new container memory layouts after the removals.*

Any operation on an anchored container is essentially a layout operator, which is a function returning a new state in the subdomain $\mathbf{R}$, corresponding to the index-value correlations after the operation. Due to the finite size of $\mathbf{O}_c$, the complexity of a layout operator is bounded by $|\mathbf{R}(o)| \cdot |\mathbf{O}_c|$. Although $|\mathbf{R}(o)|$ depends on the numbers of the insertions and removals in different branches, computing layout operators is still more lightweighted than solving complex constraints qualifying positions and keys with a sheer number of disjunctions.

Example 4.5.2 *In Figure 4.2, we have* $\mathbf{R}(o_1) = \varnothing$ *before line 4. The insertion at line 4 induces* $\mu(\mathbf{R}, o_1, \text{“id”}, \text{“a”})$*, which updates* $\mathbf{R}(o_1)$ *to* $\{((\text{“id”}, \text{“a”}), \mathsf{T})\}$*.*

4.5.4 Summary

According to the abstract states, we can determine the indirect value flows induced by the container method calls. Specifically, the points-to facts in the abstract environment $\mathbf{E}$ provide sufficient information of adding value-flow edges in the value-flow graph, based on which the client of value-flow analysis can be performed.

Example 4.5.3 *Before line 7 in the program shown in Figure 4.2, we have*

$$\mathbf{R}(o_1) = \{((\text{“id”}, \text{“a”}), \mathsf{T}), ((\text{“age”}, \mathtt{null}), \mathsf{T})\}$$

After line 7, we can obtain $\mathbf{E}(i) = \{(\text{“a”}, \mathsf{T})\}$*. Finally, we add an edge from* $\text{“a”}@\ell_4$ *to* $i@\ell_7$ *to the VFG in Figure 4.3.*

Technical Challenges. Based on the abstract memory, we have to compute the abstract state for each program location. Specifically, we should resolve the following issues:

- We should identify the manipulated memory objects precisely for container method calls, posing the challenge in updating points-to facts in **E**.
- A container object may be modified and accessed at dozens of locations in the program and form different memory layouts, making it challenging to maintain **L** precisely and efficiently.
- The facts in the subdomains have sophisticated interactions, which requires a non-trivial semantic reduction operator [61]. For example, the update of **L** should be aware of **U** to determine whether strong updates should be applied.

4.6 Memory Orientation Analysis

The section presents the memory orientation analysis that addresses the technical challenges discussed in Section 4.4. We first present the abstract transformers of operations not related to containers and show how to maintain precise points-to facts (Section 4.6.1). We then define the partial abstract transformers of container method calls (Section 4.6.2), and propose a semantic reduction operator, i.e., *witness operator*, to obtain a more precise abstract state by applying strong updates to anchored containers (Section 4.6.3). Based on the partial abstract transformers and witness operators, we depcit the abstract semantics of a container method call (Section 4.6.4). We also define the abstract semantics of container traversals briefly (Section 4.6.5). Finally, we illustrate how the memory orientation analysis integrates the reasoning of container semantics into the VFG construction (Section 4.6.6). The benefits of the memory orientation analysis are further highlighted along with the formulation and the proof of the soundness (Section 4.6.7).

In what follows, we describe the abstract transformers as deductive rules of the form:

$$\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U} \vdash \ \mathtt{st} : \mathbf{E}', \mathbf{L}', \mathbf{C}', \mathbf{U}'$$

where $(\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U})$ and $(\mathbf{E}', \mathbf{L}', \mathbf{C}', \mathbf{U}')$ are the abstract memory states before and after the statement $\mathtt{st}$, respectively.

$$\text{T-Alloc}\ \frac{\mathbf{E}' = \mathbf{E}[v \to \{(o, \top)\}] \quad \mathbf{C}' = \mathbf{C}[v \to \top]}{\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U} \vdash v = \textbf{new}\ \ \tau : \mathbf{E}', \mathbf{L}, \mathbf{C}', \mathbf{U}}$$

$$\text{T-Ass-var}\ \frac{\mathbf{E}' = \mathbf{E}[v \to \mathbf{E}(u)] \quad \mathbf{C}' = \mathbf{C}[v \to \mathbf{C}(u)]}{\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U} \vdash v = u : \mathbf{E}', \mathbf{L}, \mathbf{C}', \mathbf{U}}$$

$$\text{T-Ass-lit}\ \frac{\mathbf{E}(a) = \{(o_a, \phi)\} \quad \mathbf{E}' = \mathbf{E}[v \to \{(o_a, \phi)\}] \quad \mathbf{C}' = \mathbf{C}[v \to o_a]}{\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U} \vdash v = a : \mathbf{E}', \mathbf{L}, \mathbf{C}', \mathbf{U}}$$

Figure 4.7: Abstract transformer of allocation and assignment

$$\text{T-Sequencing}\ \frac{\mathbf{M} \vdash S_1 : \mathbf{M}_0 \quad \mathbf{M}_0 \vdash S_2 : \mathbf{M}'}{\mathbf{M} \vdash S_1; S_2 : \mathbf{M}'}$$

$$\text{T-Branch}\ \frac{\begin{array}{c}\phi = \bigvee_{(o,\phi) \in \mathbf{E}(v)} (o \neq \textit{null} \wedge \phi) \\ \mathbf{M} \wedge \phi \vdash S_1 : \mathbf{M}_1 \quad \mathbf{M} \wedge (\neg\phi) \vdash S_2 : \mathbf{M}_2 \quad \mathbf{M}' = \mathbf{M}_1 \sqcup_{\mathbf{M}} \mathbf{M}_2\end{array}}{\mathbf{M} \vdash \textbf{if}\ (v)\ \textbf{then}\ S_1\ \textbf{else}\ S_2 : \mathbf{M}'}$$

Figure 4.8: Abstract transformer of sequencing and branch

4.6.1 Abstract Semantics of Non-Container Operation

Non-container operations include allocation statements, assignments, sequencing, and branches. We define their abstract transformers as follows.

Allocation Statement and Assignment. Figure 4.7 presents the abstract transformers for allocation statements and assignments. The rules are straightforward. For instance, when a memory object is allocated, we create an abstract object $o \in \mathbf{O}$. To maintain the points-to relation between v and o, we apply the strong update to the abstract environment $\mathbf{E}$ by setting $\mathbf{E}(v)$ to $\{(o, \top)\}$, indicating v must point to o after the statement. Meanwhile, v can not be a variable pointing to a literal. Thus $\mathbf{C}(v)$ is set to $\top$.

The assignment $v = u$ applies the strong update to the points-to set of v by setting $\mathbf{E}(v)$ to $\mathbf{E}(u)$, and the abstract values of v to the ones of u. Similarly, we define the abstract transformer for a literal assignment.

Sequencing and Branch. In Figure 4.8, the rule T-Sequencing defines the abstract transformer of a sequence of statements, which is the composition of the abstract transformer

of each statement. Similarly, the rule T-Branch defines the abstract transformer of a branch statement. It first conjoins the branch conditions ϕ and $\neg\phi$ with every constraint in the abstract states respectively, and then transforms the abstract states along two branches. Finally, it joins the abstract states at the end of two branches and obtains the abstract state after the branch statement. Particularly, we use the notation $\mathbf{M} \wedge \phi$ as a shorthand of conjoining ϕ with the constraints in $\mathbf{M}$.

4.6.2 Partial Abstract Transformer of Container Method Call

We proceed to depict the abstract semantics for container method calls. The overall idea is to locate the abstract container objects manipulated by the method call based on the points-to facts in $\mathbf{E}$ and then recompute $\mathbf{L}$. Because we can not determine whether the manipulated container object is an anchored container or not from $\mathbf{U}$ before the statement, we postpone the strong updates on the memory layouts of anchored containers by an additional operator (Section 4.6.3), and define a partial abstract transformer first without analyzing index-value correlations of the manipulated containers. For clarity, we first define the unary operator $\bigtriangledown_{\mathbf{R}}$ to construct the upper bound of the abstract layout of a container object.

Definition 4.6.1 (Upper-Bound Operator) *Given* $o \in \mathbf{O}_p \cup \mathbf{O}_v$ *and* $\mathbf{R}$*, we define the upper-bound operator* $\bigtriangledown_{\mathbf{R}}$*:*

$$\bigtriangledown_{\mathbf{R}}(\mathbf{R}, o) := \{(t, \bigvee_{(t',\phi)\in\mathbf{R}(o)} \phi) \mid t \in \mathcal{T}\}$$

where $\mathcal{T} := \cup_{k=0}^{N} \mathbf{O}^k$ *when* $o \in \mathbf{O}_p$ *and* $\mathcal{T} := \mathbf{O}_c \times \mathbf{O}$ *when* $o \in \mathbf{O}_v$*. Note that* $\bigtriangledown_{\mathbf{R}}(\mathbf{R}, o)$ *is the conceptual upper bound of the abstract layout of* o*. We use this notation for defining the rules while do not compute it explicitly.*

We describe the partial abstract transformers of container method calls in the following deductive form:

$$\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U} \vdash_t st : \mathbf{E}', \mathbf{L}', \mathbf{C}', \mathbf{U}'$$

$$\text{T-Ins}\ \frac{\begin{array}{c}(o_c, \phi_c) \in \mathbf{E}(c) \quad (o_u, \phi_u) \in \mathbf{E}(u) \quad (o_v, \phi_v) \in \mathbf{E}(v) \\ B = (\{(o_u, \phi_c \wedge \phi_u)\}, \{(o_v, \phi_c \wedge \phi_v)\}) \\ \mathbf{B}' = \mathbf{B}[o_c \to \mathbf{B}(o_c) \sqcup_{\mathbf{B}} B] \quad \mathbf{R}' = \mathbf{R}[o_c \to \bigtriangledown_{\mathbf{R}}(\mathbf{R}, o_c)] \\ \mathbf{U}' = \mathbf{U}[o_c \to \textbf{ite}(\mathbf{C}(o_u) \in \mathbf{O}_c,\ \mathbf{U}(o_c),\ 0)]\end{array}}{\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U} \vdash_t c.\textbf{insert}(u, v) : \mathbf{E}, \mathbf{L}', \mathbf{C}, \mathbf{U}'}$$

$$\text{T-Rem}\ \frac{\begin{array}{c}(o_c, \phi_c) \in \mathbf{E}(c) \quad (o_u, \phi_u) \in \mathbf{E}(u) \\ \mathbf{R}' = \mathbf{R}[o_c \to \bigtriangledown_{\mathbf{R}}(\mathbf{R}, o_c)] \quad \mathbf{U}' = \mathbf{U}[o_c \to \textbf{ite}(\mathbf{C}(o_u) \in \mathbf{O}_c,\ \mathbf{U}(o_c),\ 0)]\end{array}}{\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U} \vdash_t c.\textbf{remove}(u) : \mathbf{E}, \mathbf{L}', \mathbf{C}, \mathbf{U}'}$$

$$\text{T-Acc}\ \frac{\begin{array}{c}(o_c, \phi_c) \in \mathbf{E}(c) \quad \mathbf{B}(o_c) = (B_u, B_v) \\ \mathbf{C}' = \mathbf{C}[v \to \top] \quad \mathbf{E}' = \mathbf{E}[v \to \{(o_e, \phi_c \wedge \phi_e) \mid (o_e, \phi_e) \in B_v\}]\end{array}}{\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U} \vdash_t v = c.\textbf{access}(u) : \mathbf{E}', \mathbf{L}, \mathbf{C}', \mathbf{U}}$$

Figure 4.9: Partial abstract transformers of container method call

Figure 4.9 presents the partial abstract transformers for container method calls. First, the rule T-Ins defines the partial abstract transformer of an insertion. For each abstract objects pointed by c, u, and v, it transforms the abstract memory state in three steps.

- For an abstract container object o_c pointed by c, the rule updates its abstract value in $\mathbf{B}$ by inserting o_u and o_v with the constraint $\phi_c \wedge \phi_u$.
- To update $\mathbf{R}$, T-Ins simply sets the abstract value of o_c to its conceptual upper bound to obtain a sound approximation by the upper-bound operator in Definition 4.6.1.
- The abstract value of o_c in the uniqueness domain is also updated and set to 0 if the abstract value of o_u in the constant domain is not literal. Otherwise, it is preserved and equal to the original abstract value.

Similarly, we define the rule T-Rem for a removal. The difference is that T-Rem does not change $\mathbf{B}$ with the assumption that no element is removed to guarantee the soundness.

Lastly, T-Acc defines the partial abstract transformer of an access method call. $\mathbf{C}(v)$ is set to be $\top$, as v is not assigned by a literal. Meanwhile, T-Acc iterates each abstract container object o_c pointed by c and the abstract objects o_e in its value set, and o_e is exactly the accessed object. Finally, T-Acc applies the strong update to $\mathbf{E}(v)$ by enforcing v point to o_e under $\phi_c \wedge \phi_e$, where ϕ_c and ϕ_e are the paired constraints of o_c and o_e, respectively.

Example 4.6.1 *Before line 5 in Figure 4.2, we have* $\mathbf{U}(o_1) = 1$, $\mathbf{R}(o_1) = \{(\text{“id”}, \text{“a”}), \mathsf{T})\}$, $\mathbf{B}(o_1) = (\{(\text{“id”}, \mathsf{T})\}, \{(\text{“a”}, \mathsf{T})\})$, *and* $\mathbf{C}(ind_2) = \text{“age”} \in \mathbf{O}_c$, *where* ind_2 *is the insertion index, so we have* $\mathbf{U}'(o_1) = 1$ *after line 5. Applying the rule* T-Ins, *we have*

$$\mathbf{B}'(o_1) = (\{(\text{“id”}, \mathsf{T}), (\text{“age”}, \mathsf{T})\}, \{(\text{“a”}, \mathsf{T}), (null, \mathsf{T})\})$$

$$\mathbf{R}'(o_1) = \bigtriangledown_{\mathbf{R}}(\mathbf{R}, o_1) = \{(t, \mathsf{T}) \mid t \in \mathbf{O}_c \times \mathbf{O}\}$$

As defined in the rule T-Ins, the values of the manipulated container object in $\mathbf{B}$ are merged by $\sqcup_{\mathbf{B}}$. According to Definition 4.5.3, the disjunctions in constraints enable the strong update on $\mathbf{B}$. However, while the rules support precise reasoning about $\mathbf{B}$, we can not obtain how objects are stored in each container because $\mathbf{C}$ and $\mathbf{U}$ are opaque to $\mathbf{R}$.

4.6.3 Witness Operator

To utilize anchored containers and obtain more precise memory layouts, we instantiate the witness operators [151] for semantic reduction [61]. Based on Definition 4.5.2 in Section 4.5.1, $\mathbf{U}'$ determines all the anchored containers at the statement st. Therefore, the key idea underlying the witness operators is to sharpen $\mathbf{R}$ as long as we find that an abstract container object is anchored (according to the subdomain $\mathbf{U}$). Specifically, the witness operator takes as input the abstract memory states $\mathbf{M} := (\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U})$ before the statement and $\mathbf{M}' := (\mathbf{E}', \mathbf{L}', \mathbf{C}', \mathbf{U}')$ after applying the partial abstract transformer, and finally returns a smaller abstract memory state $\mathbf{M}'' := (\mathbf{E}'', \mathbf{L}'', \mathbf{C}'', \mathbf{U}'')$ compared with $\mathbf{M}'$. We describe the witness operator as deductive rules of the following form:

$$(\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U}), (\mathbf{E}', \mathbf{L}', \mathbf{C}', \mathbf{U}') \vdash_w st : (\mathbf{E}'', \mathbf{L}'', \mathbf{C}'', \mathbf{U}'')$$

Figure 4.10 presents the witness operators for container method calls. With the facts in $\mathbf{C}$ and $\mathbf{U}$, it is possible to transform $\mathbf{R}$ to smaller states than $\mathbf{R}'$ in an insertion and removal, and obtain a smaller state $\mathbf{E}''$ than $\mathbf{E}'$ in an access method call.

- For an insertion, the rule W-Ins first examines whether u points to a literal and the abstract container object o_c pointed by c is deterministic. If o_c is an anchored container, the layout operator μ updates its memory layout maintained by $\mathbf{R}$. Because

$$\text{W-Ins}\ \frac{\begin{array}{c}\mathbf{E}(c) = \{(o_c, \phi_c)\} \quad (o_v, \phi_v) \in \mathbf{E}(v) \quad \mathbf{C}(u) = o_u \\ A(o_v, \phi_v) = \widetilde{\sqcup}_{\mathbf{R}}\{(t, \phi \wedge \phi_v) \mid (t, \phi) \in \mu(\mathbf{R}, o_c, o_u, o_v)\} \\ \mathbf{R}'' = \mathbf{R}'[o_c \to \widetilde{\sqcup}_{\mathbf{R}}\{A(o_v, \phi_v) \mid (o_v, \phi_v) \in \mathbf{E}(v)\}] \\ \mathbf{U}'(o_c) = 1 \quad \mathbf{L}'' = (\mathbf{B}', \mathbf{R}'') \quad \mathbf{M}'' = (\mathbf{E}', \mathbf{L}'', \mathbf{C}', \mathbf{U}')\end{array}}{\mathbf{M}, \mathbf{M}' \vdash_w c.\mathbf{insert}(u, v) : \mathbf{M}''}$$

$$\text{W-Rem}\ \frac{\begin{array}{c}\mathbf{E}(c) = \{(o_c, \phi_c)\} \quad \mathbf{C}(u) = o_u \quad \mathbf{R}'' = \mathbf{R}'[o_c \to \omega(\mathbf{R}, o_c, o_u)] \\ \mathbf{U}'(o_c) = 1 \quad \mathbf{L}'' = (\mathbf{B}', \mathbf{R}'') \quad \mathbf{M}'' = (\mathbf{E}', \mathbf{L}'', \mathbf{C}', \mathbf{U}')\end{array}}{\mathbf{M}, \mathbf{M}' \vdash_w c.\mathbf{remove}(u) : \mathbf{M}''}$$

$$\text{W-Acc}\ \frac{\begin{array}{c}\mathbf{E}(c) = \{(o_c, \phi_c)\} \quad \mathbf{C}(u) = o_u \quad E = \{(o_e, \phi_c \wedge \phi_e) \mid (o_e, \phi_e) \in \pi(\mathbf{R}, o_c, o_u)\} \\ \mathbf{U}'(o_c) = 1 \quad \mathbf{E}'' = \mathbf{E}[v \to E] \quad \mathbf{M}'' = (\mathbf{E}'', \mathbf{L}', \mathbf{C}', \mathbf{U}')\end{array}}{\mathbf{M}, \mathbf{M}' \vdash_w v = c.\mathbf{access}(u) : \mathbf{M}''}$$

Figure 4.10: Witness operator of container method call

the number of the abstract memory objects pointed by v can be larger than 1, all the updated abstract layouts $\mathbf{R}$ of o_c are joined by $\widetilde{\sqcup}_{\mathbf{R}}$. Similarly, the rule W-Rem applies the strong update to $\mathbf{R}$ and removes the stored objects from the memory layout of an anchored container by ω.

- The rules of the witness operators for access method calls are straightforward. If u points to a literal and c points to an anchored container, W-Acc utilizes the layout operator π to collect the memory objects o_e paired with the index o_u in $\mathbf{R}$ and then applies the strong update to $\mathbf{E}(v)$ by enforcing v point to o_e.

Example 4.6.2 *After applying the rule T-Ins for the insertion at line 5 in Figure 4.2, we have* $\mathbf{U}'(o_1) = 1$, *indicating that* o_1 *is an anchored container. Thus, we have*

$$\begin{aligned}\mathbf{R}''(o_1) &= \widetilde{\sqcup}_{\mathbf{R}}\{(t, \phi \wedge \mathsf{T}) \mid (t, \phi) \in \mu(\mathbf{R}, o_1, \text{“age”}, \mathtt{null})\} \\ &= \{((\text{“id”}, \text{“a”}), \mathsf{T}), ((\text{“age”}, \mathtt{null}), \mathsf{T})\}\end{aligned}$$

Generally, a semantic reduction operator in the combined domain can be expensive, as it requires pair-wise or clique-wise operators [61, 152]. We notice that join operators are applied multiple times in the rule W-Ins, iterating the k-tuples or pairs of abstract objects in $\mu(\mathbf{R}, o_c, o_u, o_v)$ and computing the constraint of each k-tuple or pair. A naive design has quadratic time complexity in the total number of k-tuples or pairs. We optimize witness

operators by maintaining a hash value for each k-tuple and pair so that the join operator can be applied in linear time complexity. Specifically, if (t_1, ϕ_1) and (t_2, ϕ_2) satisfy the condition that t_1 and t_2 have the same hash value, we are aware that t_1 and t_2 are equal and then create the disjunction $\phi_1 \vee \phi_2$ paired with t_1 (i.e., t_2).

4.6.4 Abstract Semantics of Container Method Call

For each container method call, we can easily compute its abstract semantics by applying witness operators after the partial abstract transformers. Specifically, we have the abstract transformer $\mathbf{M} \vdash st : \mathbf{M}'$ for a container method call st if and only if $\mathbf{M} \vdash_t st : \mathbf{M}_0$ and $\mathbf{M}, \mathbf{M}_0 \vdash_w st : \mathbf{M}'$.

In contrast to the partial abstract transformers in Figure 4.9, the witness operators demand the post-states of the subdomains after applying the partial abstract transformers, such as $\mathbf{U}'$, to identify anchored containers for strong updates. However, the partial abstract transformers in Figure 4.9 induces the transition of each subdomain independently, permitting the parallel updates of the subdomains. We separate the witness operators from the partial abstract transformers to achieve better efficiency in the transition.

It is worth noting that witness operators only affect the abstract memory states when the container method calls manipulate an anchored container at a deterministic index. For example, we can safely skip other computations of witness operators if u does not point to a literal. Although we compose witness operators with partial abstract transformers eagerly, we can end unnecessary witness operators by scheduling the computations of the premises of the rules in Figure 4.10.

With the benefit of witness operators, we can obtain more precise container memory layouts by achieving the semantic reduction. We state the correctness of the witness operators in Figure 4.10 as the following theorem.

Theorem 4.6.1 *Given a container method call* st *and an abstract memory state* $\mathbf{M}$ *before* st*, we have*

$$\mathbf{M} \vdash_t st : \mathbf{M}' \ \wedge \ \mathbf{M}, \mathbf{M}' \vdash_w st : \mathbf{M}'' \ \Rightarrow \ \mathbf{M}'' \sqsubseteq_{\mathbf{M}} \mathbf{M}'$$

Proof. We sketch the proof for the case in which st is an insertion. Without the loss of

generality, we only prove the correctness of W-Ins for position-dependent containers. For value-dependent containers, the proofs can be provided in almost the same way. Also, we can construct the similar proofs for the other two kinds of container method calls.

According to the definitions of W-Ins in Figure 4.10, we only need to prove that $\mathbf{L}'' \sqsubseteq_{\mathbf{L}} \mathbf{L}'$. Based on the definition of W-Ins, we have $\mathbf{L}'' = (\mathbf{B}', \mathbf{R}'')$. Hence, we need to prove that for an arbitrary $o_p \in \mathbf{O}_p$ and $((o_1^1, \ldots, o_1^k), \phi_1) \in \mathbf{R}''(o_p)$, there exists $((o_2^1, \ldots, o_2^k), \phi_2) \in \mathbf{R}'(o_p)$ such that $o_1^i = o_2^i (1 \leqslant i \leqslant k)$ and ϕ_1 implies ϕ_2. Based on the definition of T-Ins and W-Ins, $\mathbf{R}''$ and $\mathbf{R}'$ only differ at o_c pointed by c, where $\mathbf{E}(c) = \{(o_c, \phi_c)\}$. We have

$$A(o_v, \phi_v) = \widetilde{\sqcup}_{\mathbf{R}}\{(t, \phi \wedge \phi_v) \mid (t, \phi) \in \mu(\mathbf{R}, o_c, o_u, o_v)\}$$
$$\mathbf{R}'' = \mathbf{R}'[o_c \rightarrow \widetilde{\sqcup}_{\mathbf{R}}\{A(o_v, \phi_v) \mid (o_v, \phi_v) \in \mathbf{E}(v)\}]$$

Consider an arbitrary $((o_1, o_2, \ldots, o_m), \phi'') \in \mathbf{R}''(o_c)$, we can derive the following fact from the definition of μ:

$$\exists((o_1, o_2, \ldots, o_{o_i - 1}, o_{o_i + 1}, \ldots, o_m), \phi) \in \mathbf{R}(o_c), \quad \phi \wedge \phi_v = \phi''$$

Based on the definition of $\bigtriangledown_{\mathbf{R}}$, we have $\mathbf{R}'(o_c) = \{(t, \bigvee_{(t', \phi) \in \mathbf{R}(o_c)} \phi) \mid t \in \cup_{k=0}^{N} \mathbf{O}^k\}$. Let $t = (o_1, o_2, \ldots, o_m)$, and $\phi' = \bigvee_{(t', \phi) \in \mathbf{R}(o_c)} \phi$. We have ϕ implies ϕ' according to the property of the logical disjunction. Meanwhile, we have ϕ'' implies ϕ based on $\phi \wedge \phi_v = \phi''$, so we get ϕ'' implies ϕ'. Thus, we have proved that for each $((o_1, o_2, \ldots, o_m), \phi'') \in \mathbf{R}''(o_c)$, there exist $((o_1, o_2, \ldots, o_m), \phi') \in \mathbf{R}'(o_c)$ such that ϕ'' implies ϕ'. Q.E.D.

Remark. The precision enhancement provided by witness operators also propagates to non-anchored containers because of the interactions among the subdomains **E**, **B**, and **R**. With the benefit of precise index-value correlations in **R**, the witness operators in Figure 4.10 also generate more precise points-to facts in **E**, based on which the transformers in Figure 4.9 produce more precise ownership in **B** for general containers.

Another interesting benefit of witness operators is that the objects with fields can be precisely analyzed in a field sensitive manner. Essentially, such an object is a particular kind of an anchored container. The constant indexes of the manipulations are exactly the field names of the object. In real-world programs, container objects are often used as the fields of a user-defined object. A field-insensitive analysis can hardly apply strong updates upon container memory layouts, as it does not identify which container objects

$$\textsf{Fix}\frac{\mathbf{M} \vdash S : \mathbf{M}_1 \quad \mathbf{M} = \mathbf{M}_1}{\mathbf{M} \vdash fix(S) : \mathbf{M}} \qquad \textsf{Fix}\frac{\mathbf{M} \vdash S : \mathbf{M}_1 \quad \mathbf{M} \neq \mathbf{M}_1 \quad (\mathbf{M} \bigtriangledown \mathbf{M}_1) \vdash fix(S) : \mathbf{M}'}{\mathbf{M} \vdash fix(S) : \mathbf{M}'}$$

$$\textsf{T-Loop}\frac{\begin{array}{c}(o_c, \phi_c) \in \mathbf{E}(c) \quad (B_u, B_v) = \mathbf{B}(o_c) \\ E_u = \widetilde{\sqcup}_{\mathbf{E}}\{(o_u, \phi \wedge \phi_c) \mid (o_u, \phi) \in B_u\} \quad E_v = \widetilde{\sqcup}_{\mathbf{E}}\{(o_v, \phi \wedge \phi_c) \mid (o_v, \phi) \in B_v\} \\ \mathbf{E}_1 = \mathbf{E}[u \to E_u, v \to E_v] \quad \mathbf{C}_1 = \mathbf{C}[u \to \top, v \to \top] \quad (\mathbf{E}_1, \mathbf{L}, \mathbf{C}_1, \mathbf{U}) \vdash fix(S) : \mathbf{M}'\end{array}}{\mathbf{M} \vdash \textbf{foreach } (u, v) \textbf{ in } c \textbf{ do } S \textbf{ od} : \mathbf{M}'}$$

Figure 4.11: Abstract transformer of container traversal

are pointed by the field precisely. Therefore, the witness operators can compute the precise points-to facts for each field, which further enables strong updates for the container objects pointed by the fields.

4.6.5 Semantics of Container Traversal

For a traversal, the rules in Figure 4.11 iterate the loop body to obtain the fixed point. For example, the rule T-Loop first enumerates the objects stored in a container and enforces u and v point to these objects. Besides, u and v can not point to certain literals, so $\mathbf{C}(u)$ and $\mathbf{C}(v)$ are set to $\top$. The rule Fix finally computes the fixed point for the loop body.

To assure the termination, we define and apply the widening operator $\bigtriangledown$ [153] if the abstract state $\mathbf{M}$ before an iteration is not equal to the abstract state $\mathbf{M}_1$ after the iteration. The path constraint after widening is set to be *true* if it is changed in the iteration. The finite sizes of $\mathbf{X}$ and $\mathbf{O}$ guarantee that $\mathbf{M}_1 = \mathbf{M}$ must hold after applying the rule Fix finite times, and the rule T-Loop must be terminating.

4.6.6 Value-Flow Graph Construction

Based on the points-to facts in $\mathbf{E}$, we can identify the accessed container object, add the value-flow edges labeled with constraints, and finally, stitch value flows from different functions by function summaries to construct a global VFG. Compared with previous value flow analyses [3, 115, 154], the memory orientation analysis extends the abstract memory model to analyze the semantics of container method calls, finally discovering the value flows through containers precisely.

Table 4.1: Rules of computing value-flow edges

Statement	Condition	Edge
$v = a;$	$\exists o_a,\ (o_a, \phi_a) \in \mathbf{E}(a) \wedge (o_a, \phi_v) \in \mathbf{E}(v)$	$a \to v$: $\phi_a \wedge \phi_v$
$v = u;$	$\exists o,\ (o, \phi_v) \in \mathbf{E}(v) \wedge (o, \phi_u) \in \mathbf{E}(u)$	$u \to v$: $\phi_v \wedge \phi_u$
$v = c.access(u)$	$\exists o \exists w,\ (o, \phi_v) \in \mathbf{E}(v) \wedge (o, \phi_w) \in \mathbf{E}(w)$	$w \to v$: $\phi_v \wedge \phi_w$

Following the existing approaches [3, 115], the memory orientation analysis computes the abstract memory and constructs the VFG in two phases as follows.

Intraprocedural Analysis. We construct the VFG for a function straightforwardly based on the rules in Table 4.1. For an assignment, we check the points-to sets of the variables on the left and right-hand sides and add the edge if they are not disjoint. For an access method call, a value-flow edge is produced between w and v when their points-to sets are not disjoint after the statement. The guarded constraints of the edges are exactly the conjunctions of the constraints of the points-to facts in $\mathbf{E}$. Particularly, we do not invoke SMT solvers on the constraints but store them in the graph as an edge label. To simplify the constraints, we follow the previous work [3] and utilize lightweight semi-decision procedures to filter out apparent contradictions.

Interprocedural Analysis. In the presence of function calls, we introduce the abstract objects for the formal parameters at the function entry and the formal return values at the function exit. Following existing techniques [1, 2, 3], we also add auxiliary parameters and return values to support depicting side effects. Then, we compute the abstract memory in the intraprocedural analysis and build the function summary according to the abstract memory [63], which are the edges between the parameters to the returns in the VFG of a single function, abstracting the effect of calling the function [155]. Finally, we inline the function summary of the callee at each call site located in the caller function in a bottom-up manner, yielding a global VFG of the whole program.

Example 4.6.3 *For the function* bar *in Figure 4.2, the last element of* o_3 *is accessed at line 18, so we add an auxiliary parameter node* ids_arg1 *to the VFG in Figure 4.3. Based on the abstract state before line 14, we can determine that* $i@\ell_7$ *and* $j@\ell_{11}$ *can be the last element, inducing two edges from* $i@\ell_7$ *and* $j@\ell_{11}$ *to* ids_arg1*, respectively. Similarly, we add the other two auxiliary parameter nodes, i.e.,* hs_arg1 *and* hs_arg2*, and connect them with* $null@\ell_5$ *and* "a"$@\ell_4$*, respectively,*

to form the interprocedural value flow.

4.6.7 Discussion

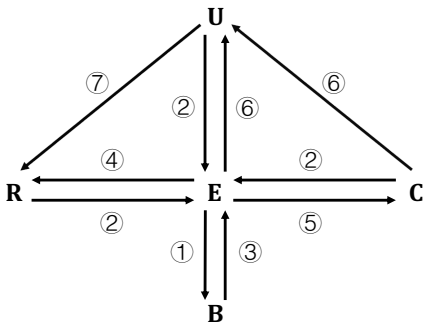
Benefit of the Combined Domain. Our memory orientation rests on the combined domain to apply strong updates to the memory layouts of anchored containers, allowing a more precise solution than the one obtained by solving each subdomain separately. Essentially, **C** supports the constant propagation [48] and affects the state transition in the uniqueness domain **U**. Even if an index is computed at runtime, we can still effectively identify whether it is constant or not. Furthermore, **E** and **U** enable us to introduce the witness operators for semantic reduction [61] and obtain more precise abstract states.

Figure 4.12 shows the interactions between the subdomains, which are indicated by the edges labeled with the sets of the rules. In total, there are ten edges in Figure 4.12 showing ten ways of interactions of the subdomains. We discuss two typical interactions as follows.

- The edge from **U** to **R** indicates the precision benefit introduced by the rules W-Ins and W-Rem. They apply the strong updates upon the memory layouts of the anchored containers and compute their precise index-value correlations in **R**.
- The edge from **E** to **B** indicates that we can obtain more precise ownership even if the container is not an anchored container, as the rule T-Ins propagates the precision benefit from **E** to **B**.

Overall, the combined domain and its abstract transformers serve as a critical role in the memory orientation analysis, eventually promoting the precision of value flow analysis.

Soundness of the Memory Orientation Analysis. Lastly, we discuss the soundness of the memory orientation analysis. An anchored container is essentially a data structure with a set of fields. According to the partial abstract transformers in Figure 4.9, the uniqueness domain **U** indicates whether a container object must be an anchored container or not. This implies that we perform strong updates conservatively, as we under-approximate the set of anchored containers. As long as we identify anchored containers, we finally rea-



Rule Set ID	Rules
①	{T-Ins}
②	{W-Acc}
③	{T-Acc}
④	{T-Ins, T-Rem, W-Ins, W-Rem}
⑤	{T-Ass-lit}
⑥	{T-Ins, T-Rem}
⑦	{W-Ins, W-Rem}

Figure 4.12: The interactions between the subdomains and the corresponding rules

son their index-value correlations in the way of existing field-sensitive analyses [156, 51], which ensures the soundness theoretically.

Formally, we state the soundness of abstract semantics by the following theorem.

Theorem 4.6.2 *Given a program* P*, there exists an abstraction function* α *such that for any concrete memory state* M *and abstract memory state* $\mathbf{M}$ *if* $\alpha(M) \sqsubseteq_{\mathbf{M}} \mathbf{M}$*, then*

$$M \vdash P : M' \ \wedge \ \mathbf{M} \vdash P : \mathbf{M}' \ \Rightarrow \ \alpha(M') \sqsubseteq_{\mathbf{M}} \mathbf{M}'$$

Proof. We sketch the proof of the theorem as follows. The key of the proof is to construct the abstraction function α. According to Definition 4.3.1 and Definition 4.5.1, we can define the function $\sigma : \mathcal{V} \to \mathbf{O}$ as follows to abstract the values in the concrete memory by the abstract objects in the abstract memory.

- If v is an address in the concrete memory, σ maps it to the abstract memory object based on the allocation-site abstraction, Specifically, $\sigma(v)$ is the abstract memory object allocated by the statement applying the address v.
- If v is a literal, σ maps v to the corresponding literal object $o \in \mathbf{O}_c$.

Given a concrete memory M at the program location ℓ, we can define $(\widehat{\mathbf{E}}, \widehat{\mathbf{L}}, \widehat{\mathbf{C}}, \widehat{\mathbf{U}}) := \alpha(M)$ by utilizing σ. First, we construct the abstract environment $\widehat{\mathbf{E}}$ based on E. According to the concrete memory M, we can easily computes the path constraint ϕ by creating the conjunction of the equality of the variables in the branch conditions before ℓ. Thus, we define $\widehat{\mathbf{E}}$ as follows for any $v \in \mathcal{V}$:

$$\widehat{\mathbf{E}}(\sigma(v)) = \{(\sigma(E(v)), \phi)\}$$

Second, we construct the abstract layout $\widehat{\mathbf{L}} := (\widehat{\mathbf{B}},\ \widehat{\mathbf{R}})$. Here, we consider two cases in which $\sigma(\nu)$ is a position-dependent and value-dependent abstract container object, respectively.

- $\sigma(\nu) \in \mathbf{O}_p$: We can obtain the memory layout of the position-dependent container object storing at the address ν based on the concrete memory M, i.e., $L(\nu, i) = (i, \nu_i)$, where $1 \leqslant i \leqslant k$. Therefore, we have

$$\widehat{\mathbf{B}}(\sigma(\nu)) = \{(\{(o, T) \mid o \in \mathbf{O}\},\ \{(\sigma(\nu_i), \phi) \mid 1 \leqslant i \leqslant k\})\}$$

$$\widehat{\mathbf{L}}(\sigma(\nu)) = \{((\sigma(\nu_1), \cdots, \sigma(\nu_k)),\ \phi)\}$$

- $\sigma(\nu) \in \mathbf{O}_v$: We can obtain the memory layout the value-dependent container object storing at the address ν based on the concrete memory M, i.e., $L(\nu, \kappa_i) = (\kappa_i, \nu_i)$, where $1 \leqslant i \leqslant k$. Similarly, we have

$$\widehat{\mathbf{B}}(\sigma(\nu)) = (\{(\sigma(\kappa_i), \phi) \mid 1 \leqslant i \leqslant k\},\ \{(\sigma(\nu_i), \phi) \mid 1 \leqslant i \leqslant k\})$$

$$\widehat{\mathbf{L}}(\sigma(\nu)) = \{((\sigma(\kappa_i),\ \sigma(\nu_i)),\ \phi) \mid 1 \leqslant i \leqslant k\}$$

Third, we construct the state in constant domain $\widehat{\mathbf{C}}$. For any variable v in the program, we enforce $\widehat{\mathbf{C}}(v) = \sigma(\theta)$ if v is always equal to a certain literal θ before ℓ. If v has not been initialized, we set $\widehat{\mathbf{C}}(v)$ to $\bot$. Otherwise, we let $\widehat{\mathbf{C}}(v)$ be $\top$.

Fourth, we can compute $\widehat{\mathbf{U}}(\sigma(\nu))$ in the similar way of defining $\widehat{\mathbf{C}}$. Specifically, $\widehat{\mathbf{U}}(\sigma(\nu)) = 1$ if the container object storing at the address ν satisfy the condition of an anchored container before the program location ℓ. Otherwise, we set $\widehat{\mathbf{U}}(\sigma(\nu))$ to 0.

Hence, we obtain the abstraction function α, which maps M to $\widehat{\mathbf{M}} = (\widehat{\mathbf{E}},\ (\widehat{\mathbf{B}},\ \widehat{\mathbf{R}}),\ \widehat{\mathbf{C}},\ \widehat{\mathbf{U}})$. Given the concrete semantics in Figure 4.6, we can examine the relation $\alpha(M')$ and $\mathbf{M}'$ based on Definitions 4.9 and 4.10 straightforwardly. Intuitively, the composition of the partial abstract transformers and the witness operators updates the abstract memory conservatively, preserving the relation that $\alpha(M') \sqsubseteq_{\mathbf{M}} \mathbf{M}'$. Meanwhile, the monotonicity of the join operator $\sqcup_{\mathbf{M}}$ guarantees the soundness of analyzing the program in the presence of the branches and loops. Finally, the sound function summary implies the soundness of analyzing the program with function calls. Q.E.D.

According to the rules of computing value-flow edges in Table 4.1, we can further obtain the following corollary based on Theorem 4.6.2, which states the soundness of the overall value-flow analysis. We omit the sketch of its proof in the paper, as it can be obtained from the soundness of computing the abstract memory immediately.

Corollary 4.6.1 *Given a program* P*, if a value* u *at the program location* ℓ_1 *flows to a value* v *at the program location* ℓ_2 *in a concrete execution of* P*, then there must exist a value-flow edge from* $u@\ell_1$ *to* $v@\ell_2$ *in the VFG constructed in the memory orientation analysis.*

4.7 Demand-Driven Reachability Analysis

Once we obtain the VFG by the memory orientation analysis in Section 4.6, we can reduce the container-aware value-flow problem to a reachability problem [157, 3], which enable us to effectively investigate data propagation within data-centric application code. For a specific client, we conduct a demand-driven reachability analysis by traversing the graph and collecting the value-flow facts of interest. The section presents two fundamental clients, namely thin slicing and value-flow bug detection, to demonstrate that our approach benefits program understanding and improves memory safety, respectively. We utilize the motivating example in Figure 4.2 and its VFG in Figure 4.3 to illustrate the details of the clients throughout the section.

4.7.1 Thin Slicing

Program slicing identifies a subset of the program relevant to a program variable and a statement, called the *seed*. It has wide applications in program understanding [123, 124, 125] and debugging [131, 132]. Different from traditional slicing, the *thin slice* for a seed includes only the statements that affect the values of the variable directly, called the *producer statements*, and exclude the dependencies of the base pointer and control dependencies [123]. Thus, thin slices are smaller than conventional slices and support more precise program understanding.

To identify the producer statements, we conduct the reachability analysis by backward traversing the VFG from the seed. As blamed in [123], data structures are a major source of slice pollution. The precise reasoning of container semantics enables the slicer to obtain more precise slices for given programs.

Example 4.7.1 *Consider the variable* q *and the statement* ids.peek() *at line 18 in Figure 4.2. We can obtain the set of the slices* S *as follows by a backward traversal from* $q@\ell_{18}$*:*

$$\begin{aligned} S = \{\, & s_1 : \text{“a”}@\ell_4 \hookrightarrow i@\ell_7 \hookrightarrow ids\_arg1 \hookrightarrow q@\ell_{18}, \\ & s_2 : \text{“b”}@\ell_6 \hookrightarrow j@\ell_{11} \hookrightarrow ids\_arg1 \hookrightarrow q@\ell_{18}, \\ & s_3 : null@\ell_9 \hookrightarrow j@\ell_{11} \hookrightarrow ids\_arg1 \hookrightarrow q@\ell_{18} \,\} \end{aligned}$$

If the analysis does not distinguish the objects in containers, the thin slice also includes the insertions at line 5, which is a spurious producer statement. Specifically, we have the set of the slices S'*:*

$$S' = S \cup \{s_4 : null@\ell_5 \hookrightarrow i@\ell_7 \hookrightarrow ids\_arg1 \hookrightarrow q@\ell_{18}\}$$

4.7.2 Value-Flow Bug Detection

Value-flow bugs cover a wide category of program bugs, such as NPE [30], memory leak [4], and taint vulnerabilities [126, 158]. For example, detecting NPE suffices to perform a forward graph traversal, checking the reachability of the value-flow path from *null* to the dereferenced pointer. Similarly, taint vulnerability detection is essentially the problem of analyzing the reachability from the sources to the sinks specified in the taint specifications [25].

In many data-centric systems, such as Web applications [127, 12], value flows are often propagated through containers, some of which may trigger the bugs. Our approach strengthens the bug detection for these programs and, thus, improves system reliability from the application side.

Example 4.7.2 *The NPE detector traverses the VFG in Figure 4.3 from the* null *values to the dereferenced pointers, i.e.,* p, q, *and* r *at line 21. It discovers that the value flow is reachable from* $null@\ell_5$ *to* $p@\ell_{21}$*, forming the path*

$$p_1 : null@\ell_5 \hookrightarrow hs\_args1 \hookrightarrow p@\ell_{17} \hookrightarrow p@\ell_{21}$$

Thus, the NPE detector reports the NPE without false positives. However, a container mashing-based analysis reports two false positives caused by the following spurious value flows even if it is path-sensitive:

$$p_2 : \mathtt{null}@\ell_5 \hookrightarrow \mathtt{hs\_args2} \hookrightarrow \mathtt{r}@\ell_{19} \hookrightarrow \mathtt{r}@\ell_{21}$$
$$p_3 : \mathtt{null}@\ell_5 \hookrightarrow \mathtt{i}@\ell_7 \hookrightarrow \mathtt{id\_arg1} \hookrightarrow \mathtt{q}@\ell_{18} \hookrightarrow \mathtt{q}@\ell_{21}$$

4.7.3 Summary

The VFG of a given program precisely summarizes how values flow through containers, enabling the clients to solve the instances of the value-flow problem by a demand-driven reachability analysis. If necessary, the path constraints are collected on demand and then solved by an SMT solver. Our approach judiciously delays constructing and reasoning about the disjunctions until SMT solving in the reachability analysis. The solver only checks the constraints of certain paths and bypass irrelevant ones, further promoting the scalability of our approach. Also, the precision often goes arm in arm with scalability in the analysis [159, 160]. The strong updates in the memory orientation analysis reduce the facts in each subdomain, yielding a more sparse VFG. Moreover, it can decrease the number of the traversed paths in the reachability analysis, alleviating the overall overhead.

4.8 Implementation

We have implemented ANCHOR based on the static analysis platform PINPOINT [3, 115] in Ant Group, using Z3 [161] as the SMT solver. ANCHOR supports a variety of value-flow analyses, such as the value-flow bug detection and thin slicing. Following PINPOINT, ANCHOR achieves the context-, flow-, field-, and path-sensitivity in the client analysis. Our work extends the memory model of PINPOINT, enabling the precise reasoning of container memory layouts. In this section, we mainly present the details on the implementations of the memory orientation analysis and the reachability analysis.

***Memory Orientation Analysis in* ANCHOR**. In the memory orientation analysis, ANCHOR analyzes the collections in JCF, Java legacy collections, and data structures in Java EE, which are widely utilized in real-world data-centric applications [41, 12]. Table 4.2

Table 4.2: List of containers

Framework	Name	Category	Framework	Name	Category
JCF	ArrayList	position	Legacy	Vector	position
JCF	LinkedList	position	Legacy	Stack	position
JCF	HashSet	value	Java EE	ServletContext	value
JCF	TreeSet	value	Java EE	ServletRequest	value
JCF	LinkedHashSet	value	Java EE	HttpServletRequest	value
JCF	HashMap	value	Java EE	HttpServletResponse	value
JCF	TreeMap	value	Java EE	HttpServletRequestWrapper	value
JCF	LinkedHashMap	value	Java EE	HttpServletResponseWrapper	value
Legacy	Properties	value	Java EE	HttpSession	value
Legacy	Dictionary	value	Java EE	JspContext	position, value
Legacy	Hashtable	value	Java EE	PageContext	position, value

shows their names and categories. Specifically, we provide the container specification in a configuration file to specify the concrete semantics of each container method, such as inserting at the end of a position-dependent container, and removing the pair at a specific key in a value-dependent container. In the memory orientation analysis, ANCHOR loads the configuration file to identify container method calls, and updates abstract memory states by applying corresponding partial abstract transformers in Figure 4.9 and the witness operators in Figure 4.10. Particularly, ANCHOR is only concerned with the memory layouts of the containers and does not perform the reasoning of other sophisticated properties, such as the largest key of a TreeMap object and the first-inserted key in a LinkedHashMap object. It is worth mentioning that several kinds of Java EE containers are essentially a composition of multiple position-dependent containers or value-dependent containers. For example, a JspContext object maintains the JspWriter objects sequentially and the attribute objects with the keys. ANCHOR analyzes such container objects separately, regarding each of them as an object with two container-typed fields.

***Reachability Analysis in* ANCHOR**. In our reachability analysis, we collect and solve the constraints on demand in the traversal. Instead of leveraging a full-feature SMT solver, we also implement several semi-decision procedures as the intra-procedural preprocessing procedures, such as unit propagation, to determine unsatisfiable or valid constraints in a light-weighted manner [161], most of which can be achieved in linear time. For general cases, we model the variables in the program by bit vectors in the constraints, and set the length of a bit vector to the bit width. To avoid solving the formula in a large size, we adopt an eager strategy to prune the infeasible paths before they reach the sink nodes.

Specifically, we solve the condition of a path at specific program locations in the traversal even if it has not reached a sink node. If the current path condition has been unsatisfiable, we can safely stop the traversal, as it can not form a feasible path.

4.9 Evaluation

To demonstrate the utility, we address the following research questions:

- **RQ1:** How universe are anchored containers?
- **RQ2:** How efficient is ANCHOR in constructing the VFG?
- **RQ3:** How effective is ANCHOR in thin slicing?
- **RQ4:** How precise and scalable is ANCHOR in detecting value-flow bugs?

We conduct four experiments to answer the research question. First, we count anchored containers to show the universality of the concept. Second, we measure the time and memory overhead of the VFG construction for real-world programs. Third, we count the producer statements as the size of a thin slice to measure the precision of thin slicing. Finally, as a case study, we use taint vulnerabilities and null pointer exception to measure the precision and scalability in detecting value-flow bugs.

Subjects. We select 18 open-source data-centric applications on GitHub that are actively maintained and contain intensive usage of various containers. They cover different sizes (ranging from 19 KLoC to 5.12 MLoC) and diverse domains (such as RPC frameworks, data management systems, etc.). Besides, many of these projects, such as MyBatis, HBase, and Hadoop, are the fundamental infrastructure of data-centric applications. They are extensively and frequently scanned by academic static analyzers [129, 162, 27], and industrial tools, and thus expected to have every high quality. We also select several open-sourced applications in Ant Group, including sofa-rpc, atlas, and dubbo, to show our commercial value for the company. Particularly, we choose the OWASP benchmark projects [147] as the subjects to evaluate taint vulnerability detection, as configuring taint specifications for real-world programs might be subjective.

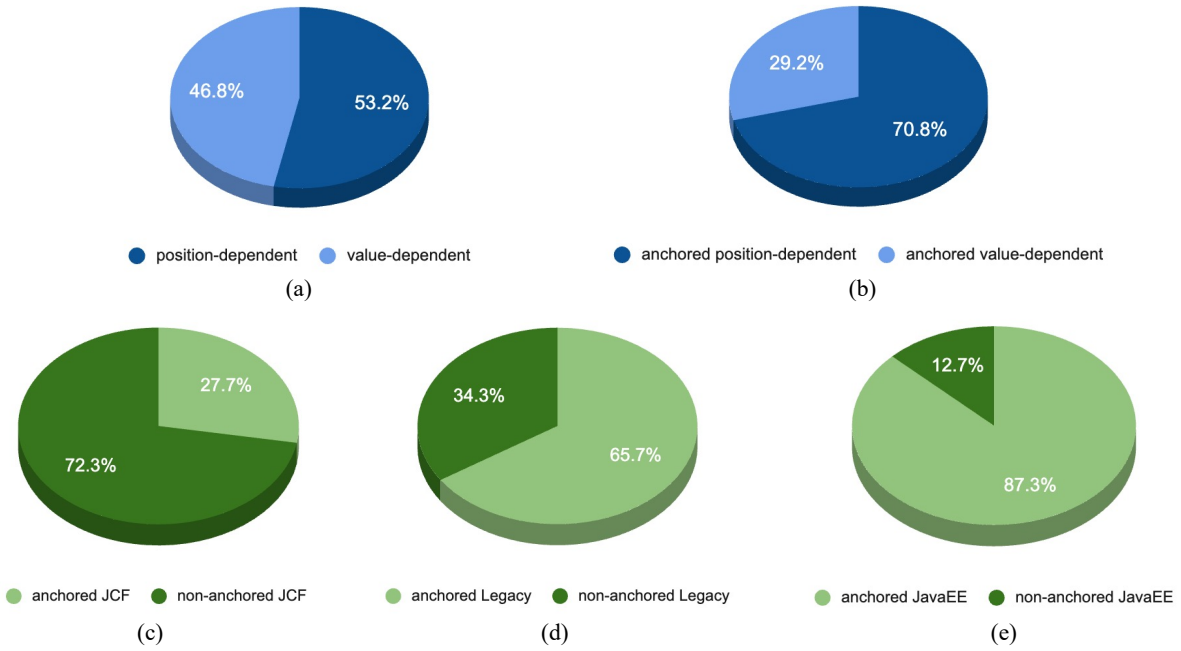


Figure 4.13: Proportions of different kinds of containers. (a): Proportions of position-dependent and value-dependent containers; (b): Proportions of anchored position-dependent and anchored value-dependent containers; (c), (d), and (e): Proportions of anchored and non-anchored containers in different frameworks.

Environment. We evaluate ANCHOR on a 64-bit machine with 40 Intel(R) Xeon(R) CPU E5-2698 v4@2.20GHz and 512GB of physical memory. Following previous studies [163, 3], we set the time limit of an SMT call to 10 seconds. Any analysis is run with a limit of 6 hours and 150GB of memory.

4.9.1 Identifying Anchored Containers

To show the prevalence of anchored containers, we count the anchored containers at the exit of each project by the memory orientation analysis. Specifically, we examine the uniqueness domain **U** at the exit and count the number of the container objects o which is mapped 1 by **U**. Several container objects are anchored containers at specific program locations, e.g., the HashMap object o_2 at line 7 in Figure 4.2. However, we do not consider them in such a fine-grained manner, although they can still promote the value-flow analysis. We also count the non-anchored containers in each project and measure the distribution of anchored containers in different types and frameworks.

Table 4.3 shows the numbers of anchored containers in the column **#AC**. The result shows that anchored containers widely exist in real-world programs. On average, there are 1.13 anchored containers in 1 KLoC, which demonstrates their prevalence. Particularly, over 1 thousand anchored containers exist in the project NetBeans, posing the necessity of analyzing them precisely and efficiently. Besides, The column **#NAC** in Table 4.3 shows the number of non-anchored containers. Although non-anchored containers take up a larger proportion in real-world programs, the orientation analysis can still effectively improve the precision of many value-flow clients, which will be evidenced by the answers to other research questions.

Figure 4.13 shows more details of the proportions of various kinds of containers. First, Figure 4.13 (a) reveals that position-dependent containers, such as ArrayList and LinkedList, are slightly more frequently used than value-dependent containers, such as HashMap and Dictionary. Second, Figure 4.13 (b) shows that position-dependent containers take up 70.8% of all the anchored containers, while the proportion of value-dependent containers is only 29.2%. The phenomenon mainly comes from the common practice of developers, as they often add or remove an element at the beginning or the end of a position-dependent container, and use more non-literal keys in a value-dependent container. Third, Figure 4.13 (c), (d), and (e) show the proportions of anchored and non-anchored containers in three frameworks. Specifically, the collections in JCF, which are general-purposed containers, are often used with non-literal keys or constant indexes in a flexible manner, and the proportion of non-anchored containers reaches 72.3%. In contrast, 83.7% Java EE data structures are anchored containers, as they are mostly used to store specific messages, e.g., network requests and responses, which often take literals as keys.

Answer to RQ1: Anchored containers widely exist in real-world programs. There are 1.13 anchored containers in 1 KLoC of the experimental subjects. The proportions of anchored containers in the JCF collections, Java legacy collections, and Java EE data structures are 27.7%, 65.7%, and 83.7%, respectively.

Table 4.3: The numbers of anchored containers and overhead of building the VFG

Project	Description	Size (KLoC)	#AC	#NAC	VFG-N		VFG-O		VFG-S	
					Time (min)	Mem (GB)	Time (min)	Mem (GB)	Time (min)	Mem (GB)
GraphJet	Graph processing system	19	35	85	0.1	0.4	0.1	0.4	0.1	0.4
mapper	Server application	22	18	77	0.3	0.6	0.4	0.7	0.3	0.6
light-4j	Microservice platform	44	48	182	0.3	1.5	0.3	1.6	0.3	1.6
roller	Server application	54	87	267	1.2	1.7	1.4	1.9	1.2	1.8
MyBatis	ORM framework	61	81	220	1.0	2.8	1.1	3.1	1.1	3.1
sofa-rpc	RPC framework	74	72	238	1.3	3.7	1.5	3.9	1.4	3.8
Glowstone	Server application	86	125	303	1.4	3.1	1.7	3.4	1.5	3.3
DolphinScheduler	Eventing infrastructure	90	132	457	1.3	2.7	1.6	2.9	1.5	2.9
atlas	Server application	142	226	551	1.7	4.1	2.1	4.7	2.0	4.6
Struts	Web framework	170	197	765	2.9	5.1	3.3	5.6	3.2	5.5
dubbo	RPC framework	184	173	736	3.0	5.5	3.5	6.1	3.2	5.9
IoTDB	Data management system	236	445	1,498	6.7	10.0	7.2	10.9	6.8	10.2
Spring-Boot	Web framework	346	396	1,152	7.1	10.4	7.9	12.1	7.5	11.8
Cassandra	Database system	538	534	1,250	11.3	15.3	13.1	18.4	12.5	17.7
Hibernate-ORM	ORM framework	787	452	1,288	13.4	20.8	16.6	23.1	14.5	22.3
HBase	Data management system	791	677	1,626	14.9	21.6	17.2	24.3	15.8	23.7
Hadoop	Data management system	1,811	769	3,041	25.7	34.5	28.6	38.6	27.5	37.4
NetBeans	IDE platform	5,122	1,273	5,029	54.8	81.5	61.1	86.2	57.3	84.9

4.9.2 Constructing Value-Flow Graph

To evaluate the scalability of ANCHOR, we investigate the overhead of ANCHOR in the VFG construction. Specifically, we set up two configurations to construct the VFG. In the first configuration (`VFG-O`), we perform the memory orientation analysis to utilize the anchored containers, while in the second configuration (`VFG-S`), we smash the container objects as many existing value-flow analyzers [54, 25]. To better quantify the overhead of analyzing container semantics, we also add the configuration `VFG-N` as a blank control group, in which we do not analyze container method calls.

The columns **VFG-N**, **VFG-O**, and **VFG-S** in Table 4.3 show the overhead of time and memory under the three configurations, respectively. We can find that

- The memory orientation analysis introduces negligible overhead compared with the analysis based on smashing containers. Both of them finish the construction for any project in 62 minutes with 86.2G peak memory. For the projects with less than 1 MLoC, two analyses only demand around 18 minutes and 25G memory.
- Compared with the analysis under `VFG-N`, the memory orientation analysis consumes at most 24.9% more time and 20.3% more memory. When analyzing the

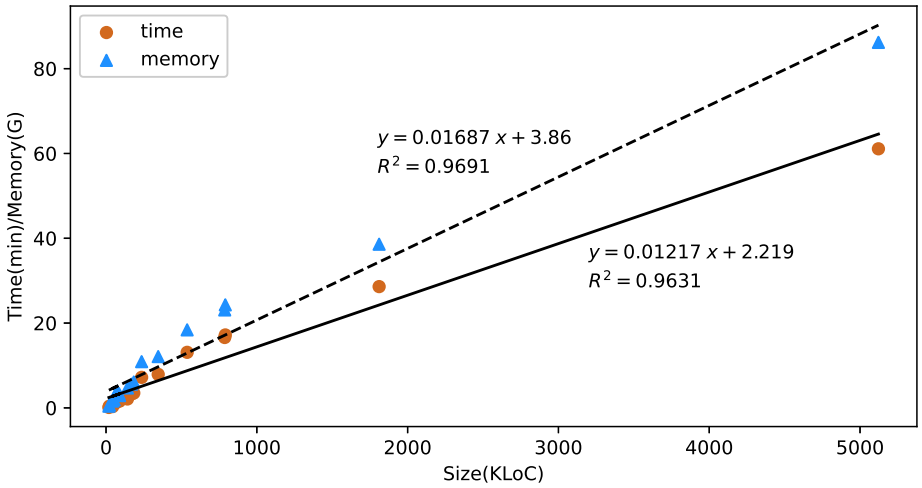


Figure 4.14: Scalability of the VFG construction under the configuration VFG−O

project NetBeans with 5.12 MLoC, ANCHOR only spends extra 6.3 minutes and 4.7G peak memory on the precise reasoning of container semantics.

- We also adopt regression analysis to study the observed scalability under VFG−O, of which the result is shown in Figure 4.14. The R-squared values for time and memory are 0.9631 and 0.9691, respectively, which indicates the overhead grows nearly linearly at a gentle rate and shows the potential scalability of ANCHOR.

The scalability of ANCHOR in the VFG construction benefits from two major designs. First, we perform a delay reasoning of value-flow paths, and only encode the path condition symbolically in the VFG without any explicit solving process. Second, fewer program facts are generated in the abstract state **M** in the presence of the strong updates, which makes the abstract transformers possibly avoid more computation. The observed linear scalability promotes the practicality of ANCHOR to analyze the large-scale programs in the real world.

Lastly, it is worth mentioning that we construct the VFG for the whole program, which can support a variety of value-flow clients. When we focus on a specific client, e.g., the NPE detection, we could concentrate on particular value flows, and only analyze containers with specific features, such as the containers that may contain *null* values. However, our experimental data shown in Table 4.3 and Figure 4.14 has demonstrated the low overhead of the memory orientation analysis. Moreover, ANCHOR essentially analyzes non-

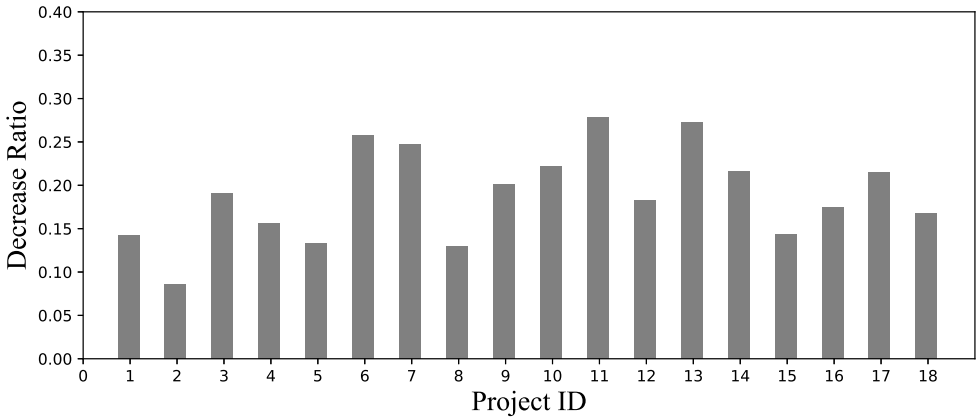


Figure 4.15: Decrease ratio of slice sizes under `TS-O` over `TS-S`

anchored containers as pointers by container smashing and regards anchored containers as data structures with a set of fields, which does not introduce significant overhead theoretically.

> **Answer to RQ2:** ANCHOR features the linear scalability in the value-flow graph construction for real-world programs and finishes the analysis in 62 minutes with 86.2 GB of peak memory for the program with 5.12 MLoC.

4.9.3 Answering Thin Slicing Queries

To Show the precision improvement in the thin slicing, we follow the previous studies [123, 164] and compare the sizes of thin slices under two configurations, i.e., `TS-O` and `TS-S`, in which we perform the memory orientation analysis and smash the container objects, respectively. Due to the soundness of two thin slicers, a smaller average size of thin slices indicates a higher precision of the thin slicer. For each project, we randomly select 100 pairs of an access method call and its return value as the seeds. Particularly, we do not solve constraints to make the analysis light-weighted in the scenario of program understanding.

Figure 4.15 shows the decrease ratio of thin slice sizes under `TS-O` over the ones of `TS-S`, in which project IDs are assigned based on the project sizes. It is shown that 17.1% fewer producer statements are discovered under `TS-O` than `TS-S` on average. Besides,

Table 4.4: NPE detection result

Project	NPE-O			NPE-S			Infer		
	Time (min)	Mem (GB)	#FP/#R	Time (min)	Mem (GB)	#FP/#R	Time (min)	Mem (GB)	#FP/#R
GraphJet	0.8	1.3	0/0	0.8	1.2	1/1	0.5	0.8	2/2
mapper	1.2	1.9	0/0	1.1	1.7	1/1	1.0	1.9	1/1
light-4j	3.8	2.3	0/2	3.1	2.7	0/2	4.9	1.6	2/4
roller	4.9	4.1	0/0	4.3	3.8	1/1	2.1	5.7	4/4
MyBatis	9.3	6.0	0/1	8.9	7.4	2/3	7.2	10.1	5/6
sofa-rpc	7.6	5.4	0/1	7.9	6.1	1/2	5.6	4.3	2/2
Glowstone	14.0	7.5	0/0	11.9	8.3	1/1	12.9	13.7	3/3
DolphinScheduler	12.8	6.3	0/0	12.5	6.5	0/0	10.2	8.9	1/1
atlas	14.6	8.1	0/0	13.5	9.3	2/2	NA	NA	NA
Struts	17.9	9.0	0/1	20.1	9.6	0/1	NA	NA	NA
dubbo	18.7	9.3	0/4	19.3	9.5	0/4	NA	NA	NA
IoTDB	26.5	14.5	0/2	26.2	17.8	1/3	28.4	12.1	5/7
Spring-Boot	31.3	16.8	1/1	30.5	16.1	2/2	34.6	20.7	3/3
Cassandra	52.4	21.9	0/2	49.8	21.4	3/5	43.2	33.9	6/7
Hibernate-ORM	66.5	35.2	0/0	69.6	38.1	1/1	51.1	46.3	4/4
HBase	69.9	42.4	0/0	67.1	39.5	1/1	NA	NA	NA
Hadoop	141.7	66.7	0/4	127.8	73.9	8/12	113.7	89.5	14/17
NetBeans	297.3	121.5	1/4	271.4	139.6	15/18	OOM	OOM	OOM
			2/22			**40/60**			**53/62**

NA means we fail to run INFER on the experimental projects.
OOM means INFER runs out of memory.

we compare the thin slices generated under two configurations. It is found that the thin slices generated under `TS-S` contain all the statements in the slices generated under `TS-O`. Therefore, the size decrease of the thin slices indicates that ANCHOR provides more precise slices under the configuration `TS-O` effectively, which benefits from the precise VFG constructed by the memory orientation analysis. Therefore, we conclude that the memory orientation analysis brings better precision in thin slicing than smashing containers. The more precise thin slices can provide better insight for the developers to understand and debug the program in the development.

Answer to RQ3: ANCHOR eliminates 17.1% producer statements that are spurious on average for thin slicing in real-world projects compared with the thin slicer smashing containers.

4.9.4 Detecting Value-Flow Bugs

Following the previous experiments, we set up the configurations `Taint-O`/`Taint-S` and `NPE-O`/`NPE-S` for the detections of the two types of bugs, respectively.

Detecting Taint Vulnerabilities

To evaluate the effectiveness of ANCHOR in the taint vulnerability detection, we choose the OWASP benchmark projects [147], which is a Java test suite with thousands of exploitable test cases of taint vulnerabilities. The vulnerabilities covered by OWASP benchmark projects include cross-site scripting attacks, information leakage, improper error-handling attacks, etc. Particularly, we select all the programs using containers as the experimental subjects, only concentrating on quantifying the benefit of analyzing container-manipulating programs.

We evaluate ANCHOR upon the collected subjects under two configurations `Taint-O` and `Taint-S`, in which we perform the memory orientation analysis and the container mashing, respectively. The results show that ANCHOR discovers all the 352 taint flows through containers with no false positive under `Taint-O`, while the false-positive ratio reaches 31.0% (158/510) under `Taint-S`. Therefore, the memory orientation analysis can significantly improve the precision of analyzing the OWASP benchmark projects, demonstrating the practical use in detecting taint vulnerabilities in the presence of containers.

Detecting Null Pointer Exception

To measure the precision and scalability of the NPE detection, we evaluate ANCHOR upon the real-world projects under the configurations `NPE-O` and `NPE-S`. Particularly, we count the NPE reports caused by the value flows through containers to show the impact of reasoning container semantics. We run the prominent bug detector INFER to compare the overhead and precision [165, 166]. Table 4.4 shows the numbers of reported bugs, false positives, and the overhead, based on which we summarize the following findings.

Precision and Overhead. Overall, ANCHOR analyzes the experimental subjects with high efficiency and scalability. First, the reports of the analysis under `NPE-O` are subsumed by the ones under `NPE-S`, and the false-positive ratios are 9.1% (2/22) and 66.7% (40/60), respectively. Upon the submission, 12 of the true positives had been confirmed by the developers [148]. Besides, it is worth noting that most of the bugs are discovered in the popular infrastructure projects, such as MyBatis, Struts and Hadoop, showing the impact of ANCHOR for infrastructure reliability.

```
1 public PropertiesConfiguration() {
2   Set<String> propertiesNames = getInputStrsFromUser();
3   List<PropertiesProvider> providers = new ArrayList<>();
4   for (String name : propertiesNames) {
5     if ("true".equals(name)) {
6       providers.add(null);
7     } else {
8       providers.add(getExtension(name));
9     }
10  }
11  Properties properties = ConfigUtils.getProperties();
12  for (PropertiesProvider provider : providers) {
13   properties.putAll(provider.initProperties());
14  }
15  ConfigUtils.setProperties(properties);
16 }
```

Figure 4.16: A confirmed NPE in the project dubbo

Second, ANCHOR finishes analyzing the project NetBeans with 512.2 MLoC in 5 hours within 121.5 GB peak memory under NPE−O. An interesting finding in the evaluation is that the memory orientation analysis reduces the overhead in several projects, such as sofa-rpc, Struts, and Hibernate-ORM. The major reason is that the strong updates introduce fewer value-flow facts, which further form fewer value-flow paths, alleviating the overhead of the graph traversal and constraint solving. Also, our work can be further improved by reducing the time and memory overhead. In the evaluation, we only measure the extra overhead introduced by our approach. The techniques of reducing the overhead of static analyzers can benefit ANCHOR seamlessly in an orthogonal manner [115].

Case Study. We show two typical NPE reports discovered under the two configurations. Figure 4.16 shows a confirmed NPE bug in the project dubbo, which is reported under NPE−S and NPE−O. If the name of the property is equal to "true", a *null* value is inserted into the ArrayList object at line 6 and finally dereferenced in the traversal. Under the configuration NPE−S, ANCHOR maintains the ownership of the container object pointed by providers, and thus finds that provider can be *null* in the traversal, which causes an NPE at line 13. Meanwhile, ANCHOR also reports the NPE under the configuration NPE−O due to its soundness. It is also worth mentioning that Figure 4.16 shows a typical pattern of container usage. Specifically, the elements are dereferenced in the traversal under conditions unrelated to their indexes. Even if the container is non-anchored, ANCHOR does not report a false positive under NPE−O as long as there exists a *null* value in the container.

Figure 4.17 shows two false positives in the project NetBeans reported under the con-

```
1 public void action(String w) {
2   Map<String, Object> config = new HashMap<>();
3   config.put("WORD", w == null ? "" : w);
4   config.put("IS_WHOLE", Boolean.FALSE);
5   config.put("BLOCK", null);
6   run((String)config.get("WORD"),
7         (Boolean)config.get("IS_WHOLE"));
8 }
```

Figure 4.17: Two false positives in NetBeans reported under the configuration NPE−S

figuration NPE−S. The values paired with "WORD" and "IS_WHOLE" are not *null*, while the analysis does not distinguish them from the *null* value paired with "BLOCK", causing two false positives at lines 6 and 7. In contrast, ANCHOR avoids reporting the false positives, as the HashMap object is an anchored container, of which the index-value correlation is precisely tracked by ANCHOR. Figure 4.17 demonstrates a common usage pattern of anchored value-dependent containers, which often exists in the configuration modules of a project. The values paired with literal keys are produced by different expressions introducing different program facts, which finally yields specific value-flow bugs at particular keys. Our memory orientation analysis effectively identifies this pattern and analyzes index-value correlations precisely, further supporting precise value-flow bug detection.

Comparison with Infer. The column **Infer** in Table 4.4 shows that ANCHOR and INFER share the similar overhead. INFER introduces 53 false positives in 62 reports, and all the true positives reported by INFER are also detected by ANCHOR. The fundamental reason of its high false-positive ratio is that INFER can not support precise container reasoning, e.g., it reports the two false positives in Figure 4.17. Also, INFER can not fully track path conditions when the path variables do not collude with preconditions, introducing the infeasible value flows in the NPE detection. After multiple attempts, INFER still fails to analyze several projects because of the crash and out-of-memory, denoted by NA and OOM, respectively. In contrast, ANCHOR finishes analyzing all the projects in the given budget of time and memory, showing the superiority in terms of scalability.

There are other static analyzers detecting NPEs in real programs [167]. Particularly, INFER analyzes the data structures with bi-abduction reasoning and achieves the field-, flow-, and context-sensitivity with good scalability, sharing the similar style of our work. Thus, we select INFER for comparison to show the advantages of ANCHOR. We also seek to evaluate COMPASS [63] while it is outdated for the operating systems we can set up.

Also, COMPASS is proposed to analyze C/C++ programs, and its implementation does not support analyzing Java programs.

> **Answer to RQ4:** ANCHOR detects all the taint vulnerabilities in the OWASP benchmark programs using containers with no false positive. Also, it uncovers 20 null pointer exceptions in 18 real-world Java programs with 9.1% as its false-positive ratio and finishes analyzing the program with 5.12 MLoC in 5 hours.

4.9.5 Threats to Validity

There are two major threats to the validity of our approach. The first threat to the validity is whether our approach can be generalized to the containers in a variety of third-party libraries. Actually, our memory orientation analysis only relies on the specifications of container methods. As explained in Section 4.8, the developers can classify the semantics of the methods and abstract them by the container methods in the language syntax defined in Figure 4.5. This process does not involve much expert knowledge, and thus supports the generality of our approach.

The second threat to the validity of our work is whether our approach supports a sound value-flow client analysis. Recall that Theorem 4.6.2 and Corollary 4.6.1 guarantee the soundness of value-flow analysis. For a specific client, such as the NPE detection, we can discover all the value-flow paths and collect the path conditions by traversing the VFG. However, we set the time budget of solving a constraint in the implementation, possibly discarding feasible paths when the solver fails to solve the path conditions in 10 seconds, which might introduce unsoundness to a specific value-flow client.

4.9.6 Discussion

This section presents more discussions on the benefit of anchored containers, the limitations, and future work.

Benefit of Anchored Containers. The experiments demonstrate that the anchored containers enable the analysis to obtain high precision with low overhead. Although precise reasoning of general containers requires constructing and solving the constraints in a

more sophisticated logic theory, the modification patterns of the anchored containers permit us to specialize the container axioms [168] and acquire the value flows based on the modification history. The combined domain in Section 4.5 essentially extends the solving procedure and supports constructing more precise VFG in the memory orientation analysis. Meanwhile, the experiment of NPE detection shows that the anchored containers can make the precision go arm in arm with scalability in several analyses of the subjects. As guaranteed by Theorem 4.6.1, more facts are reduced by the strong updates in the subdomains in the presence of the anchored containers, which prevents the reachability analysis from examining spurious value-flow paths. The phenomena have been discussed in the recent static analyses [159, 160]. The anchored containers play an important role in unleashing the precision and efficiency of the analysis simultaneously.

***Path sensitivity in* ANCHOR**. In the memory orientation analysis, we establish a combined abstract domain embodied with path constraints, which enables us to achieve precise abstraction of points-to facts and container memory layouts. The encoded path constraints can eventually promote the precision of downstream clients, such as value-flow bug detection. Particularly, the false positive ratio would be much higher if we gave up path sensitivity in our memory model. However, it should be noted that the superiority of ANCHOR over container smashing does not depend on the precision enhancement introduced by path sensitivity. Even if our analysis is path insensitive, ANCHOR can still obtain more precise memory layouts of anchored containers than the ones obtained by container smashing, which implies a lower false positive ratio than the one of a path-insensitive container smashing-based approach.

***Limitation of* ANCHOR**. Our study shows the effectiveness of ANCHOR in analyzing container-manipulating programs, but several limitations still exist as follows.

- ANCHOR only analyzes the memory layout of the containers, and it does not concern other properties of containers, such as the size, the emptiness, and insertion order, which also affect the value-flows in the program. Although we have not found the spurious results caused by the unawareness of these properties in the experiment, it can indeed decrease the precision and recall of ANCHOR theoretically.

- ANCHOR takes the manually-written container specifications as input to identify the

container types and the behavior of their methods. For example, we manually investigate all the class definitions in JavaEE to collect the container-like data structures in Table 4.2. However, the manual annotation often involves a huge laborious effort, especially in the presence of numerous third-party libraries [23, 12].

- ANCHOR does not reason the index-value correlations of non-anchored containers, which introduces the precision loss when analyzing the value flows through them. As shown in Figure 4.13 (c), there is a large proportion of JCF collections that are non-anchored. Although the anchored containers can improve the precision of analyzing their ownership, their index-value correlations are still blurred in the memory orientation analysis. This prevents ANCHOR from further improving the precision of several value-flow analyses, such as thin slicing.

Future Work. It could be promising to explore the following directions to design an automatic, fast, and precise static analyzer for data-centric applications using containers. First, it would be meaningful to track more container properties to obtain more precise value flows. For example, the size of a container could support more precise reasoning of the path condition if a branch condition involves its size. Second, the automatic inference of container specifications would enable the static analyzer to identify the container types and analyze client programs in a more automatic manner. More data structures in the third-party libraries, such as Apache Commons Collections [11], Trove [169], and Google Guava Collection [170], could be discovered and identified as containers, which benefits the subsequent analyses [55, 12, 63]. Third, it would be worth designing more advanced analyses to reason the non-anchored containers. The relations among the indexes of container method calls could support discovering more precise value flows in general cases. For example, a must-not alias analysis would enable us to prune spurious value flows through non-anchored containers according to their keys, which could further increase the precision of value-flow analysis. Fourth, it would also be a promising direction to design an on-demand VFG construction algorithm. The values unrelated to a client can be skipped so that we can safely ignore specific value flows through containers, which can further decrease the overhead of the analysis. Lastly, anchored containers can also widely exist in the programs written in other programming languages, especially Python and JavaScript programs. For instance, a dictionary with constant keys is often utilized

to maintain the content of a json file or a web message. Identifying and precisely analyzing anchored containers in the domain-specific Python or JavaScript programs would significantly promote the downstream analyses upon them.

4.10 Conclusion

We have described ANCHOR to analyze value flows through containers to investigate data propagation within data-centric applications. ANCHOR identifies anchored containers for strong updates in a light-weighted memory orientation analysis and discovers the precise value flows induced by container method calls. As a result, it supports producing more precise thin slices and uncovering 20 NPEs with only two false positives. ANCHOR outperforms the state-of-the-art value flow analyses and container reasoning techniques in terms of precision and scalability, and features in the ability to analyze millions of lines of code. The underlying insight of ANCHOR can benefit various clients upon data-centric applications, such as program understanding and bug detection, which improves system reliability from the application side.

CHAPTER 5

INFERRING API ALIASING SPECIFICATIONS FROM LIBRARY DOCUMENTATION

5.1 Introduction

In modern programming languages, programmers often develop data-centric applications based on various libraries, which offer fundamental building blocks for application-side implementation. Undoubtedly, the behaviors of library APIs directly affect the functionality of the application code. As investigated by existing studies [22, 23], several library APIs are essentially generalized store and load operations, forming aliasing relations through the load-store match. For example, the APIs HashMap.put and HashMap.get conduct the store and load operations, respectively. When they are invoked upon the same HashMap object with the same first parameters successively, the return value of HashMap.get can be aliased with the second parameter of HashMap.put. To identify value flows in the application code, a static analyzer relies on API aliasing specifications to understand how library APIs manipulate memory, which play critical roles for pointer analysis and other downstream clients. According to our investigation, many existing static analysis techniques rely on manually specified library API aliasing specifications [25, 55, 12]. However, the emergence of third-party libraries introduces a large number of APIs, making laborious effort unacceptable in practice.

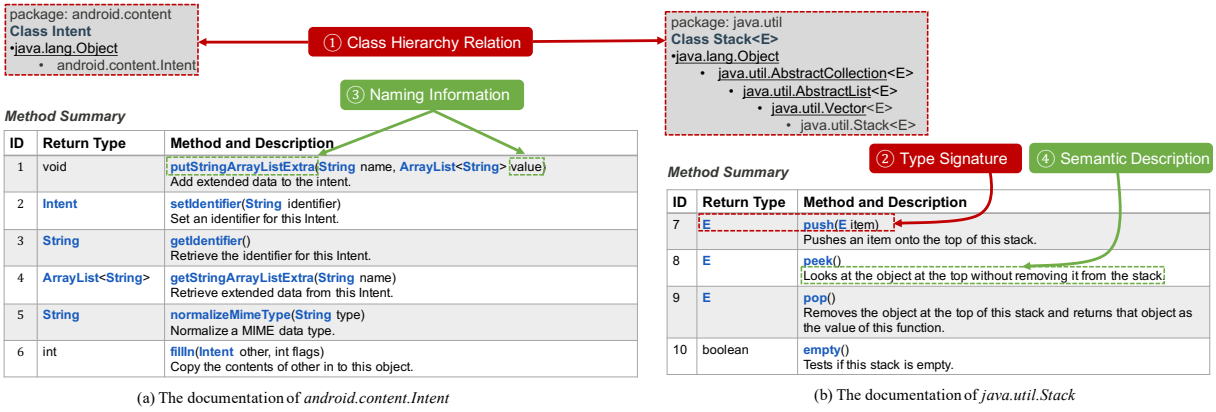
This work targets the API aliasing specification inference problem to support library-aware alias analysis. Existing approaches infer API aliasing specifications from three perspectives. The first line analyzes the source code statically [67, 66]. Although it can derive the function summaries as the API aliasing specifications, the solution suffers the scalability problem due to deep call chains [171]. More importantly, the implementation of several library APIs can depend on native code, such as System.arraycopy in the implementation of java.util.Vector, which makes static analysis intractable [22]. The second line of the techniques constructs unit tests via active learning to trigger the execution of library APIs,

so as to infer aliasing relations in the runtime [22]. Compared to static analysis-based inference techniques, they are more applicable when the source code of the library is unavailable. However, it can be infeasible to generate unit tests to trigger the target library APIs due to the difficulties of constructing the parameters with complex data structures and executing APIs in specific devices or environments. Third, several researchers learn the aliasing specifications from applications using libraries [23], which does not require the source code of the libraries or the execution of the programs. Unfortunately, their approach only discovers the API specifications used in the applications, finally causing the low recall in the inference.

This paper presents a new perspective on inferring API aliasing specifications. Different from existing studies, we utilize another important artifact of third-party libraries, namely documentation, to analyze the semantics of library APIs. As shown in Figure 5.1, library documentation contains formal semantic properties, e.g., class hierarchy relation and API type signatures, and informal semantic information, e.g., API semantic descriptions and naming information. Although the library documentation demonstrates the library API semantics in detail, it is far from trivial to derive API aliasing specifications from the documentation. First, effectively understanding the informal semantic information is quite difficult. Even if we apply the recent advance in the large language models, e.g., feeding the documentation of android.content.Intent to CHATGPT, we can only obtain nine API aliasing specifications, all of which are incorrect. Second, documentation may contain a long list of classes and APIs, significantly introducing the overhead of the specification inference. For example, feeding the lengthy documentation to CHATGPT not only increases the time overhead but also introduces a high financial cost due to the enormous token consumption.

To effectively achieve the inference with high efficiency, we propose our inference algorithm named DAINFER, which originates from three key insights:

- The class hierarchy relation and API type signatures determine the available APIs for a given class and over-approximate aliasing relations based on the types of API parameters and returns. If two values can not be aliased, we do not need to analyze the naming information and API semantic descriptions, which decreases the overhead by avoiding applying NLP models.



(a) The documentation of *android.content.Intent*

(b) The documentation of *java.util.Stack*

Figure 5.1: Library documentation example. m_i denotes the API with the ID i.

- The named entities in the names indicate the high-level semantics of the parameters and return values, which narrows down aliasing relations. In Figure 5.1(a), the named entities in the API name of Intent.getIdentifier and the parameter name of Intent.setIdentifier are the same, indicating that the return value of Intent.getIdentifier can be aliased with the parameter of Intent.setIdentifier.

- The API semantic descriptions can reveal the conducted memory operations with specific verbs, enabling us to identify API pairs that potentially form API aliasing specifications. In Figure 5.1(b), for example, the verbs "push" and "look" show that the APIs Stack.push and Stack.peek conduct the insertion and read operations upon the memory, respectively.

Based on our insights, we propose DAINFER, an algorithm to infer API aliasing specifications. Technically, we introduce a graph representation to over-approximate the aliasing relations between parameters and return values based on type information. To interpret informal semantic information, we use a large language model and a tagging model to abstract memory operation kinds and high-level semantics of API parameters/return values, respectively. Then, we reduce the specification inference problem to an optimization problem that enforces the aliasing pairs between API parameters as many as possible for precise semantic abstraction. Particularly, the optimization problem poses constraints over the results of the two NLP models. To solve the problem efficiently, we propose the neurosymbolic optimization algorithm, which interacts with the two NLP models in a demand-driven manner, achieving low resource cost in the inference.

We implemented our approach DAINFER and evaluated it upon Java classes in several popular libraries, which are widely used in many data-centric applications. Based on library documentation, DAINFER achieved the inference with a precision of 79.78% and a recall of 82.29%, consuming 5.35 seconds per class on average. We also quantified the impact of the inferred API aliasing specifications on the pointer analysis and taint analysis. It was shown that DAINFER promoted the alias analysis by discovering 80.05% more aliasing facts for the API return values and enabled the taint analysis to discover 85 more taint flows in the experimental subjects. Our main contributions of this work include:

- We introduce a new paradigm of inferring API aliasing specifications and reduce the inference problem to an optimization problem over a graph representation of the library documentation.
- We propose a novel technique, namely neurosymoblic optimization, to efficiently solve the optimization problem and effectively infer the API aliasing specifications.
- We extensively evaluate our approach over real-world libraries to demonstrate its superiority over existing techniques and quantify its impact on client analyses.

The rest of this chapter is organized as follows. Section 5.2 introduces the background of library-aware alias analysis to motivate the API aliasing specification problem inference and demonstrate the workflow of DAINFER after the discussion of existing techniques. Section 5.3 provides a formal definition of our problem. Section 5.4 and Section 5.5 demonstrates the technical design of DAINFER. The details of the implementation and evaluation are presented in Section 5.6 and Section 5.7. We finally conclude this chapter in Section 5.8.

5.2 Background and Overview

In this section, we introduce the background of API aliasing specification inference and outline our key idea of inferring API aliasing specifications from documentation.

5.2.1 Library-Aware Alias Analysis

Modern software systems, especially data-centric applications, heavily depend on various libraries. A recent study found that a Java web application can include an average of 48 libraries transitively [172]. This prevalence of library usage stimulates the demand for modeling API semantics in fundamental static analyses, such as alias analysis. However, the deep call chains and unavailable source code (e.g., native functions) complicate the scalability and applicability of static analysis. Many static analyzers use specifications to abstract the library API semantics to achieve library-aware analysis. Specifically, the API aliasing specification for an API pair (m_1, m_2) indicates: When m_1 and m_2 conduct the store and load operations upon memory, respectively, the return value of m_2 may be aliased with the parameter of m_1 if m_2 is invoked after m_1 upon the same object. Based on the specification, a static analyzer can model the library API semantics without explicitly analyzing the implementation of m_1 and m_2, ultimately promoting the scalability and applicability of the overall analysis.

Example 5.2.1 *In Figure 5.1(a), when the first parameters of* Intent.putStringArrayListExtra *and* Intent.getStringArrayListExtra *are aliased, the return value of the latter can be aliased with the second parameter of the former if they are invoked successively upon an object.*

5.2.2 Different Perspectives of Inferring API Aliasing Specifications

With the increasing number of third-party libraries, manually specifying the API aliasing specifications demands incredibly laborious effort [25, 12, 55]. To mitigate this problem, previous studies infer the API aliasing specifications from different artifacts, including the library implementation [67], application code using libraries [23], and unit tests constructed via active learning [22]. However, their solutions can be bothered with three main drawbacks. First, analyzing the library implementation suffers the scalability issue due to complex program structures, such as deep call chains, and even becomes inapplicable due to the unavailability of the implementation or the presence of native code. Second, inferring the specifications from application code using libraries may fail to achieve high recall when specific APIs are not utilized in the application code. Third, deriving the alias-

ing facts from dynamic execution of unit tests suffers the inapplicability issue when it is infeasible to construct executable unit tests in specific devices or environments.

To fill the research gap, our work proposes another perspective to infer the API aliasing specifications. We realize that there is another essential library artifact, i.e., the library documentation, demonstrating the library API semantics in a semi-formal structure. As shown in Figure 5.1, the formal semantic properties, including the class hierarchy relation and API type signatures, are explicitly provided. Meanwhile, the naming information, e.g., the parameter names and API names, shows the intent of API parameters and return values, while API semantic descriptions demonstrate the functionalities of the APIs informally. These ingredients permit us to understand how the library APIs manipulate the memory and further form aliasing relations between their parameters and return values. More importantly, the documentation is often available for analysis, as the developers tend to refer to it during the development. Hence, inferring the API aliasing specifications from documentation would exhibit better applicability than the existing techniques.

5.2.3 Overview of DAInfer

Although the documentation guides the developers in understanding the API semantics, there exists a gap between the API knowledge and API aliasing specifications. Concretely, we need to understand how the API parameters are stored and how the API return values are loaded. However, achieving this is quite complicated in front of informal semantic information. Even if we leverage the new advances in the large language models, the models cannot understand how the APIs manipulate the memory and eventually fail to identify the aliasing relations based on the documentation. Also, interacting with the LLMs via online requests can bring quite high overhead and consume a large number of tokens in the presence of the long documentation.

To address the challenges, we propose a novel inference algorithm named DAInfer, which effectively understands the API semantics and efficiently infers the API aliasing specification from library documentation. Our key idea originates from three critical observations on the aliasing relations between the parameters and return values of the library APIs as follows.

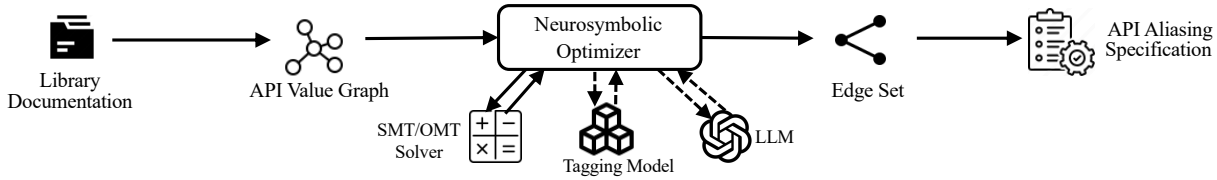


Figure 5.2: Workflow of DAINFER

- The parameters and return values should be type-consistent if they are aliased. Specifically, their types should be the same, or one of them is the sub-type/super-type of the other. Such facts are easily obtained from the class hierarchy relation and API type signatures in the documentation. In Figure 5.1, for example, we can obtain the potential aliasing relation between the return value of Intent.getIdentifier and the parameter of Intent.setIdentifier, while the second parameter of Intent.putStringArrayListExtra can not be aliased with the return value of Intent.getIdentifier.

- If the return values and parameters of two APIs are aliased, the named entities in their names tend to be the same, indicating the same high-level semantics. For example, the APIs Intent.setIdentifier and Intent.getIdentifier in Figure 5.1(a) share the same named entity identifier, indicating that they manipulate the same inner field. For general-purpose data structures, such as java.util.Stack in Figure 5.1(b), the API names of Stack.peek and Stack.pop do not have any named entities, indicating that their return values can be aliased with other parameters with consistent types.

- If a library API stores its parameters or loads the inner field as the return value, the verbs in its semantic description can reflect the memory operation kind intuitively. For example, the verbs set and insert are commonly used for the APIs storing its parameters, while the verbs get and return are prevalent in the semantic descriptions of the APIs loading the inner fields.

Based on the observations, we realize that we can leverage type information to over-approximate aliasing relations and utilize named entities and verbs to understand the high-level semantic meanings of the APIs. For any store-load API pair, we can finalize an API aliasing specification as long as we discover the parameters and return values with the same semantic meanings and consistent types. According to these insights, we design our inference algorithm DAINFER, of which the workflow is shown in Figure 5.2. Our key

technical design consists of three components.

- We introduce a new graph representation, namely the API value graph, to approximate the aliasing relations. After converting a library documentation to a normalized documentation model, we encode the potential aliasing relations in the API value graph.
- We conceptually reduce the API aliasing specification inference to an optimization problem upon the API value graph, where we select as many edges as possible to discover all the possible aliasing relations between parameters and return values. Particularly, we leverage two NLP models, namely a tagging model and a large language model, to extract the named entities and interpret semantic descriptions, respectively.
- We instantiate the optimization problem and propose an efficient neurosymbolic optimization algorithm to solve the problem, of which the solution induces the API aliasing specifications. Our neurosymbolic optimization algorithm interacts with the tagging model and the LLM in a demand-driven manner, significantly improving the efficiency of our algorithm.

Benefiting from our insights, our inference algorithm DAInfer simultaneously achieves high precision, recall, and efficiency. The high availability of library documentation also promotes the applicability of our approach in real-world scenarios. In the following sections, we will formulate our problem (Section 5.3) and provide our technical design (Section 5.4 and Section 5.5) in detail.

5.3 Problem Formulation

This section first formulates the documentation model (Section 5.3.1) and then defines the API aliasing specification (Section 5.3.2). Lastly, we provide the formal statement of the API aliasing specification inference problem and highlight the technical challenges (Section 5.3.3).

5.3.1 Documentation Model

Definition 5.3.1 (Documentation Model) *Given a library, its documentation model is* $\mathbf{L} := (\mathbf{H}, \mathbf{T}, \mathbf{N}, \mathbf{D})$*:*

- *Class hierarchy model* $\mathbf{H}$ *maps a class* $\mathtt{c}$ *to a set of classes, which are the superclasses of* $\mathtt{c}$*.*
- *Type signature model* $\mathbf{T}$ *maps* $(\mathtt{c}, \mathtt{m}, \mathtt{i})$ *to a type, where* $\mathtt{m}$ *is an API of the class* $\mathtt{c}$ *and* $\mathtt{i}$ *is the index of the parameter. Without ambiguity, we regard the index of the return value as -1.*
- *Naming model* $\mathbf{N}$ *maps* $(\mathtt{c}, \mathtt{m}, \mathtt{i})$ *to a string indicating the parameter name or API name, where* $\mathtt{m}$ *is an API of the class* $\mathtt{c}$ *and* $\mathtt{i}$ *is the index of the parameter. Without ambiguity,* $\mathbf{N}(\mathtt{c}, \mathtt{m}, -1)$ *indicates the name of the API* $\mathtt{m}$ *of the class* $\mathtt{c}$*.*
- *Description model* $\mathbf{D}$ *maps* $(\mathtt{c}, \mathtt{m})$ *to a string indicating the API semantic description.*

Example 5.3.1 *According to the documentation of the class* Intent *in Figure 5.1, we have*

$$\mathbf{H}(\textit{Intent}) = \{\textit{Object}\},\ \ \mathbf{T}(\textit{Intent}, \mathtt{m}_1, -1) = \textit{void},\ \ \mathbf{T}(\textit{Intent}, \mathtt{m}_1, 1) = \textit{ArrayList<String>}$$

$$\mathbf{N}(\textit{Intent}, \mathtt{m}_1, 0) = \textit{name},\ \ \mathbf{N}(\textit{Intent}, \mathtt{m}_1, 1) = \textit{value},\ \ \mathbf{N}(\textit{Intent}, \mathtt{m}_1, -1) = \textit{putStringArrayListExtra}$$

$\mathbf{D}(\textit{Intent}, \mathtt{m}_1)$ *is "Add extracted data to the intent". Here,* $\mathtt{m}_1$ *is* Intent.putStringArrayListExtra*. Due to space limits, we do not discuss other APIs in detail.*

Notably, we can collect all the APIs supported by a specific class and its superclasses, forming the universe of the available APIs when using the class. The naming information and API semantic descriptions are informal specifications, guiding the developers to use proper APIs in their programming contexts. Based on the documentation model, not only do developers achieve their program logic conveniently, but also analyzers can understand the behavior of each API.

5.3.2 API Aliasing Specification

To support the library-aware alias analysis, we concentrate on the API aliasing specification inference and follow an important form of aliasing specifications formulated in the prior study [23], which is defined as follows.

Definition 5.3.2 (API Aliasing Specification) *An API aliasing specification is* (m_1, m_2, P, t)*, where* m_1 *and* m_2 *are two APIs,* $P := \{(i_1^{(1)}, i_1^{(2)}), \cdots, (i_j^{(1)}, i_j^{(2)})\}$ *is a set of non-negative integer pairs, and* t *is an non-negative integer. It indicates that the return value of* m_2 *can be aliased with the* t*-th parameter of* m_1 *if*

- m_1 *is called before* m_2 *upon the same object*
- *The* $i_k^{(1)}$ *and* $i_k^{(2)}$*-th parameters of* m_1 *and* m_2 *are aliased accordingly.*

Here, $0 \leqslant i_k^{(1)} \leqslant n_1$*,* $0 \leqslant i_k^{(2)} \leqslant n_2$*, and* $0 \leqslant k \leqslant j < \mathit{min}(n_1, n_2)$*.* n_1 *and* n_2 *are the numbers of the parameters of the APIs* m_1 *and* m_2*, respectively.*

Definition 5.3.2 shows that the APIs m_1 and m_2 conduct the store and load operations upon the memory, respectively. Unlike simple load and store operations of pointers, storing and loading the values upon memory may depend on the values of other parameters. Intuitively, the set P indicates the pre-condition of the aliasing relation between the return value of m_2 and the t-th parameter of the m_1. Notably, the parameters of m_1 and m_2 are not necessarily aliased to enforce the aliasing relation between the return value of m_2 and the t-th parameter of m_1 when P is empty. We call m_1 and m_2 form a store-load API pair without ambiguity.

Example 5.3.2 *In Figure 5.1(a), we have two API aliasing specifications* $(m_1, m_4, \{(0,0)\}, 1)$ *and* $(m_2, m_3, \emptyset, 0)$*. Specifically, the API aliasing specification* $(m_1, m_4, \{(0,0)\}, 1)$ *indicates the fact that the return value of* Intent.getStringArrayListExtra *can be aliased with the second parameter of* Intent.putStringArrayListExtra *when the two APIs are invoked upon the same object and their first parameters are aliased.*

Notably, the API aliasing specification in Definition 5.3.2 is more general than the one targeted by USPEC [23]. Specifically, USPEC only infers that calling m_2 may return a value aliased with the t-th parameter of a preceding call of m_1 on the same object *if all other parameters are aliased*. However, there exist many store-load API pairs in which not all the other parameters are aliased. For instance, the API createBitmap of android.graphics.Bitmap sets the values of DisplayMetrics, Config, width, and height simultaneously, while the

method getConfig only fetches the value of Config. Our formulation in Definition 5.3.2 is expressive enough to depict such the store-load API pair. Hence, our specifications can promote downstream clients better than the ones targeted by USPEC [23].

5.3.3 Problem Statement

We aim to address the API aliasing specification inference problem from another perspective. As demonstrated in Section 5.3.1, the library documentation provides various forms of semantic properties of the library APIs. Hence, we hopefully derive the API aliasing specifications from the documentation without conducting deep semantic analysis upon the source code or program runtime information.

Notably, the API aliasing specification for a given store-load API pair may not be unique. In Example 5.3.2, for instance, $(m_1, m_4, \emptyset, 1)$ is also a valid specification, which does not pose any restrictions upon the parameters of the two APIs. In our work, we want to ensure that the inferred specifications exhibit the pre-conditions of the aliasing relations as strong as possible. Finally, we state the problem of the API aliasing specification inference as follows.

> Given a documentation model $\mathbf{L} = (\mathbf{H}, \mathbf{T}, \mathbf{N}, \mathbf{D})$, infer a set of API aliasing specifications S_{AS} such that $|P|$ is maximized for each $(m_1, m_2, P, t) \in S_{AS}$.

Technical Challenges. Although library documentation offers semantic information, solving the above problem is quite challenging. First, the naming information and API semantic descriptions can be ambiguous. Without an effective interpretation, we can not understand how the APIs operate upon the memory and identify aliasing relations between parameters and return values. Second, there are often many available APIs offered by a single class and even its superclasses. It is non-trivial to obtain high efficiency in front of a large number of available APIs for each class.

Roadmap. In this work, we propose an inference algorithm DAINFER to address the two technical challenges. Specifically, we introduce the documentation model abstraction to formulate semantic information, which enables us to reduce the original problem to an optimization problem (Section 5.4). Furthermore, we propose the neurosymbolic

optimization to efficiently solve the instantiated optimization problem (Section 5.5). Our implementation and evaluation demonstrate the effectiveness and efficiency of our approach DAINFER (Section 5.6 and Section 5.7).

5.4 Documentation Model Abstraction

This section presents the abstraction of our documentation model. We first propose the concept of the API value graph to over-approximate aliasing relations (Section 5.4.1). Besides, we introduce two label abstractions over the API value graph (Section 5.4.2), which enables us to reduce the API aliasing specification problem to an optimization problem (Section 5.4.3).

5.4.1 API Value Graph

As shown in Section 5.3.1, the formal semantic information, namely class hierarchy and the type signatures, reveals potential aliasing relations between API parameters and return values, while the informal semantic information, e.g., the names and API semantic descriptions, shows how parameters and return values are utilized in the API invocations. To depict aliasing relations that can be introduced by the API invocations, we propose a graph representation, namely the *API value graph*, as follows.

Definition 5.4.1 (API Value Graph) *Given a documentation model* $\mathbf{L} = (\mathbf{H}, \mathbf{T}, \mathbf{N}, \mathbf{D})$*, its API value graph is the labeled graph* $\mathsf{G} := (\mathsf{V}, \mathsf{E}, \ell_\mathtt{n}, \ell_\mathtt{d})$*, where*

- *The node set* V *contains API parameters and return values, which are referred to as API values.* $(\mathtt{c}, \mathtt{m}, \mathtt{i}) \in \mathsf{V}$ *if and only if* $(\mathtt{c}, \mathtt{m}, \mathtt{i}) \in \mathit{dom}(\mathbf{N})$ *or there is* $\mathtt{c}' \in \mathbf{H}(\mathtt{c})$ *such that* $(\mathtt{c}', \mathtt{m}, \mathtt{i}) \in \mathit{dom}(\mathbf{N})$*.*
- *The edge set* $\mathsf{E} \subseteq \mathsf{V} \times \mathsf{V}$ *indicates aliasing relations between API values. Specifically,* $(v_1, v_2) \in \mathsf{E}$ *if and only if* $\mathbf{T}(v_1) = \mathbf{T}(v_2)$*,* $\mathbf{T}(v_1) \in \mathbf{H}(\mathbf{T}(v_2))$*, or* $\mathbf{T}(v_2) \in \mathbf{H}(\mathbf{T}(v_1))$*.*
- *The name label* $\ell_\mathtt{n}$ *is a function that maps an API value to its name, i.e.,* $\ell_\mathtt{n}(v) = \mathbf{N}(v)$*.*
- *The description label* $\ell_\mathtt{d}$ *is a function that maps an API value to the semantic description of the API, i.e.,* $\ell_\mathtt{d}(v) = \mathbf{D}(\mathtt{c}, \mathtt{m})$*, where* $v = (\mathtt{c}, \mathtt{m}, \mathtt{i})$*.*

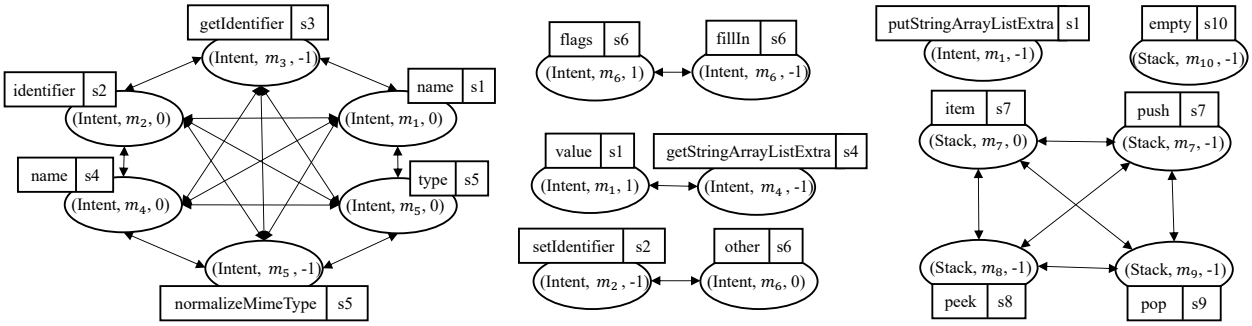


Figure 5.3: The API value graph of the documentation model induced by the documentation in Figure 5.1

The API value graph regards API values, namely API parameters and return values, as first-class citizens, and depicts their high-level semantics with labels. Intuitively, an edge from (c, m_1, i_1) to (c, m_2, i_2) indicates the fact that the two values may be aliased when m_2 is invoked after m_1 upon the same object. Meanwhile, the two labels attach the informal semantic information to API values, showing their usage intention. From a high-level perspective, the API value graph over-approximates aliasing relations according to class hierarchy relation and type signatures and still preserves informal semantic information as labels to support further specification inference.

Example 5.4.1 *Figure 5.3 shows the API value graph for the documentation model induced by the classes in Figure 5.1, where the name labels and description labels are shown in the left and right boxes, respectively.* s_i *indicates the API semantic description of* m_i *in Figure 5.1. Specifically, the edge from* $(\mathsf{Intent}, m_2, 0)$ *to* $(\mathsf{Intent}, m_5, 0)$ *indicates that the first parameters of* Intent.setIdentifier *and* Intent.normalizeMimeType *may be aliased when the two APIs are invoked successively.*

5.4.2 Label Abstraction

According to Definition 5.4.1, the edges of the API value graph approximate aliasing relations over API values based on their types. However, not all the aliasing relations can hold when using APIs. In Figure 5.1, for example, the return value of getIdentifier and the first parameter of normalizeMimeType are unlikely to be aliased as the named entities in their names are different, revealing different usage intention of the two API values. To formulate this intuition, we formally introduce the concept of the *semantic unit abstraction*, which shows the high-level semantics of API values.

Definition 5.4.2 (Semantic Unit Abstraction) *A semantic unit abstraction α_τ maps a string s to a set of nouns, i.e., $\alpha_\tau(s) = \{w_i \mid \tau(w_i) = \textsf{NOUN},\ s = w_1 \odot w_2 \odot \cdots \odot w_n\}$. Here, $\odot$ is the concatenation operation, and s is the concatenation of w_i. τ is a tagging function that maps a word to a grammatical tag. Particularly, we call the named entities in $\alpha_\tau(s)$ as the* semantic units *of s.*

Example 5.4.2 *The nouns appearing in the API name of* getStringArrayListExtra *include* string, array, list, *and* extra. *Hence, we have $\alpha_\tau(\textsf{getStringArrayListExtra}) = \{\textsf{string}, \textsf{array}, \textsf{list}, \textsf{extra}\}$.*

Essentially, the semantic unit abstraction extracts the named entities from the names as semantic units, which shows the high-level semantics of API values, enabling us to refine aliasing relations according to the following two intuitions: (1) If two API values v_1 and v_2 have the names with the same semantic units, we can obtain the confidence that they are very likely to indicate the same object in the memory; (2) If the name of an API value does not have any semantic units, we can conservatively regard that it can be aliased with any other API values with consistent types. Hence, we formally define the *semantic unit consistency* to formulate the two intuitions.

Definition 5.4.3 (Semantic Unit Consistency) *Given a semantic unit abstraction α_τ upon an API value graph $G = (V, E, \ell_n, \ell_d)$, two nodes v_1 and v_2 are semantic-unit consistent, denoted by $(v_1, v_2) \in C_\tau \subseteq E$, if and only if (1) $\alpha_\tau(\mathbf{N}(v_1)) = \alpha_\tau(\mathbf{N}(v_2))$, or (2) $\alpha_\tau(\mathbf{N}(v_1)) = \emptyset \vee \alpha_\tau(\mathbf{N}(v_2)) = \emptyset$.*

Example 5.4.3 *Consider the API value graph in Figure 5.3. We have $\alpha_\tau(\textsf{getIdentifier}) = \{\textsf{identifier}\}$, so the return value of the API* getIdentifier *and the first parameter of* setIdentifier *are semantic-unit consistent for the class* Intent. *Also, we have $\alpha_\tau(\textsf{item}) = \{\textsf{item}\}$ and $\alpha_\tau(\textsf{peek}) = \emptyset$, so the return value of* peek *and the first parameter of* push *are semantic-unit consistent for the class* Stack.

Lastly, we notice that API semantic descriptions show how the API values are manipulated upon the memory. According to our problem statement in Section 5.3.3, we need to identify whether the API return value is originally stored by another API. Specifically, we give a formal definition of the concept named *memory operation abstraction* as follows.

Definition 5.4.4 (Memory Operation Abstraction) *A memory operation abstraction function* α_o *maps a semantic description* s *to* $\alpha_o(s) \subseteq M$, *where* $M = \{I, D, R, W\}$. *The elements in* M *indicate the insertion (I), deletion (D), read (R), and write (W) operation upon the memory.*

Notably, we classify common memory operations into four categories. Although the insertion and deletion are both specific kinds of write operations, there are still several write operations that simply initialize or modify specific fields of the classes, which shows the necessity of introducing the category W for the memory operation abstraction.

Example 5.4.4 *According to Figure 5.1, we have* $\alpha_o(s_1) = \alpha_o(s_2) = \{W\}$ *and* $\alpha_o(s_3) = \alpha_o(s_4) = \{R\}$ *for the class* Intent. *For the class* Stack, *we have* $\alpha_o(s_7) = \{I, W\}$, $\alpha_o(s_8) = \{R\}$, *and* $\alpha_o(s_9) = \{R, D, W\}$.

To sum up, the semantic unit abstraction and the memory operation abstraction interpret the informal semantic descriptions with the sets of semantic units and memory operations, respectively, based on which we can refine potential aliasing relations indicated by the edges of the API value graph and identify store-load API pairs. In Section 5.5.2, we will demonstrate how to instantiate the two abstractions to support the specification inference.

5.4.3 Problem Reduction

Established upon the two label abstractions, we can effectively interpret the high-level semantics of API values and the memory operations conducted by the APIs. According to our problem statement in Section 5.3.3, we need to identify the store-load API pairs in the API aliasing specification inference. Particularly, we need to infer as many aliased parameters in each API pair as possible so that the inferred specification can pose a strong pre-condition over the API parameters, which finally induces a precise abstraction of the API semantics. Therefore, we can reduce the specification inference to an optimization problem over the API value graph as follows.

Definition 5.4.5 (Optimization Problem) *Given a semantic unit abstraction* α_τ *and a memory operation abstraction* α_o *upon an API value graph* $G = (V, E, \ell_n, \ell_d)$, *find an edge set* $E^* \subseteq E$ *with a maximal size* $|E^*|$ *satisfying the following constraints:*

- *(Degree constraint) For any* $v = (c, m, i) \in V$ *and* m'*, the following two conditions are satisfied:*

$$|\{v' \mid (v, v') \in E^*,\ v' = (c, m', i')\}| \leqslant 1,\ \ |\{v' \mid (v', v) \in E^*,\ v' = (c, m', i')\}| \leqslant 1$$

- *(Validity constraint) If* $(v_1, v_2) \in E^*$*, where* $v_1 = (c, m_1, i_1)$ *and* $v_2 = (c, m_2, i_2)$*, then there exist* $u_1 = (c, m_1, t)$ *and* $u_2 = (c, m_2, -1)$ *such that* $(u_1, u_2) \in E^*$*.*

- *(Semantic unit constraint) For any* $(v_1, v_2) \in E^*$*, where* $v_1 = (c, m_1, i_1)$ *and* $v_2 = (c, m_2, i_2)$*, the semantic unit abstraction of the names of* v_1 *and* v_2 *should satisfy*

  - *If* $i_2 \neq -1$*,* v_1 *and* v_2 *are semantic-unit consistent, i.e.,* $(v_1, v_2) \in C_\tau$
  - *If* $i_2 = -1$*,* v_1 *or* v_1' *is semantic-unit consistent with* v_2*, i.e.,* $(v_1, v_2) \in C_\tau$ *or* $(v_1', v_2) \in C_\tau$*, where* $v_1' = (c, m_1, -1)$*.*

- *(Memory operation constraint) For any* $(v_1, v_2) \in E^*$*, the following two conditions are satisfied:*

  - v_1 *satisfies* $I \in \alpha_o(\ell_d(v_1)) \vee (W \in \alpha_o(\ell_d(v_1)) \wedge D \notin \alpha_o(\ell_d(v_1))$
  - v_2 *satisfies that* $R \in \alpha_o(\ell_d(v_2))$

Definition 5.4.5 aims to maximize the $|E^*|$ to discover all the aliased parameters of each store-load API pair, which enforces the inferred API aliasing specifications abstract the API semantics precisely. The four kinds of constraints are posed upon the selected edges such that the solution can be reduced to the API aliasing specifications effectively. Specifically, the degree and validity constraints ensure that the edges induce the API aliasing specification defined Definition 5.3.2. Besides, the parameters of the APIs m_1 and m_2 should be semantic-unit consistent if they are connected by a selected edge. If a selected edge connects the parameter of m_1 and the return value of m_2, then the parameter of m_1 should be semantic unit-consistent with the return value of m_2. Lastly, the memory operation constraint ensures that the APIs m_1 and m_2 are likely to be store-load API pairs.

Finally, we can obtain the specifications based on the optimal solution as follows.

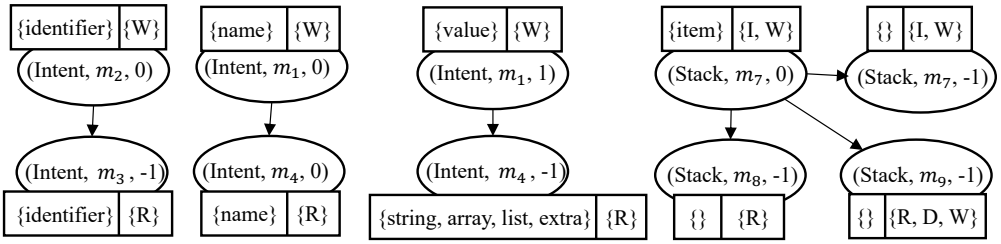


Figure 5.4: An optimal solution to the problem instance induced by the API value graph shown in Figure 5.3

Given the optimal solution E^* of the optimization problem defined in Definition 5.4.5, we can obtain the API aliasing specification $(m_1, m_2, P, t) \in S_{AS}$, where

- $P = \{(i_1, i_2) \mid ((c, m_1, i_1),\ (c, m_2, i_2)) \in E^*, i_2 \neq -1\}$
- t satisfies $((c, m_1, t), (c, m_2, -1)) \in E^*$

Example 5.4.5 *Figure 5.4 shows the optimal solution to the optimization problem over the API value graph in Figure 5.3, where the sets shown in the two boxes demonstrate the extracted semantic units and identified memory operations under the label abstractions in Examples 5.4.3 and 5.4.4. We discover six possible aliasing relations. Notably, although the semantic units of* $(\mathit{Intent}, m_4, -1)$ *are different from* $(\mathit{Intent}, m_1, 1)$*, they are exactly the same as the ones of* $(\mathit{Intent}, m_1, -1)$*, indicating that the second parameter of* m_1 *have the same semantics as the return value of* m_4*. The optimal solution finally induces the API aliasing specifications in Example 5.3.2.*

By reducing the original problem to the optimization problem in Definition 5.4.5, we only need to tackle two important sub-problems for the specification inference. First, we have to instantiate two label abstractions effectively so that we can precisely interpret the semantic meanings of the names and the memory operation kinds. Second, we need to design an efficient optimization algorithm to solve the optimization problem and further convert the optimal solution to the API aliasing specifications. In Section 5.5, we will provide the technical details of addressing the two sub-problems.

5.5 Inferring Specification via Neurosymbolic Optimization

This section presents the technical details of our algorithm DAINFER. Specifically, we demonstrate the overall algorithm in Section 5.5.1 and detail the label abstraction instantiation in Section 5.5.2. Besides, we present the neurosymbolic optimization in Section 5.5.3 to instantiate and solve the optimization problem given in Definition 5.4.5. Lastly, we summarize our approach and highlight its advantages in Section 5.5.4.

5.5.1 Overall Algorithm

As demonstrated in Section 5.4.3, we can reduce the API aliasing specification inference problem to an instance of the optimization problem given in Definition 5.4.5. Technically, we propose and formulate our specification algorithm in Algorithm 3, which takes as input a documentation model $\mathbf{L}$ and generates a set of API aliasing specifications S_{AS} as output. First, we derive the API value graph G from the documentation model $\mathbf{L}$ based on Definition 5.4.1 (Line 1). Second, we instantiate two label abstractions, i.e., α_τ and α_o, and further construct an instance of the optimization problem $\mathcal{P}$ defined in Definition 5.4.5 (Lines 2–3). Third, we propose the neurosymbolic optimization to solve the instance of the optimization problem $\mathcal{P}$ (Lines 4–5), and finally convert the optimal solution E^* to a set of API aliasing specifications S_{AS} (Line 6). Particularly, Definition 5.4.1 has demonstrated how to construct the API value graph, and converting the optimal solution to the specification is also explicitly formulated at the end of Section 5.4.3. In the rest of this section, we will provide more technical details on the label abstraction instantiation (Section 5.5.2) and the neurosymbolic optimization algorithm (Section 5.5.3), which finalize the functions getSemanticUnitAbs, getMemoryOperationAbs, and neuroSymOpt in Algorithm 3, respectively.

5.5.2 Label Abstraction Instantiation

According to Definitions 5.4.2 and 5.4.4, the two label abstractions demand two different kinds of natural language processing (NLP) techniques. Specifically, the semantic unit abstraction requires attaching the grammatical tags, while the memory operation abstraction replies to an NLP model to understand how an API manipulates the memory. In what fol-

Algorithm 3: Inference Algorithm

Input: $\mathbf{L}$: Documentation model;
Output: S_{AS}: A set of API aliasing specifications;

```
G ← constructAVG(L);
α_τ ← getSemanticUnitAbs();
α_o ← getMemoryOperationAbs();
P ← (L, G, α_τ, α_o);
E* ← neuroSymOpt(P);
S_AS ← convert(E*);
return S_AS;
```

lows, we will detail how to instantiate the two label abstractions with two different NLP models, respectively.

Instantiating Semantic Unit Abstraction.

According to common programming practices, the developers of libraries follow typical naming conventions [173], such as camel case, pascal case, and snake case. For example, userAccount is a parameter name using camel case, and get_account_balance is an API name using snake case. Notably, the sub-words are often separated with an underscore or begin with an uppercase letter. Hence, we can easily decompose each name s into the concatenation of several sub-words and further determine the tag of each sub-word.

However, we notice that the names of APIs or their methods can hardly be valid phrases or sentences. Simply applying the part-of-speech (POS) tagging would tag almost all the sub-words as the nouns. To obtain more precise tagging results, we leverage an existing probability model trained in Brown Corpus [174], which can return all the possible grammatical tags of each sub-word along with the occurrences. This enables us to determine whether or not a sub-word is more likely to be a noun. Formally, we instantiate the semantic unit abstraction as follows.

Definition 5.5.1 (Instantiation of Semantic Unit Abstraction) *Assume that* g_τ *maps a word* w *to a set of tag-occurrence pairs* $\{(\tau_j, k_j)\}$. *Then, the tagging function* τ *is defined as* $\tau(w) = \arg\max_{\tau_j} o(\tau_j)$, *where* $(\tau_j, o(\tau_j)) \in g_\tau(w)$, *which further instantiates* α_τ *in Definition 5.4.2.*

Example 5.5.1 *Consider the API* setIdentifier *in Figure 5.1. After splitting the API name into*

"set" and "identifier", we discover that "set" is more likely to be a verb than a noun, while "identifier" is very likely to be a noun. Hence, our instantiated semantic unit abstraction α_τ *maps* setIdentifier *to* {identifier}*, identifying* identifier *as the semantic unit of the API.*

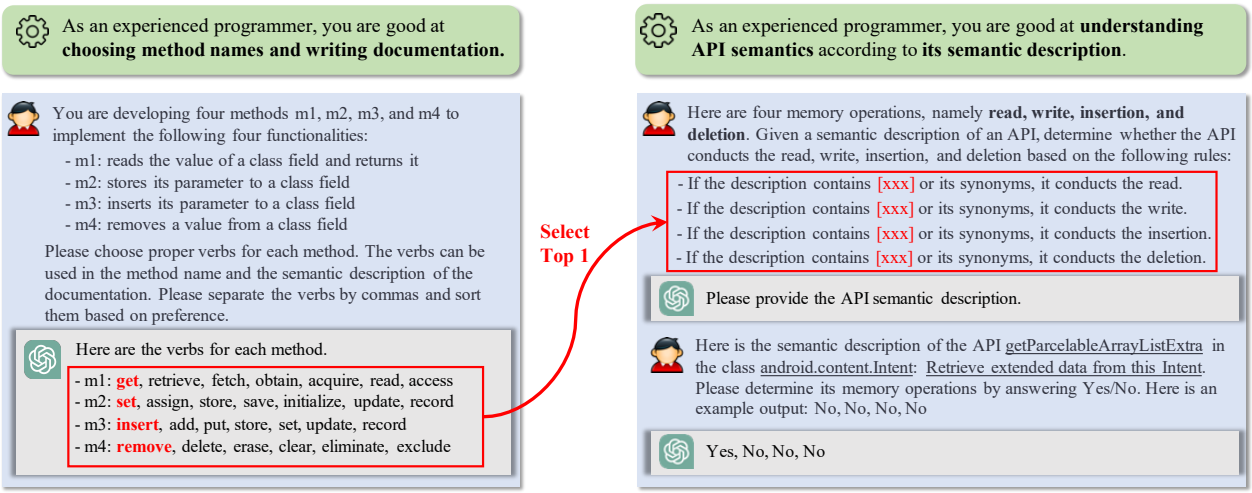
Instantiating Memory Operation Abstraction.

To instantiate an effective memory operation abstraction, we leverage an important programming practice: The developers often summarize the API functionality in a full sentence or a verb-object phrase as its semantic description. Particularly, the verbs in the semantic description intuitively depict the memory operations conducted by the API. Therefore, it is possible to instantiate an effective memory operation abstraction based on the verbs in the semantic description. However, the verbs used in the semantic descriptions can vary a lot, even if the APIs conduct the same kind of memory operation. For example, when describing an API conducting the memory insertion, developers can choose different verbs, e.g., "put", "insert", and "push". The diverse choices of the verbs describing a specific memory operation would make the inference suffer low recall if we just adopted a grep-like approach based on string matching.

Inspired by recent progress in the NLP community, we realize that the latest advances in the Large Language Models (LLMs) may provide new opportunities for resolving this issue [175, 176, 177]. Specifically, the LLMs have excellent abilities in text understanding, especially under the guidance of few-shot examples or descriptions of rules. Hence, we propose a two-stage prompting-based approach to instantiate the memory operation abstraction, which is demonstrated in Figure 5.5.

- First, we design the prompt in Figure 5.5 (a) to retrieve the verbs describing each memory operation. To obtain the most typical verbs, we enforce the LLM sort the verbs based on the preference.
- Second, we select the top-1 verbs recommended in the first stage and construct the prompt guiding the memory operation abstraction, which is shown in Figure 5.5(b). Finally, we obtain a response containing four "Yes"/"No" separated by commas.

Based on the above prompting process, we can obtain an instantiation of the memory operation abstraction, which is formally formulated as follows.



(a) Retrieve typical verbs via prompting

(b) Instantiate the memory operation abstraction via prompting

Figure 5.5: Instantiate the memory operation abstraction via two-staged prompting

Definition 5.5.2 (Instantiation of Memory Operation Abstraction) g_o *is the LLM obtained via two-staged prompting in Figure 5.5. Then the memory operation abstraction* α_o *satisfies that* $op \in \alpha_0(s)$ *if and only if the corresponding answer of* op *in* $g_o(s)$ *is "Yes", where* $op \in M$.

Example 5.5.2 *As shown in Figure 5.5(b), the output of the LLM is "Yes, No, No, No", indicating that the API* Intent.getStringArrayListExtra *only conducts the memory read. Hence, we have* $\alpha_o(s_4) = \{R\}$, *where* s_4 *is the semantic description of the API* Intent.getStringArrayListExtra.

Notably, our two label abstractions are achieved with different overheads. Specifically, the semantic unit abstraction only relies on the existing tagging model that can be applied efficiently. In contrast, the memory operation abstraction has to post the request to the online LLM model, of which the request frequency and number are restricted. To obtain high efficiency and decrease the token cost of the LLM, we have to propose an effective solving procedure, which will be demonstrated in Section 5.5.3.

5.5.3 Neurosymbolic Optimization

Based on the above intuitions, Algorithm 4 formulates our neurosymoblic optimization algorithm by utilizing the two NLP models in a demand-driven manner. Specifically, we process each API pair in each iteration. Initially, we check whether or not the degree

Algorithm 4: Neurosymbolic optimization

Input: $\mathcal{P}$: An optimization problem instance;
Output: E^*: The optimal solution;

```
1  foreach (c, m1), (c, m2) do
2      φd ← deriveDegreeConstraints(P);
3      φv ← deriveValidityConstraints(P);
4      if SMTSolve (φd ∧ φv)=UNSAT then
5          continue;
6      φs ← deriveSUConstraints(P);
7      if SMTSolve (φd ∧ φv ∧ φs)=UNSAT then
8          continue;
9      φo ← deriveMOConstraints(P);
10     E' ← Solve(obj(P), φd ∧ φv ∧ φs ∧ φ0);
11     E* ← E* ∪ E';
12 return E*;
```

constraint ϕ_d and the validity constraint ϕ_v are satisfied (Lines 2–5). If both of them are satisfied, we apply the tagging model to derive the semantic unit constraint ϕ_s (Line 6) and further examine the satisfiability of the conjunction of the three constraints (Line 7). If it is satisfiable, we apply the LLM to achieve the memory operation abstraction, and derive the memory operation constraint (Line 9). Based on an existing OMT solving technique [178], we select the maximal number of edges connecting the API values (Line 10) and append them to the set E^* (Line 11), which is returned as the optimal solution to the optimization problem.

Example 5.5.3 *Consider the APIs of* Intent *in Figure 5.1(a). When processing the APIs* Intent.fillIn *and* Intent.getIdentifier*, the validity constraint is not satisfied as there are no type-consistent parameters or return values. Hence, we do not apply the tagging model or the LLM. For the APIs* Intent.setIdentifier *and* Intent.normalizeMimeType*, we find that their parameters and return values are not semantic-unit consistent, so we do not invoke the LLM with their semantic descriptions.*

It is worth noting that the semantic unit constraint can also be instantiated by applying the LLM with a proper prompt. However, pairwise examining the names of an API and its parameters with the LLM can introduce huge overhead and hinder the opportunity to optimize the efficiency of Algorithm 4 with the lazy strategy of applying the models.

Hence, we conduct the named-entity recognition with a tagging model, which is more light-weighted than the LLMs.

5.5.4 Summary

Benefiting from our label abstraction instantiations and neurosymbolic optimization, our inference algorithm DAINFER features with two important advantages. First, our label abstraction instantiations make DAINFER effectively identify the high-level semantics of API values and the memory operations conducted by the APIs. Particularly, DAINFER has a good generalization ability in understanding the semantic description for the memory operation abstraction, which promotes the precision and recall of the specification inference. Second, we choose a lazy strategy of applying the NLP models in the neurosymoblic optimization. Noting that invoking an online LLM service can introduce more overhead than SMT solving, our neurosymoblic optimization algorithm can finally obtain the high efficiency of our inference algorithm.

Lastly, it is worth noting there exist several existing studies attempting to infer other kinds of API specifications from library documentation, such as taint specification [179] and memory management specification [180]. However, their approaches can not be easily extended to infer API aliasing specifications in our problem. Although existing techniques [179, 180] analyze API pairs, such as a pair of a source and a sink, the targeted specifications are not as sophisticated as the ones targeted in our work. Apart from identifying store/load API pairs, we still need to determine the possible aliasing facts between parameters and return values of the two APIs, which demands another domain-specific solution to derive such specifications from the documentation. Our work is the first trial to derive API aliasing specifications from documentation, demonstrating the opportunity to infer fundamental program facts with NLP models, especially LLMs, to support static analysis clients.

5.6 Implementation

We implement the approach DAINFER as a prototype. It parses the library documentation to extract the documentation model, which is further fed to the inference algorithm.

In what follows, we demonstrate more implementation details to convenience the reproduction of our work.

Documentation parsing. We implement the documentation parser by using *soup* Python package. For each documentation page describing the API semantics, we can extract the four kinds of information, including class hierarchy relation, API type information, naming information, and API semantic descriptions, Since library documentation pages almost have a uniform format, we do not have to make major changes to the implementation of the parser to adapt to different libraries.

Label abstraction instantiation. To instantiate the semantic unit abstraction, we utilize the conditional frequency distributions tool with Brown Corpus provided by Natural Language Toolkit [181] to determine whether or not a word is the most likely to be a noun. To instantiate the memory operation abstraction, we adopt gpt-3.5-turbo model with chat completions API to interpret the API semantic descriptions [182]. Specifically, we invoke the interface ChatCompletion.create to feed the constructed prompts to the LLM and fetch its response. In our implementation, we set the temperatures of the two stages of prompting to 0.7 by default.

Neurosymoblic optimization. We implement the neurosymbolic optimization based on Z3 solver [161, 178]. For any pair of APIs, we introduce $(n_1 + 1) \cdot (n_2 + 1)$ boolean variables to indicate whether the values of two APIs are aliased or not, where n_1 and n_2 are the numbers of the API parameters. To accelerate the LLM inference process, we parallelize the invocations of the LLM in eight threads. To avoid redundantly applying the tagging model and the LLM, we introduce the memorization technique to store the tagging result of each word and the response of the LLM upon each API semantic description. If a word or an API semantic description has been processed before, our algorithm directly reuses the previous result instead of applying the NLP models again.

5.7 Evaluation

To quantify the effectiveness and efficiency of DAINFER, we propose the following three research questions.

- **RQ1:** How effectively and efficiently does DAINFER infer the specifications?
- **RQ2:** How does DAINFER compare against other approaches?
- **RQ3:** How does DAINFER benefit library-aware static analysis clients?

5.7.1 Experimental Setup

All the experiments are performed on a 64-bit machine with 40 Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20 GHz and 512 GB of physical memory. We invoke the Z3 SMT solver with its default options.

Subjects. To make the comparison, we evaluate ATLAS [22], USPEC [23], and DAINFER upon the same set of Java classes. Specifically, the Java classes are collected from: (1) The classes of which the specifications are manually specified in FLOWDROID [25]; (2) The classes appearing in the inference results of USPEC [23]. Since the dataset of ATLAS [22] is not publicly available, we cannot conduct experiments on it. In total, our benchmark contains 167 Java classes offering 8,342 APIs, which range from general-purpose libraries, including Android framework and Java Collections Framework, to specific-usage libraries, such as Gson. Particularly, all the container types targeted by CRES and ANCHOR are all covered by our benchmark.5 Without ambiguity, we call the first and the second kinds of the classes form FLOWDROID benchmark and USPEC benchmark, respectively.

5.7.2 Effectiveness and Efficiency

Effectiveness. We run DAINFER upon the documentation models of the subjects to obtain the inferred specifications. To obtain the ground truth, we have to examine the API semantics and investigate each API pair for the class. Notice that investigating all the classes demands tremendous manual effort. Following the recent study [23], we randomly select 60 classes that offer 2,771 APIs in total. For each API, we need to examine whether it forms store-load API pairs with other APIs offered by the same class, of which the number can reach 50 on average. To make the manual examination more reliable, we refer to the specifications specified by the developers of FLOWDROID, and meanwhile, investigate

Table 5.1: Efficiency of DAINFER and its ablations

Tool	# Tagging	# LLM	Token Cost	Time Cost (sec)
DAINFER	32,325	2,950	726,425	892.93
DAINFER-TYPE	32,325	5,164	1,276,254	1,734.63
DAINFER-EXHAUSIVE	58,846	8,090	1,994,017	2,844.26

the library documentation and implementation simultaneously. Eventually, we obtain 988 API aliasing specifications as the oracle.

According to our investigation, we find that DAINFER achieves high precision and recall upon the experimental subjects. Specifically, it successfully infers 2,680 API aliasing specifications. For the randomly selected 60 classes, DAINFER infers 1,019 API aliasing specifications, 813 of which are correct, achieving a precision of 79.78%. After examining all the APIs of the selected classes, we discover that DAINFER misses 175 specifications, achieving a recall of 82.29%. Interestingly, we collect the specifications where the API names contain "get" or "set", and discover that such specifications only take up 33.49% of all the inferred ones. It shows that DAINFER can precisely understand how APIs operate upon the memory even if the API names contain diverse verbs. We also compare our inference results with the specifications in the FLOWDROID and USPEC benchmarks. It is shown that DAINFER infers 170 out of the total 210 specifications in FLOWDROID benchmark and 65 out of the total 82 specifications inferred by USPEC, achieving 81.0% and 79.3% recall upon the two benchmarks, respectively. The above results demonstrate that our approach can effectively infer the API aliasing specifications from the documentation.

Efficiency. We quantify the efficiency of DAINFER with four metrics, including the number of applying the tagging model, the number of applying the LLM, the token cost, and the time cost. As shown in Table 5.1. DAINFER applies the tagging model 32,325 times and interacts with the LLM 2,950 times using 722,009 tokens, and the overall time cost is 892.93 seconds (around 15 minutes). According to the billing strategy of OpenAI, we only need to pay 1.09 USD when using DAINFER for API aliasing specification inference.

We also conduct the ablation study to demonstrate the benefit of neurosymbolic optimization. Specifically, the ablation DAINFER-EXHAUSIVE applies the two NLP models to all the APIs while the ablation DAINFER-TYPE applies the NLP models to the APIs satisfying the degree constraint and the validity constraint. As shown in Table 5.1, DAINFER-

TYPE invokes the LLM 5,164 times with 1,276,254 tokens in total and finishes analyzing all the subjects in 1,734.63 seconds. Besides, DAINFER-EXHAUSIVE has to apply the tagging models 58,846 times and invoke the LLM 8,090 times using 1,994,017 tokens, of which the whole process finishes in 2,844.26 seconds. The key reason for the differences between the ablations is that the solving steps at Lines 5 and 9 in Algorithm 4 can effectively reduce the numbers of applying the tagging model and the LLM, respectively, when the conjunctions of the constraints are unsatisfiable. Also, the token cost is naturally reduced when our algorithm invokes the LLM in a demand-driven manner. Compared to DAINFER-TYPE and DAINFER-EXHAUSIVE, DAINFER achieves the inference with 1.94× and 3.19× speed-ups. Hence, our neurosymbolic optimization efficiently supports the specification inference.

Answer to RQ1: DAINFER successfully derives 2,680 API aliasing specifications for 167 Java classes in 15 minutes, achieving a precision of 79.78% and a recall of 82.29% upon examined Java classes.

5.7.3 Comparison with Existing Techniques

We compare DAINFER with two most recent studies on API aliasing specification inference, i.e., ATLAS [22] and USPEC [23]. Besides, we construct another baseline, LLM-ALIAS, which feeds the library documentation to CHATGPT as input and generates API aliasing specifications via in-context learning. The few-shot examples used in the in-context learning are publically available [183].

***Comparison with* ATLAS.** We run the released tool ATLAS [184] upon the total 167 classes and finish the inference in 74.48 minutes. It is shown that the generated aliasing specifications only depict the potential aliasing relations between parameters and return values, while they all miss the pre-conditions under which such aliasing relations hold. For example, ATLAS only obtains that the return value of HashMap.get can be aliased with the second parameter of HashMap.put, missing the pre-condition over their first parameters. The restrictive templates used in the inference introduce the imprecision of the inferred specifications, which is also reported in the prior study [23]. Moreover, ATLAS fails to generate the specifications for 111 classes in the experimental subjects, such as android.os.Intent and android.os.Configuration. The root cause is that ATLAS fails to infer

the specifications when the creation of library function parameters is non-trivial, or the unit test execution demands a specific environment. Lastly, it is worth mentioning that the output of ATLAS is the library implementation derived from the execution of unit tests. Automatically converting it into the specifications defined in Definition 5.3.2 requires static analysis techniques, while the analyzers with different precision would yield different API aliasing specifications. Hence, we do not quantify the precision and recall of the inference results of ATLAS in a more fine-grained manner.

***Comparison with* USPEC.** USPEC is not open-sourced due to its commercial use [23], so we asked the authors for the raw data of their evaluation. According to their results, USPECS successfully obtains 124 API aliasing specifications upon 62 classes. Unfortunately, the precision of USPEC only reaches 66.1% (82/124). For instance, USPEC generates the incorrect aliasing specification $(\mathsf{HashMap.put}, \mathsf{HashMap.get}, \{(0,1)\}, 0)$ for the class java.util.HashMap. The root cause is that USPEC infers possible aliasing relations according to the usage events. However, the keys and values of HashMap objects may have the same types, making the inference algorithm unable to distinguish them with usage events only. We also quantify the recall of USPEC based on our labeled specifications in Section 5.7.2. It is shown that USPEC misses 370 API aliasing specifications. The recall of inferring API aliasing specifications is only 18.14%. The root cause of its low recall is that USPEC can only generate the aliasing specifications for the APIs that are used in the applications.

***Comparison with* LLM-ALIAS.** We compare DAINFER with LLM-ALIAS, which directly queries CHATGPT with the documentation. Its response of is a natural language sentence indicating an API aliasing specification. Due to laborious effort, we examine the inference results for 60 classes randomly selected in Section 5.7.2. The results show that LLM-ALIAS generates 801 API aliasing specifications for examined classes, only 113 of which are correct, achieving a precision of 14.11%(113/801) and a recall of 11.44%(113/988). Among 688 incorrect specifications, 60 specifications indicate the correct aliasing relations between parameters and return values, while they do not pose any restrictions over API parameters as the pre-conditions. The results show that vanilla LLMs without special designs have poor performance in understanding the concept of aliasing relation. In contrast, DAINFER achieves quite satisfactory precision and recall, which benefits from our insightful problem reduction and efficient neurosymbolic optimization.

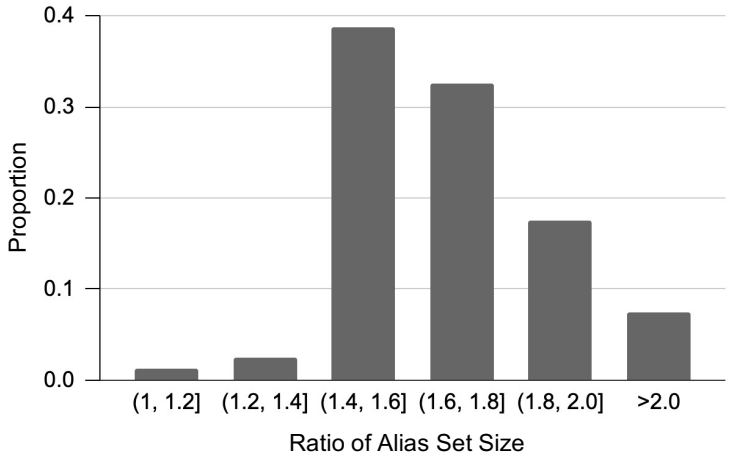


Figure 5.6: The results of alias analysis

Answer to RQ2: DAINFER achieves much higher precision and recall than USPEC and ATLAS, unleashing the power of LLMs in understanding library documentation.

5.7.4 Effects on Client Analysis

To demonstrate the impact of DAINFER on library aware analysis, we choose two fundamental clients, namely alias analysis and taint analysis, to quantify how it benefits downstream clients. Particularly, alias analysis is a fundamental pre-analysis of program optimization, while taint analysis support taint-style vulnerability detection to enhance the program reliability. We expect to demonstrate the impact of DAINFER in promoting the performance and reliability with the results of the two clients.

Effect on Alias Analysis. We conduct the field and context-sensitive alias analysis upon 15 Java projects based on existing studies [3, 154] with two settings. In the first setting, named Alias-Empty, we provide empty API specifications, i.e., discarding all the possible alias facts introduced by library API calls. In the second setting, named Alias-Infer, we apply the inferred correct API aliasing specifications to the alias analysis. We quantify the alias set sizes of the return values of library APIs under the two specifications.

Figure 5.6 shows the distribution of the ratios of alias set sizes. On average, the alias set sizes of the return values are increased by 80.05% with the benefit of our inferred specifications. The alias set sizes of 96.25% return values of the library API calls are increased

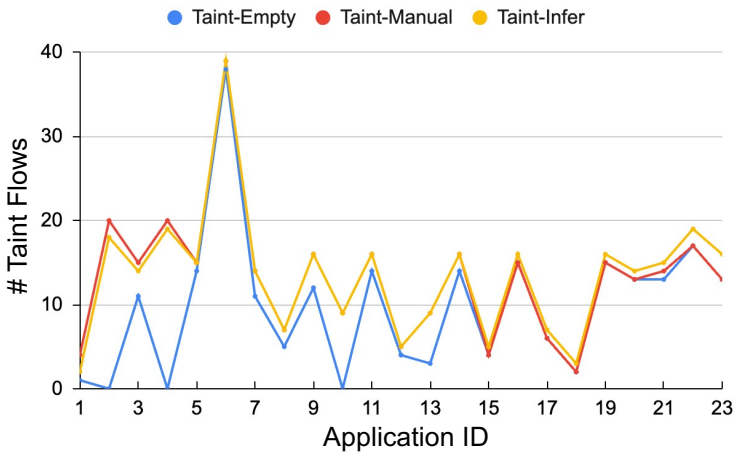


Figure 5.7: The results of taint analysis

by at least 40%. Because our pointer analysis is sound, the increase in the alias set size demonstrates that DAINFER promotes the alias analysis in discovering more alias facts in the applications using libraries.

Effect on Taint Analysis. We choose three different settings of specifications for FLOWDROID to conduct the taint analysis, namely Taint-Empty, Taint-Manual, and Taint-Infer. Here, Taint-Empty and Taint-Infer are similar to the two settings in the alias analysis. Under the setting Taint-Manual, we apply the manual specifications provided by FLOWDROID directly. We select 23 popular Android applications in F-Droid [185], which cover different program domains, including navigation, security, and messaging applications.

Figure 5.7 shows the numbers of the taint flows discovered under the three settings. Specifically, FLOWDROID discovers 225 taint flows under Taint-Empty, while it finds 304 taint flows under Taint-Manual. Notably, 79 out of 304 taint flows are induced by the aliasing relations among API parameters and returns. When we run FLOWDROID under Taint-Infer, it discovers 310 taint flows, 85 of which are discovered based on the correct API aliasing specifications inferred by DAINFER. There are six taint flows in three apps not discovered by FLOWDROID under the setting Taint-Infer due to false negatives of our inference algorithm. However, 12 taint flows discovered under Taint-Infer are not discovered under Taint-Manual. The results demonstrate that DAINFER promotes the taint analysis in discovering more taint flows.

Answer to RQ3: The inferred API aliasing specifications enable alias analysis to discover 80.05% more alias facts for the return values of library APIs and support taint analysis in finding 85 more taint flows upon the selected Android apps.

5.7.5 Discussion

Threats to Validity. Our work has three main threats to the validity. First, we select the classes evaluated in existing works and manual specifications in FLOWDROID, which can introduce the subject selection bias. However, the selected classes are general and cover commonly used ones. For example, the manual specifications in FLOWDROID cover the Android platform and Java standard library classes. Our evaluation can show the impact of our approach in understanding such fundamental classes. Second, following previous studies [23], we manually investigate the documentation and implementation to determine the ground truth for each class. Although we conduct the cross-checking according to the manual specifications provided by FLOWDROID, we still introduce the oracle bias, especially when several inferred specifications do not appear in the FLOWDROID benchmark. Third, DAINFER depends on an online generative AI service, of which the availability and stability can affect the performance and effectiveness of our approach. Particularly, the temperature setting of the two-staged prompting may affect the inference results. To measure the temperature sensitivity, we also quantify the precision and recall of DAINFER under different temperatures. The differences of the smallest and largest precision and recall are only 4.55% and 2.63%, respectively.

Limitations and Future Work. Our approach has several drawbacks that demand further improvements. First, DAINFER can not determine whether an API creates a new object. When the developers create any new objects, our inferred specifications can only depict data flow facts instead of aliasing relations. For example, DAINFER infers an API aliasing specification for java.util.Map that the return value of Map.computeIfPresent can be aliased with the second parameter of Map.put when their first parameters are aliased. This is a wrong specification as computeIfPresent returns null value or a newly computed value instead of any existing values stored in the fields. Second, the semantic unit consistency requires two strings to be equal. In our evaluation, however, we notice that sev-

eral semantic units are not the same strings while they indicate the same concept. For example, the first parameters of SparseArray.set and SparseArray.valueAt in the class android.util.SparseArray are key and index, respectively. The two different strings are actually the indicators of the same semantic concept. Hence, DAINFER can not infer the correct specification for the two APIs. Third, our documentation model is mainly applicable to Java libraries, as Java documentation can offer the four kinds of program properties explicitly. For other programming languages, such as Python, we can hardly obtain any class hierarchy relation in their documentation, which makes it impossible to over-approximate aliasing facts with the assistance of type information. Lastly, DAINFER depends on the effectiveness of the LLM used in the memory operation abstraction. Currently, we cannot figure out whether the LLM has seen the experimental data during the pre-training phase and demonstrate how DAINFER performs upon library documentation that is not used for pre-training the LLM.

To further improve DAINFER, we can explore several directions in the future. First, we can train a new word-embedding model to measure the similarity between different strings. It can hopefully support us in identifying the semantic units indicating the same concept even if they are not the same string. Second, it is promising to obtain a domain-specific LLM via fine-tuning. Specifically, we can leverage existing static analyzers to scan the source code of open-source libraries, obtain their memory operation kinds, and, particularly, determine whether the APIs create objects. Such offline models have better potential for interpreting the API semantic descriptions and making the neurosymbolic optimization break the efficiency bottleneck in interacting with the LLM.

5.8 Conclusion

We proposed a new approach DAINFER to infer API aliasing specifications from documentation. DAINFER adopts the tagging model and the large language model to interpret informal semantic information in the documentation and reduces the inference problem to an optimization problem, which can be efficiently solved by our neurosymbolic optimization algorithm. The inferred specifications are further fed to static analysis clients for analyzing the applications using libraries. Our evaluation demonstrated the high preci-

sion and recall of DAINFER in the inference, and also showed its significant impact in promoting the library-aware pointer analysis and taint analysis. The inferred specifications reveal the details of data manipulation conducted by library APIs, effectively assisting the enhancement of reliability and performance with better library understanding.

CHAPTER 6

VERIFYING DATA CONSTRAINT EQUIVALENCE IN FINTECH SYSTEMS

6.1 Introduction

With the development of E-commerce, FinTech systems have become increasingly essential to industrial production. As a typical kind of data-centric systems, they are composed of a cluster of database-backed applications manipulating large amounts of sensitive data [186]. Any incorrect data value can yield system misbehaviors and cause immeasurable financial losses. To ensure reliability, it is a common practice to specify target properties as data constraints [36, 35, 16] for data validation. If a data constraint is violated, developers receive an alert for further diagnosis.

Unfortunately, the continuous submissions from developers make data constraints accumulate rapidly and can even introduce redundancy. In a global FinTech company, Ant Group, 103 developers submitted 2,306 data constraints in the first quarter of 2022. Unaware of previous submissions, they create equivalent data constraints, which gradually become the technical debt [187], wasting computing resources and increasing the burden of system maintenance. To resolve the redundancy, the developers expect to search the existing equivalent data constraints before submitting new ones, thereby avoiding redundant submissions. Besides, quality assurance teams are eager to examine data constraint repositories regularly, seeking more opportunities for optimizing data validation based on the equivalence relation. Thus, it is relevant to verify the data constraint equivalence for performance enhancement of a FinTech system.

Goal and Challenges. We aim to design a decision procedure determining whether two data constraints are equivalent. However, it is stunningly challenging to find a solution fitting industrial requirements. First, the decision procedure should be highly efficient, as FinTech systems often contain tens of thousands of data constraints, which amplify the efficiency bottleneck greatly. Any inefficiency in the decision procedure can result

in significant burdens of adoption. Second, it is crucial to guarantee soundness and prove the equivalence as completely as possible. Otherwise, it would remove necessary data constraints or miss equivalent ones, resulting in financial losses or hiding opportunities for further optimization, respectively. In reality, data constraints can involve various data types, increasing the difficulty of achieving these objectives simultaneously.

Existing Effort. There have been two lines of research on equivalence verification. One line of the techniques leverages the specified rewrite rules and checks whether a program can be transformed to the other via *term rewriting* [188, 87]. Although the rewrite rules theoretically ensure soundness, they can only identify restrictive forms of equivalent patterns [70], and the vast search space of applying rewrite rules also brings great overhead [90]. The other line encodes the program semantics with logical formulas and performs the symbolic reasoning by invoking an SMT solver [75, 95, 189]. It provides a general approach to verify the equivalence, while an SMT solver is not efficient enough to reason a large number of data constraints. The solver has to be invoked thousands of times in the equivalence clustering and searching, accumulating the overhead and finally degrading the overall efficiency [190].

Insight and Solution. Our key idea originates from two critical observations. First, non-equivalent data constraints often contain different variables, literals, or operators. For example, the data constraint in Figure 6.1(a) examines the attributes *oid* and *in* in the table *t*, while the data constraint in Figure 6.1(b) examines the attributes *iid* and *new* instead. The lexical differences guide the generation of concrete values to make two data constraints evaluate differently. Second, equivalent data constraints often converge towards similar syntactic structures. For instance, the data constraints in Figure 6.1(d) and Figure 6.1(c) only differ in the orders of assertions, branches, and commutative operands after eliminating user-defined variables. The isomorphic syntactic structures are the witness of their equivalence. Thus, we can leverage the lexical differences and syntactic isomorphism to efficiently refute and prove the equivalence, respectively, avoiding unnecessary SMT solving for better performance.

Based on the insight, we present EQDAC, an efficient decision procedure for the equivalence verification. We establish a first-order logic (FOL) formula as the symbolic representation to depict the semantics. To refute the equivalence, we perform the *divergence*

```
s = 'IN';
if(contains(t.ty,s))
  assert(t.in > 0);
else
  assert(t.out > 0);
assert(t.amt > 0);
assert(t.oid != 0);
```

(a)

```
if(contains(t.ty,'IN')){
  assert(t.old == t.new - t.in);
} else {
  assert(t.old == t.new + t.out);
}
assert(t.oid != 0);
assert(t.iid != 0);
```

(b)

```
s = 'IN';
if(not contains(t.ty,s))
  assert(t.out > 0);
else
  assert(t.new > 0);
assert(t.amt > 0);
assert(t.iid != 0);
```

(c)

```
assert(t.iid != 0);
assert(t.oid != 0);
if(not contains(t.ty,'IN'))
  cash = t.out + t.new;
else
  cash = t.new - t.in;
assert(cash == t.old);
```

(d)

Figure 6.1: Examples of data constraints

analysis to explore the symbolic representations and generate the concrete values of variables, which simultaneously make one data constraint hold and the other violated. To prove the equivalence, we conduct the *isomorphism analysis* with a tree isomorphism algorithm [191] to examine whether the two symbolic representations can be transformed into each other by reordering the clauses and commutative terms. We combine the two analyses with the SMT solving, which determines the logical equivalence of the symbolic representations, finally obtaining a three-staged decision procedure.

We implement EQDAC and evaluate it upon a FinTech system in Ant Group, which maintains 30,801 data constraints in total. Leveraging EQDAC, we discover that 11,538 data constraints have at least one equivalent variant in the system, indicating that 7,842 data constraints are redundant. EQDAC finishes the equivalence clustering in three hours and achieves the equivalence searching in 1.22 seconds per data constraint. Except for the SMT solving, the stages of EQDAC can be proven to work in polynomial time. Benefiting from EQDAC, the CPU time can be reduced by 15.48% in the process of data validation. We also prove the soundness and completeness of EQDAC theoretically for a given syntax of data constraints. In summary, we make the following major contributions:

- We formulate the data constraint equivalence problem, which is critical for optimizing data validation in a FinTech system.
- We propose a sound and complete decision procedure EQDAC to efficiently support

the equivalence clustering and searching of data constraints.

- We implement EQDAC and evaluate it upon the data constraints in Ant Group, showing that it efficiently detects a significant number of equivalent data constraints.

In the rest of the chapter, the content is organized as follows. Section 6.2 introduces the background of data constraints in FinTech systems and motivates the problem of resolving equivalent data constraints. Section 6.3 demonstrates the key idea of our approach with a motivating example, which is followed by the formal statement of data constraint equivalence problem in Section 6.4. Sections 6.5 and 6.6 illustrates the technical details of our decision procedure. We present the implementation and evaluation details in Sections 6.7 and 6.8, respectively, and summarize the work in Section 6.9.

6.2 Background and Motivation

This section presents the background and highlights the motivation of our work.

6.2.1 Equivalent Data Constraints in FinTech Systems

FinTech systems, a typical kind of data-centric systems, usually consist of a cluster of database-backed applications manipulating large amounts of user data. To improve the system reliability, the developers often specify data constraints to describe target data properties and set up a data validation (DV) platform to examine them during the system runtime. Once a data constraint is violated, developers can receive detailed runtime information to guide further system diagnosis.

In reality, many development teams continuously submit data constraints to a central DV platform. For example, around 100 teams in Ant Group actively submit data constraints daily to the platform. Unaware of existing submissions, developers often submit data constraints equivalent to existing ones. Besides, the developers tend to be conservative about removing constraints, as they do not want to risk missing data errors. In this context, the DV platform examines equivalent data constraints redundantly, which causes unnecessary resource consumption, e.g., CPU time, disk IO, and network traffic. Finally,

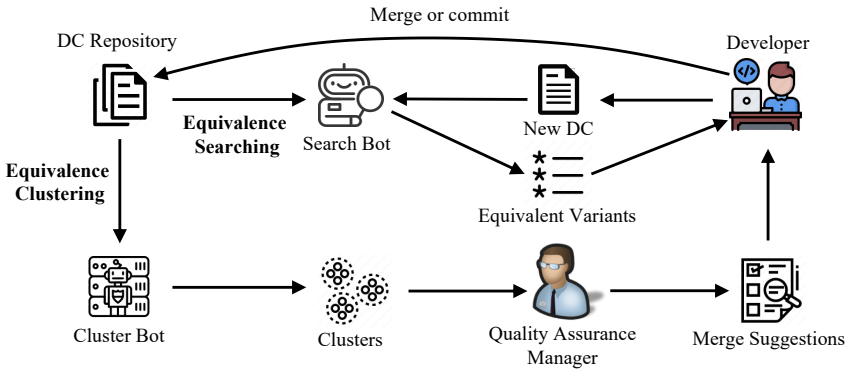


Figure 6.2: The workflow of equivalence searching and clustering

the accumulation of equivalent data constraints becomes the technical debt [187] of a FinTech system: Thus, it is crucial to tackle equivalent data constraints in the maintenance, which promotes resource-saving of a FinTech system.

6.2.2 Resolving Equivalent Data Constraints

To resolve the technical debt, the developers of Ant Group propose two demands, namely equivalence clustering and equivalence searching, to tackle equivalent data constraints. Specifically, they expect to integrate two bots into the CI/CD workflow [192] of a FinTech system as follows.

- *Equivalence searching*: A developer commits a new data constraint to the bot for searching existing equivalent variants. The list of equivalent variants assists the developer in deciding whether to merge it with any existing one. The workflow is shown in the upper part of Figure 6.2.
- *Equivalence clustering*: A quality assurance (QA) manager exports all the data constraints to the bot, which divides the data constraints into equivalence clusters. Then, the QA manager summarizes merge suggestions and sends them to developers for further confirmation. The lower part of Figure 6.2 shows the workflow of equivalence clustering.

Generally, the two bots resolve the redundancy from two perspectives, respectively. First, the equivalence searching conducts the instant checking of newly-submitted data

constraints, enabling the developers to avoid redundancy if possible. Second, the equivalence clustering supports the nightly scan of the whole repository of data constraints. The QA managers can inspect the clustering information to find opportunities for merging equivalent ones. During the development cycle, the two bots can serve as two lines of defense for redundancy issues in the CI/CD workflow.

To automate the overall workflow, we need an efficient decision procedure to verify whether two data constraints are equivalent. Specifically, the two bots would invoke the decision procedure to determine the equivalence in the clustering and searching, respectively. In this work, we aim to design an effective solution for verifying data constraint equivalence, and promoting two clients with our decision procedure.

6.3 EqDAC in a Nutshell

This section presents a motivating example to show our insight (Section 6.3.1) and outlines our decision procedure (Section 6.3.2).

6.3.1 Motivating Examples

Verifying the data constraint equivalence is non-trivial in industrial scenarios. First, the cost of the decision procedure can accumulate significantly due to the vast number of data constraints [15]. Second, the decision procedure can prune necessary data constraints or miss equivalent ones if it is not sound or complete, increasing the risk of data security and hiding the opportunity for optimization. Thus, we need to simultaneously ensure the soundness, completeness, and efficiency of the decision procedure.

Figure 6.1 shows four data constraints as examples. Specifically, the data constraints in Figure 6.1(a) and Figure 6.1(b) depict the properties where three attributes of the table t have positive values in two cases, and the values of *oid* and *iid* are not 0, respectively. Besides, the data constraints in Figure 6.1(d) and Figure 6.1(c) describe the property where the changes to the account balances are equal to the transferred cash amount, and the ids of the two accounts, i.e., `iid` and `oid`, are not 0. According to the examples, we can obtain the following two important observations:

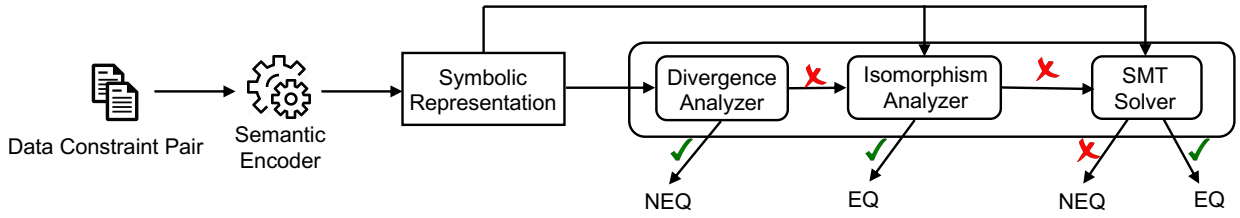


Figure 6.3: Schematic overview of our decision procedure EQDAC

- Non-equivalent data constraints tend to have different lexical tokens, such as database attributes, literals, and operators. For example, the data constraints in Figure 6.1(a) and Figure 6.1(b) examine different attributes. It is likely to generate the values of table attributes, making them evaluate differently.

- Equivalent data constraints often only differ in the orders of commutative operands and independent statements after eliminating user-defined variables. For instance, the data constraints in Figure 6.1(c) and Figure 6.1(d) share the isomorphic syntactic structure, which implies their equivalence.

Based on the observations, we realize that the lexical differences and syntactic isomorphism enable us to efficiently refute and prove the equivalence, respectively. If we generate "good" concrete values making two data constraints evaluate differently or find the isomorphism between the syntactic structures, we can avoid SMT solving and achieve high efficiency.

6.3.2 Outline of Decision Procedure

According to our insight, we design an efficient, sound, and complete decision procedure for verifying data constraint equivalence. To depict the data constraint semantics, we propose the semantic encoding to construct a FOL formula in a restrictive form as its symbolic representation, which eliminates user-defined variables (e.g., the variable *cash* in Figure 6.1(c)). Based on the symbolic representations, our decision procedure works in three stages, as shown in Figure 6.3.

- The divergence analysis explores the symbolic representations with the guidance of lexical differences, aiming to generate concrete values that make data constraints

evaluate differently. For example, it explores the clause induced by the last assertion in Figure 6.1(a) and assigns 0 to the attribute *oid* to violate the assertion. Also, it concretizes the variables in Figure 6.1(b) to make the data constraint satisfied.

- The isomorphism analysis constructs the parse trees of the symbolic representations and examines whether the parse trees are isomorphic. The analysis abstracts away the order of commutative constructs, such as independent statements and commutative operands. For example, it discovers the isomorphic structures in Figure 6.1(d) and Figure 6.1(c), blurring the orders of assertions and the operands of $+$ and $==$.
- If the first two analyses can not refute or prove the equivalence, we invoke an SMT solver to check the logical equivalence of the symbolic representations. To ensure soundness and completeness, we perform the SMT encoding with a decidable fragment in the combined theory of bit-vector, floating-point arithmetic, and string.

Apart from soundness and completeness, EQDAC also features a theoretical guarantee of complexity. The symbolic representation construction, the divergence analysis, and the isomorphism analysis can work in polynomial time to the size of the abstract syntax tree of a data constraint. Our evaluation also provides strong evidence of the EQDAC's high efficiency in the equivalence clustering and searching.

6.4 Problem Formulation

This section presents the syntax (Section 6.4.1) and formulates the data constraint equivalence problem (Section 6.4.2).

6.4.1 Data Constraint Syntax

Figure 6.4 summarizes the syntax. A *variable* is a data variable $v_d \in \mathcal{V}_d$ indicating the value of a table attribute, or an user-defined variable $x \in \mathcal{V}_u$ storing the value temporally. Its value can be a finite-length integer, a floating point number, or a string. A *literal* is a constant value. An *arithmetic expression* can be a literal, a data variable, or a compound arithmetic expression. A *comparison expression* compares arithmetic expressions and user-defined variables, or examines the strings with predicates $p \in \mathcal{P}$. A *Boolean expression* is

$$
\begin{aligned}
&\mathcal{V} := v_d \mid x \\
&\mathcal{L} := \{l_i \mid i \geqslant 1\} \\
&\mathcal{A} := l \mid v_d \mid a_1 \oplus a_2 \\
&\mathcal{C} := a_1 \odot a_2 \mid x_1 \odot x_2 \mid a \odot x \mid x \odot a \mid p(v, l) \mid p(v_1, v_2) \\
&\mathcal{B} := c \mid b_1 \textbf{ and } b_2 \mid b_1 \textbf{ or } b_2 \mid \textbf{not } b \mid \textbf{ite}_b(c_0, b_1, b_2) \\
&\mathcal{S} := x = a \mid \textbf{assert}(b) \mid s_1; s_2 \mid \textbf{ite}_s(c_0, s_1, s_2) \\
&\mathcal{R} := s+ \\
&\oplus := + \mid - \mid \times \mid \div \\
&\odot := > \mid < \mid \geqslant \mid \leqslant \mid == \mid \neq \\
&\mathcal{P} := \{\text{prefixOf, suffixOf, contains, equals}\}
\end{aligned}
$$

Figure 6.4: The syntax of data constraints

a comparison expression or a compound expression with logical connectives. A *statement* is an assignment, an assertion, a sequencing, or an $\textbf{ite}_s$ statement. Particularly, the conditions in $\textbf{ite}_b$ expressions and $\textbf{ite}_s$ statements only relate to data variables. Finally, a *data constraint* consists of finite statements. All its assertions are expected to hold for given database tables.

The syntax is expressive enough to specify target properties in real-world scenarios. It covers all the patterns in [35], such as value comparison, conditional comparison, etc. Also, user-defined variables support writing data constraints flexibly. Arithmetic operations and string predicates support expressing complex properties, e.g., comparing the sums of cash amounts and matching between string variables.

6.4.2 Data Constraint Equivalence Problem

Before stating the problem, we first introduce the notions of interpretation and semantic equivalence as follows.

Definition 6.4.1 (Interpretation and Model) *An interpretation* I *maps each data variable* v_d *to a value in its domain.* I *is a model of a data constraint* r*, denoted by* $I \models r$*, if all the assertions hold under* I*.*

Example 6.4.1 *The following interpretation* I *is a model of the data constraint in Figure 6.1(a).*

$$I = [t.ty \mapsto \text{'IN'}, t.in \mapsto 1, t.out \mapsto 0, t.oid \mapsto 1, t.amt \mapsto 1]$$

An interpretation indicates the values of table attributes. A data constraint induces a set of interpretations making its assertions hold. Formally, we define the semantic equivalence.

Definition 6.4.2 (Semantic Equivalence) *The data constraints* r_1 *and* r_2 *are semantically equivalent, denoted by* $r_1 \simeq r_2$*, if and only if*

$$\forall I : I \models r_1 \Leftrightarrow I \models r_2$$

Example 6.4.2 *Based on Example 6.4.1, we can construct*

$$I' = I[t.new \mapsto 0, t.iid \mapsto 1]$$

I' *is not a model of the data constraint in Figure 6.1(b), while it is a model of the data constraint in Figure 6.1(a), indicating that they are not semantically equivalent.*

In this work, we aim to propose a decision procedure to verify whether r_1 is semantically equivalent to r_2 for a given data constraint pair (r_1, r_2). However, finding a sound, complete, and efficient solution is challenging. Theoretically, any instance of SAT problem [193] can be reduced to an instance of our problem by constructing two proper data constraints in polynomial time. Formally, we state the complexity barrier of our problem as follows.

Theorem 6.4.1 (Complexity Barrier) *Data constraint equivalence problem is NP-hard.*

Proof. We only need to prove that we can reduce any instance of the SAT problem to an instance of the data constraint equivalence problem in polynomial time.

Consider an arbitrary propositional logic formula ψ, which contains n variables denoted by a_i $(1 \leqslant i \leqslant n)$. We first construct a database table with n attributes, namely v_i, where $1 \leqslant i \leqslant n$. Meanwhile, we introduce a constant set containing n unique constants, denoted by $L = \{\ell_1, \ell_2, \cdots, \ell_n\}$. We then construct the assertion statement **assert**$(f(v_1, v_2, \cdots, v_n))$. Here the boolean expression $f(v_1, v_2, \cdots, v_n)$ is constructed by replacing a_i with the equality constraint $v_i == \ell_i$ in ψ. Such the assertion statement is exactly the data constraint r we want. Obviously, ψ is unsatisfiable if and only if r is semantically equivalent to

$$\textbf{assert}(f(v_1, v_2, \cdots, v_n) \textbf{ and } \neg f(v_1, v_2, \cdots, v_n))$$

The reduction can be achieved in linear time to the size of the formula ψ. Therefore, the data constraint equivalence problem is NP-hard. Q.E.D.

Roadmap. To verify the equivalence, we propose a symbolic representation to encode the semantics (Section 6.5) and design an efficient decision procedure (Section 6.6). Particularly, we introduce light-weighted reasoning to refute and prove the equivalence efficiently, which is our main technical contribution. By fusing our light-weighted reasoning with SMT-based analysis, our decision procedure features soundness and completeness, and achieves high efficiency in supporting the equivalence clustering and searching.

6.5 Semantic Encoding

This section introduces the symbolic representation to depict the semantics (Section 6.5.1), presents the symbolic evaluation (Section 6.5.2), and summarizes the benefit at the end (Section 6.5.3).

6.5.1 Symbolic Representation

A data constraint is essentially a program with data variables as inputs. The values of data variables determine the values of all the variables and expressions. Based on the intuition, we propose the concepts of symbolic terms and conditions to depict the values of variables and expressions.

Definition 6.5.1 (Symbolic Term) *A symbolic term τ represents the value of a variable or a literal in either of the forms:*

- $\tau := v_d$ *or* $\tau := l$ *is a data variable or a literal, respectively.*
- $\tau := \tau_1 \oplus \tau_2$ *is a compound term with an arithmetic operator.*

Definition 6.5.2 (Symbolic Condition) *A symbolic condition ϕ is a FOL formula in one of the following forms:*

- *An atomic condition is an arithmetic comparison of two symbolic terms or a string comparison, i.e.,* $\phi := \tau_1 \odot \tau_2$ *or* $\phi := p(\tau_1, \tau_2)$*, where* $p \in \mathcal{P}$ *is a string predicate.*

- *A compound condition is a FOL formula with logical connectives, i.e.,* $\phi := \phi_1 \wedge \phi_2$, $\phi := \phi_1 \vee \phi_2$, *or* $\phi := \neg\phi_0$.

Example 6.5.1 *In Figure 6.1(c), the values of* cash *can be represented by the terms* $\mathtt{t.out} + \mathtt{t.new}$ *and* $\mathtt{t.new} - \mathtt{t.in}$. *The condition of the* $\boldsymbol{ite}_s$ *statement is encoded by* $\neg$`contains`$(\mathtt{t.ty}$, *'IN'*$)$.

The symbolic terms represent the values of variables, literals, and arithmetic expressions, while the symbolic conditions encode the values of Boolean expressions, providing the ingredient for defining the symbolic representations.

Definition 6.5.3 (Symbolic Representation) *For a data constraint* r, *its symbolic representation is a symbolic condition* φ *satisfying*

- *For any interpretation* I, $I \models r$ *if and only if* $I \models \varphi$.
- *The negations only occur before string atomic constraints.*

Intuitively, the symbolic representation encodes the semantics faithfully with a FOL formula, which only relates to data variables and exclude redundant negations. It abstracts away user-defined variables and blurs syntactic differences in terms of negations effectively, enabling us to design light-weighed reasoning for equivalence verification. In what follows, we show how to construct the symbolic representation in detail.

6.5.2 Symbolic Evaluation

Now we propose the symbolic evaluation to construct the symbolic representation. Basically, the symbolic evaluation consists of two stages, which collects the values of Boolean expressions in each assertion, and eliminates unnecessary negations, respectively. Before delving into details, we first introduce the notion of the symbolic state.

Definition 6.5.4 (Symbolic State) *Given a data constraint* r, *the symbolic state* $\mathbf{S}$ *at program location* ℓ *is* $(\mathbf{E}, \Phi)$, *where*

- *An* environment $\mathbf{E}$ *maps a variable* v *or an arithmetic expression* e *to a term-condition pair set* $\{(\tau, \phi)\}$, *indicating that* v *or* e *evaluates to the same value of* τ *when* ϕ *holds.*

- *A* property Φ *is a symbolic condition that summarizes the assertions in* $\mathfrak{r}$ *before the program location* ℓ*.*

Example 6.5.2 *After the first assertion in Figure 6.1(c), we have*

$$\mathbf{E} = [\mathrm{t.iid} \mapsto \{(\mathrm{t.iid}, \mathsf{T})\},\ 0 \mapsto \{(0, \mathsf{T})\}] \quad \Phi = (\mathrm{t.iid} \neq 0)$$

Now we present the technical details of the symbolic evaluation. In the first stage, we evaluate the variables and expressions to obtain a FOL formula depicting the semantics, which only relates to the data variables. Specifically, we define the evaluation rules in Figure 6.5 and Figure 6.6.

- The rule `ASSIGN` evaluates the RHS with the rules `VAR` and `AE` in Figure 6.6, and applies the strong update to $\mathbf{E}$, enforcing the user-defined variable ν and the expression $\mathfrak{a}$ have the same value. It successfully evaluates user-defined variables, making the symbolic terms only relate to the data variables.
- The rule `ASSERT` evaluates the Boolean expression $\mathfrak{b}$ to a symbolic condition ψ. It then connects ψ and the original property Φ with a logical conjunction. This, in turn, forms a property that accumulates the conditions of the assertions.
- The rules `SEQ` and `ITE-S` are defined straightforwardly. `SEQ` applies the evaluation rules of two components sequentially. `ITE-S` evaluates the two cases separately and joins two symbolic states according to the branch condition.

We omit the rules of evaluating string comparisons and other compound Boolean expressions due to limited space, which are similar to the rules `ACmp` and `ITE-E`. Based on the rules, we evaluate a data constraint stepwise. Initially, the symbolic state is a pair of empty mapping and a *true* value. By applying the rule of each statement along control flow paths, we obtain the symbolic state at each program location and finally summarize all the assertions with the property Φ_e at the exit, which depicts the semantics of the data constraint.

$$\text{ASSIGN}\frac{\mathbf{E} \vdash_e a \rightsquigarrow V \quad \mathbf{E}' = \mathbf{E}[v \mapsto V]}{\mathbf{E}, \Phi \vdash v = a \rightsquigarrow \mathbf{E}', \Phi}$$

$$\text{ASSERT}\frac{\mathbf{E} \vdash_b b \rightsquigarrow \psi \quad \Phi' = \Phi \wedge \psi}{\mathbf{E}, \Phi \vdash \textbf{assert}(b) \rightsquigarrow \mathbf{E}, \Phi'}$$

$$\text{SEQ}\frac{\mathbf{S} \vdash s_1 \rightsquigarrow \mathbf{S}_1 \quad \mathbf{S}_1 \vdash s_2 \rightsquigarrow \mathbf{S}'}{\mathbf{S} \vdash s_1; s_2 \rightsquigarrow \mathbf{S}'}$$

$$\text{ITE-S}\frac{\begin{array}{c}\mathbf{E} \vdash_b c_0 \rightsquigarrow \gamma_1 \quad \gamma_2 = \neg\gamma_1 \quad \mathbf{E}, \Phi \vdash s_i \rightsquigarrow \mathbf{E}_i, \Phi_i \\ \mathbf{E}' = [u \mapsto \bigcup_{i=1}^{2}\{(\tau_i, \phi_i \wedge \gamma_i) | (\tau_i, \phi_i) \in \mathbf{E}_i(u)\}]\end{array}}{\mathbf{E}, \Phi \vdash \textbf{ite}_s(c_0, s_1, s_2) \rightsquigarrow \mathbf{E}', \text{ite}(\gamma_1, \Phi_1, \Phi_2)}$$

Figure 6.5: Evaluation rules of statements

$$\text{VAR}\frac{u \in \mathcal{L} \cup \mathcal{V}_d \quad U = \{(u, \mathsf{T})\}}{\mathbf{E} \vdash_e u \rightsquigarrow U}$$

$$\text{AE}\frac{\begin{array}{c}a_i \in \mathcal{A} \quad \mathbf{E} \vdash_e a_i \rightsquigarrow U_i \\ A = \{(t_1 \oplus t_2, \phi_1 \wedge \phi_2) \mid (t_i, \phi_i) \in U_i\}\end{array}}{\mathbf{E} \vdash_e a_1 \oplus a_2 \rightsquigarrow A}$$

$$\text{ACmp}\frac{\begin{array}{c}u_i \in \mathcal{A} \cup \mathcal{V}_u \quad \mathbf{E} \vdash_e u_i \rightsquigarrow U_i \\ B = \{(t_1 \odot t_2) \wedge \phi_1 \wedge \phi_2 \mid (t_i, \phi_i) \in U_i\}\end{array}}{\mathbf{E} \vdash_b u_1 \odot u_2 \rightsquigarrow \bigvee_{\phi \in B} \phi}$$

$$\text{ITE-E}\frac{\mathbf{E} \vdash_b c_0 \rightsquigarrow \gamma_0 \quad \mathbf{E} \vdash_b b_i \rightsquigarrow \gamma_i}{\mathbf{E} \vdash_b \textbf{ite}_b(c_0, b_1, b_2) \rightsquigarrow (\gamma_1 \wedge \gamma_0) \vee (\gamma_2 \wedge \neg\gamma_0)}$$

Figure 6.6: Helper rules evaluating expressions

Example 6.5.3 *Consider the data constraint in Figure 6.1(c). We obtain* $\Phi = \phi_1 \wedge ((\phi_2 \wedge \phi_4) \vee (\phi_3 \wedge \neg\phi_4))$ *at its exit, where*

$$\phi_1 = (t.iid \neq 0) \wedge (t.oid \neq 0) \; \phi_2 = (t.out + t.new = t.old)$$

$$\phi_3 = (t.new - t.in = t.old) \;\; \phi_4 = \neg \mathit{contains}(t.ty, \text{`}IN\text{'})$$

In the second stage, we eliminate the negations in Φ_e that do not apply to atomic string constraints. Technically, we first transform Φ_e into the negation normal form (NNF), in which the negation applies only to atomic formulas. Then, we eliminate the negation before each atomic arithmetic constraint by changing the comparison operator, e.g., transforming $\neg(t.a \geqslant t.b)$ to $t.a < t.b$. Notably, the above transformations can be achieved by the breadth-first search upon the parse tree of Φ_e, where the symbolic representation is constructed on the fly. The overall time complexity is linear to the size of Φ_e.

Example 6.5.4 *In Example 6.5.3, we eliminate the negations and get the symbolic representation* $\varphi = \phi_1 \wedge ((\phi_2 \wedge \phi_4) \vee \phi')$, *where* $\phi' = (\mathtt{t.new} - \mathtt{t.in} = \mathtt{t.old}) \wedge \mathtt{contains}(\mathtt{t.ty}, \text{'IN'})$.

6.5.3 Summary

The symbolic representation is essentially a Boolean function of data variables, featuring the following three benefits:

- The symbolic representations preserve the lexical differences in terms of data variables, literals, and operators, which can indicate the possible non-equivalence.
- The symbolic evaluation evaluates the user-defined variables, abstracting away the difference in terms of their names, which do not affect the semantics.
- The elimination of unnecessary negations normalizes the FOL formulas and yields isomorphic symbolic representations for more equivalent data constraints.

Thus, the semantic encoding exposes lexical differences and syntactic isomorphism for light-weight reasoning, which efficiently refutes and proves the equivalence (Section 6.6.1 and Section 6.6.2).

6.6 Decision Procedure

In this section, we first introduce the divergence analysis (Section 6.6.1) and isomorphism analysis (Section 6.6.2) for efficiently refuting and proving the equivalence, respectively. We then combine the two analyses with SMT solving to establish the decision procedure (Section 6.6.3). In what follows, we denote the data constraints by r_1 and r_2 and their symbolic representations by φ_1 and φ_2 for demonstration.

6.6.1 Divergence Analysis

Based on Definition 6.5.3, φ_1 and φ_2 depict the semantics of two data constraints faithfully. We can safely refute the equivalence if there exists an interpretation I making them

evaluate to different truth values. However, it is non-trivial to obtain such a desired interpretation efficiently. The random sampling may hit a desired interpretation successfully after failing many attempts, which can degrade the efficiency significantly. To resolve the problem, we attempt to explore specific Boolean structures of φ_1 and φ_2 and concretize the data variables within the structures. Formally, we introduce the *degrees of freedom* to guide the exploration.

Definition 6.6.1 (Degrees of Freedom) *For two symbolic representations φ_1 and φ_2, the degrees of freedom of a clause ϕ occurring in φ_1 is*

$$\mathcal{DF}(\phi \mid \varphi_1, \varphi_2) = \frac{1}{h(\phi)} \cdot \sum_{M \in \{V_d, L, O\}} |M(\phi) \setminus M(\varphi_2)|$$

$h(\psi)$ is the height of the parse tree of ψ. $V_d(\psi)$ contains the data variables in ψ but excludes arithmetic operands. $L(\psi)$ and $O(\psi)$ contain the literals and operators in ψ, respectively.

Intuitively, a larger degrees of freedom indicates a higher possibility of making ϕ evaluate to a target truth value:

- First, a smaller value of $h(\phi)$ indicates the opportunity of finding the desired interpretation with fewer explorations.
- Second, a larger value of $|M(\phi) \setminus M(\varphi_2)|$ indicates that ϕ has more unique lexical tokens absent in φ_2. The concretizations of data variables in ϕ and φ_2 are less intertwined.
- Third, $V_d(\phi)$ excludes arithmetic operands, as arithmetic operations can increase the difficulty of concretization.

Example 6.6.1 *Consider the data constraints in Figure 6.1(a) and Figure 6.1(b). According to Section 6.5, their symbolic representations are*

$$\varphi_1 = ((t.in > 0 \wedge \phi_c) \vee (t.out > 0 \wedge \neg\phi_c)) \wedge \phi_a \wedge \phi_o$$

$$\varphi_2 = ((t.out > 0 \wedge \neg\phi_c) \vee (t.new > 0 \wedge \phi_c)) \wedge \phi_a \wedge \phi_i$$

where $\phi_a = (t.amt > 0)$, $\phi_o = (t.oid \neq 0)$, $\phi_i = (t.iid \neq 0)$, and $\phi_c =$ `contains`$(t.ty, \text{'IN'})$. Let ϕ denote the first clause of φ_1. We have $V_d(\phi) \setminus V_d(\varphi_2) = \{t.in\}$, $L(\phi) \subseteq L(\varphi_2)$, and

Algorithm 5: Divergence analysis

Input: φ_1, φ_2: Two symbolic representations;
Output: Whether $\exists \mathrm{I} : \neg(\mathrm{I} \models \varphi_1 \leftrightarrow \mathrm{I} \models \varphi_2)$

```
1  foreach (φ1, φ2) ∈ {(φ1, φ2), (φ2, φ1)} do
2      I ← ⊥;
3      status ← T;
4      explore(φ1, φ1, φ2, F);
5      explore(φ2, φ2, φ1, T);
6      if status is T then
7          return true;
8  return unknown;
9  Procedure explore(φ, φ, φ′, tv)
10     if status then
11         if φ is atomic then
12             if FreeVar(φ, I) ≠ ∅ then
13                 I ← concretize(φ, tv);
14             else
15                 status ← check(I ⊨ φ = tv);
16         else if (LC(φ), tv) ∈ {(∧, T), (∨, F)} then
17             foreach φi ∈ C(φ) do
18                 explore(φi, φ, φ′, tv);
19         else
20             φ′ ← arg max_{φi∈C(φ)} DF(φi | φ, φ′);
21             explore(φ′, φ, φ′, tv);
```

$\mathrm{O}(\phi) \subseteq \mathrm{O}(\varphi_2)$ *Thus, we have* $\mathcal{DF}(\phi \mid \varphi_1, \varphi_2) = \frac{1}{3}$. *Similarly, we have* $\mathcal{DF}(\phi_a \mid \varphi_1, \varphi_2) = 0$ *and* $\mathcal{DF}(\phi_o \mid \varphi_1, \varphi_2) = 1$.

Based on the degrees of freedom, we propose the divergence analysis to generate a desired interpretation. Algorithm 5 shows its technical details. It receives two symbolic representations φ_1 and φ_2 and attempts to generate a desired interpretation enforcing them evaluate differently (lines 1–7). The function *explore* traverses the clauses level by level (lines 9–21), handling three kinds of clauses ϕ with specific strategies:

- If ϕ is atomic, we concretize the free variables to make ϕ evaluate to tv (lines 12–13). If there is no free variable, we check whether ϕ evaluates to tv under I (line 15).
- If ϕ is a connected with $\wedge$ and tv is *true*, or ϕ is connected with $\vee$ and tv is *false*, we explore all the clauses in ϕ and enforce them evaluates to tv (lines 16–18).

- Otherwise, we select the clause ϕ' with the maximal degrees of freedom and enforce it evaluate to tv (lines 20–21).

If each clause evaluates to the target value, Algorithm 5 finds the desired interpretation, thereby refuting the equivalence.

Example 6.2.2 *In Example 6.6.1, φ_1 is connected with the logical conjunction. We only need to select and explore one of its clauses if we want to make φ_1 evaluate to false. The third clause ϕ_o has a larger degrees of freedom than the other two, so we select it and assign 0 to* `t.oid`*. Similarly, we can enforce φ_2 evaluate to* true*, which finally refutes the equivalence.*

Lastly, it is worth mentioning that the data constraints with different lexical tokens are often non-equivalent, while it is unsound to refute the equivalence directly based on lexical differences. In contrast, our divergence analysis essentially utilizes lexical differences to guide the interpretation generation, which supports refuting the equivalence soundly.

6.6.2 Isomorphism Analysis

As the FOL formulas, the symbolic representations φ_1 and φ_2 are logically equivalent if we can transform φ_1 to φ_2 by reordering commutative sub-formulas and terms in φ_1. For example, the evaluation of a FOL formula does not depend on the order of the clauses connected with the logical disjunction and conjunction. Also, any permutation of the operands of commutative arithmetic operators, such as addition and multiplication, always yields the logically equivalent formula. In other words, we can prove the data constraint equivalence safely by identifying the isomorphism between φ_1 and φ_2.

Based on the above key idea, we propose the isomorphism analysis to determine whether the parse trees of φ_1 and φ_2 are isomorphic, which is formulated in Algorithm 6. Using the AHU algorithm [191] for tree isomorphism checking, Algorithm 6 proves the data constraint equivalence if the parse trees are isomorphic (lines 1–2). Particularly, the functions *SCT* and *STT* process the clauses and terms of a symbolic representation in a top-down manner, respectively, creating tree nodes and leaf nodes in the parse tree.

- When processing a non-atomic formula φ, *SCT* creates a tree node to store the logical connective, and appends all the parse trees of its clauses (lines 5–6).

Algorithm 6: Isomorphism analysis

Input: φ_1, φ_2: Two symbolic representations;
Output: Whether $\forall I : I \models \varphi_1 \leftrightarrow I \models \varphi_2$

```
1  if AHUcheck(SCT(φ1), SCT(φ2)) then
2  |  return true;
3  return unknown;
4  Procedure SCT(ϕ)
5  |  if ϕ : ϕ1 ⊛ ··· ⊛ ϕk and ⊛ ∈ {∧, ∨, ¬} then
6  |  |  return Tree(⊛, {SCT(ϕi) | 1 ⩽ i ⩽ k});
7  |  else if ϕ : τ1 ⊛ τ2 or ϕ : ⊛(τ1, τ2) then
8  |  |  if ⊛ ∈ {=, ≠, equals} then
9  |  |  |  return Tree(⊛, {STT(τ1), STT(τ2)});
10 |  |  else if ⊛ ∈ {<, ⩽} then
11 |  |  |  ⊛′ ← flip(⊛);
12 |  |  |  return Leaf(⊛′, STT(τ1), STT(τ2));
13 |  |  else
14 |  |  |  return Leaf(⊛, STT(τ1), STT(τ2));
15 Procedure STT(τ)
16 |  if Op(τ) = {⊛} and ⊛ ∈ {+, ∗} then
17 |  |  return Tree(⊛, Operand(τ));
18 |  else if τ : τ1 ⊕ τ2 then
19 |  |  return Leaf(⊕, STT(τ1), STT(τ2));
20 |  else
21 |  |  return Leaf(τ);
```

- For an atomic condition, *SCT* creates a tree node if the comparison operator is in $\{=, \neq\}$ or the string predicate is *equals* (lines 8–9). Otherwise, it adds a leaf node to make sub-trees nonexchangeable (lines 10–14). Notably, it normalizes inequalities to enforce them using $>$ and $\geqslant$ only, which supports discovering more equivalent inequalities.
- *STT* constructs a tree node if a term τ only uses addition or multiplication (lines 16–17). For other cases, *STT* creates a leaf node (lines 18–21).

Example 6.6.3 *Figure 6.7(a) and Figure 6.7(b) show the parse trees of the symbolic representations for the data constraints in Figure 6.1(d) and Figure 6.1(c), respectively.* φ^* *represents* `contains`(t.ty, *'IN'*). *Their isomorphism proves the data constraint equivalence.*

It is worth noting that the AHU algorithm in Algorithm 6 is slightly different from the standard one [191]. Originally, the AHU algorithm sorts the sub-trees by level, as the

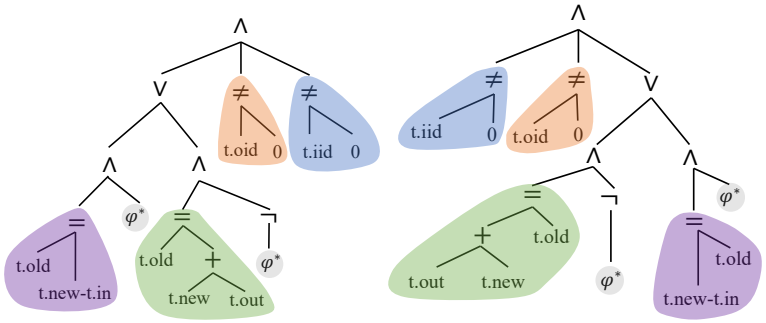


Figure 6.7: Two isomorphic parse trees

orders of the sub-trees do not matter. In our case, however, only the sub-trees of tree nodes can be arbitrarily permuted. Thus, we modify the AHU algorithm to adapt it to the isomorphism analysis, not sorting the sub-trees of each leaf node.

6.6.3 Equivalence Verification with EqDAC

Combining the above analyses with the SMT solving, we obtain the decision procedure EqDAC in Algorithm 7. We first construct the symbolic representations φ_1 and φ_2 via the semantic encoding. If we can not refute or prove the equivalence with the first two analyses (lines 2–5), an SMT solver examines whether φ_1 and φ_2 are logically equivalent (line 6) for general cases. Notably, the function *Divergent* invokes Algorithm 5 at the line 2, and explores the Boolean structures of φ_1 and φ_2 at most two times, ensuring the efficiency of the first stage.

The divergence analysis and isomorphism analysis over and under-approximate the equivalence, respectively. Although the analyses do not always determine the equivalence, they can handle a large proportion of data constraints in practice, evidenced by our evaluation. Formally, we state two theorems to formulate the theoretical guarantee.

Theorem 6.6.1 (Time Complexity) *The steps in Algorithm 7 before line 6 run in polynomial time to* N*, where* N *is the upper bound of the numbers of AST nodes for the two data constraints.*

Before proving Theorem 6.6.1, we first propose and prove Lemma 6.6.1 and Lemma 6.6.2 as follows.

Algorithm 7: Decision procedure

Input: r_1, r_2: Two data constraints;
Output: Whether $r_1 \simeq r_2$ or not

```
φ1, φ2 ← getSymReps(r1, r2);
if Divergent(φ1, φ2) is true then
    return false;
if Isomorphic(φ1, φ2) is true then
    return true;
return (SMT-Solve(¬(φ1 ↔ φ2)) is UNSAT);
```

Lemma 6.6.1 *We define the size of a symbolic condition ϕ as follows:*

$$\delta(\phi) = \begin{cases} 1 & \phi \text{ is atomic} \\ 1 + \delta(\phi_0) & \phi = \neg\phi_0 \\ \delta(\phi_1) + \delta(\phi_2) & \phi = \phi_1 \vee \phi_2 \text{ or } \phi = \phi_1 \wedge \phi_2 \end{cases}$$

Given any data constraint r, denote the node number of its abstract syntax tree by N. The size of its symbolic representation $\delta(\varphi)$ is polynomial to N.

Proof of Lemma 6.6.1. For clarity, we introduce two functions α and β:

- $\alpha(\mathbf{E}, e)$ is the number of terms that e may be equal to.
- $\beta(\mathbf{E}, e)$ is the maximal size of the symbolic condition under which e is equal to a specific term. i.e., $\beta(\mathbf{E}, e) = \max_{(\tau,\phi)\in\mathbf{E}(e)} \delta(\phi)$.

According to the rules in Figure 6.5 and Figure 6.6, $\mathbf{E}$ is only updated by the rules ASSIGN, SEQ, and ITE-S.

- Let's consider $\alpha(\mathbf{E}', v)$ and $\beta(\mathbf{E}', v)$ after applying the rule ASSIGN. If the rule ASSIGN applies the rule VAR, we have

  $$\alpha(\mathbf{E}', v) = 1, \;\; \beta(\mathbf{E}', v) = 1$$

  If the rule ASSIGN applies the rule AE, we have

  $$\alpha(\mathbf{E}', v) = 1, \;\; \beta(\mathbf{E}', v) = \beta(\mathbf{E}, a_1) + \beta(\mathbf{E}, a_2)$$

- After applying the rule ITE-S, for any $e \in \mathrm{dom}(\mathbf{E}')$, we have

$$\alpha(\mathbf{E}', e) = O(\alpha(\mathbf{E}_1, e) + \alpha(\mathbf{E}_2, e))$$

$$\beta(\mathbf{E}', e) = \max_{i \in \{1,2\}} \beta(\mathbf{E}_i, e) = O(\beta(\mathbf{E}_1, e) + \beta(\mathbf{E}_2, e))$$

- After applying the rule SEQ, the effects of the involved rules accumulate.

Based on the above equations, we can find that $\alpha(\mathbf{E}, e)$ and $\beta(\mathbf{E}, e)$ are both linear to the times of applying the rules. Thus, for any program location, we have

$$\max_{e \text{ in } r} \alpha(\mathbf{E}, e) = O(N), \quad \max_{e \text{ in } r} \beta(\mathbf{E}, e) = O(N)$$

Now, we can estimate the upper bound of $\delta(\varphi)$. According to the rules in Figure 6.5 and Figure 6.6, Φ is only updated by the rules ASSERT, ITE-S, and SEQ.

- Let's consider $\delta(\Phi')$ after applying the rule ASSERT. If the rule ASSERT applies the rule ACmp, we have

$$\delta(\Phi') - \delta(\Phi) = \sum_{(t_i, \phi_i) \in \mathbf{E}_i(u_i)} (1 + \delta(\phi_1) + \delta(\phi_2)) = O(N^2)$$

  If the rule ASSERT applies the rule ITE-E, we have

$$\delta(\Phi') - \delta(\Phi) = \delta(\gamma_1) + \delta(\gamma_2) + 3$$

  Observe that γ_1 and γ_2 are obtained by applying the rules ACmp and ITE-E. We can sum up the above two equations and obtain that

$$\delta(\Phi') - \delta(\Phi) = O(N^3)$$

- After applying the rule ITE-S, we have the following relation:

$$\delta(\Phi') = 3 + \delta(\Phi_1) + \delta(\Phi_2)$$

- The rule SEQ accumulates the effects upon $\delta(\Phi)$.

By summating all the above equations, we have

$$\delta(\Phi_e) = O(N^4)$$

Notice that the negation elimination can reduce the size of the FOL formula. Therefore, we have

$$\delta(\varphi) \leqslant \delta(\Phi_e) = O(N^4)$$

Notably, the estimated upper bounds of $\max_{e \text{ in } r} \alpha(\mathbf{E}, e)$, $\max_{e \text{ in } r} \beta(\mathbf{E}, e)$, and $\delta(\Phi_e)$ are not tight. For clarify, we only attempt to bound them with the polynomial function of N. The upper bounds can be further strengthened by the polynomial functions of the numbers of specific AST nodes.

Lemma 6.6.2 *Given a data constraint* r*, its symbolic representation* φ *can be constructed in polynomial time to* N*, where* N *is the node number of the abstract syntax tree of* r*.*

Proof of Lemma 6.6.2. We examine the time complexity of the rules in Figure 6.5 and Figure 6.6. The rules `ACmp` and `AE` can be applied in $O(N^2)$ time, as they have to iterate two sets pairwise. The other rules are all applied in $O(1)$ time, as they only need to construct a constant number of FOL formulas. Each of the above rules is applied at most $O(N)$ times. Therefore, the symbolic representation can finally be constructed in $O(N^3)$.

Proof of Theorem 6.6.1. Now, we present the proof of Theorem 6.6.1 as follows.

- First, we can obtain that φ_1 and φ_2 can be constructed in polynomial time to N based on Lemma 6.6.2.
- Second, the divergence analysis actually traverses the parse trees of φ_1 and φ_2, of which the sizes are both polynomial to N, as Lemma 6.6.1 indicates that $\delta(\varphi_1)$ and $\delta(\varphi_2)$ are polynomial to N. Thus, the divergence analysis also works in polynomial time.
- Third, the function *SCTree* constructs the parse trees in $O(\delta(\varphi_1) + \delta(\varphi_2))$ time, which is polynomial to N. The AHU algorithm also works in $O(M)$ time, where M is the node number of the tree. According to Lemma 6.6.1, M is polynomial to N. Thus, the isomorphism analysis works in polynomial time.

Therefore, the steps in Algorithm 7 before line 6 run in polynomial time to N. Q.E.D.

At the end of the section, we want to emphasize the following two points. First, we omit the discussion of several rules that are not shown in Figure 6.5 and Figure 6.6 when proving the two lemmas, e.g., the rules of evaluating the boolean expressions with logical connectives. However, the arguments are similar to the rules that are discussed in the proofs. Actually, the two lemmas hold for any data constraints in the syntax shown in Figure 6.4. Second, we do not provide the tight estimation of the complexity. As shown in the proof of Lemma 6.6.1, the upper bound of $\delta(\Phi_e)$ would be quite sophisticated, involving with the numbers of program constructs in different kinds, if we want to give a tight bound. In this work, we only tend to show that the steps before the SMT solving can be achieved in polynomial time, while the SMT solving may consume exponential time cost theoretically.

Theorem 6.6.2 (Soundness and Completeness) *For the syntax in Figure 6.4, the data constraints are semantically equivalent if Algorithm 7 returns true and vice versa.*

Proof. If two data constraints r_1 and r_2 are semantically equivalent, the divergence analysis does not return *true*, as it can not find an interpretation making their symbolic representations φ_1 and φ_2 evaluate to different truth values. Meanwhile, φ_1 and φ_2 are essentially the FOL formulas in the fragment of bit-vector theory, floating-point arithmetic theory, and word equations, which is theoretically decidable. Therefore, the SMT solving must terminate and return *UNSAT*, making Algorithm 7 returns *true*.

If Algorithm 7 returns *true*, the isomorphism analysis returns *true* or the SMT solving returns *UNSAT*. In the first case, the parse trees of two symbolic representations φ_1 and φ_2 are isomorphic, indicating that they must evaluate to the same truth values under a given interpretation, so the data constraints are semantically equivalent. In the second case, we have φ_1 and φ_2 are logically equivalent, implying the data constraint equivalence. Therefore, EQDAC is sound and complete. Q.E.D.

6.7 Implementation

We have implemented EQDAC in Python and deployed it in Ant Group. EQDAC first generates the AST of a data constraint and then translates it to the symbolic representation. We leverage the Z3 SMT solver [161] to support the SMT solving in the third stage. Particularly, we utilize the bit-vector, floating-point arithmetic, and string theory to encode variables and literals in the finite-length integer, floating point, and string types, respectively.

Based on EQDAC, we have further implemented two bots, which are shown in Figure 6.2, to conduct the equivalence clustering and searching, respectively. In the equivalence clustering, we verify the equivalence of data constraints by invoking EQDAC pairwise. Particularly, we cache the symbolic representation of each data constraint to avoid redundant construction in different invocations. Similarly, we examine the equivalence of a new data constraint and each existing one sequentially in the equivalence searching, and also generate the symbolic representation for a data constraint only once.

6.8 Evaluation

To quantify the effectiveness and efficiency, we evaluate EQDAC upon the data constraints in a FinTech system by investigating the following research questions:

- **RQ1:** How many equivalent data constraints are identified?
- **RQ2:** How efficient is EQDAC in the equivalence clustering and searching?
- **RQ3:** How important is each of the three stages?

Subjects. We collect 30,801 data constraints from a FinTech system in Ant Group, which are in the syntax shown in Figure 6.4. Averagely, a data constraint contains 9.4 data constraints and 17.6 lines of code. Despite the moderate average size, we still need to handle the large set of data constraints efficiently, which is non-trivial yet crucial in industrial scenarios. Lastly, there are 1,497 data constraints not obeying our syntax, which are not selected as the subjects. They mainly contain advanced string operations, e.g.,

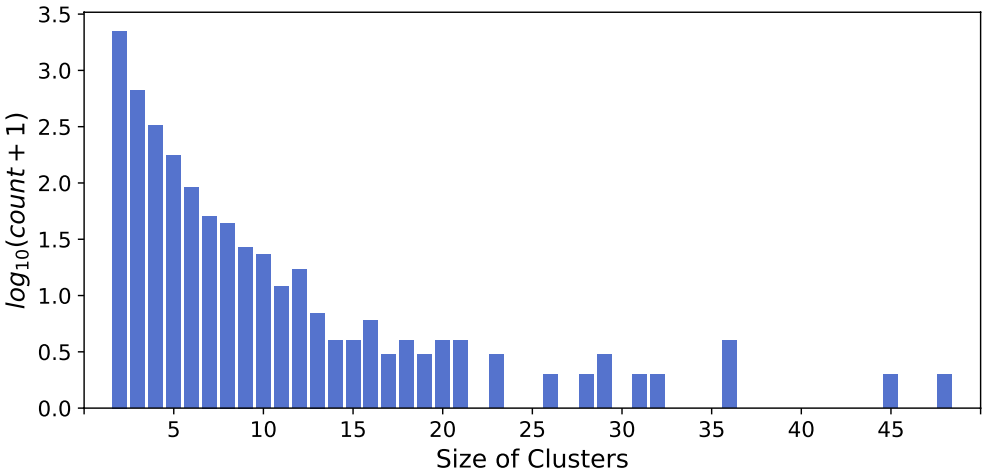


Figure 6.8: The counts and sizes of clusters

substring and *replaceAll*, and system calls, e.g., *getTimeZone*. In this work, EQDAC focuses on the data constraints in our given syntax, covering most of the data constraints (95.4%) in the FinTech system of Ant Group.

Environment. We conduct all the experiments on a 64-bit machine with 40 Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20 GHz and 512 GB of physical memory. We invoke the Z3 SMT solver with its default options. We run the experiments with a time limit of 12 hours and a memory limit of 16 GB.

Availability. We release the code and sample constraints in GitHub repository [194]. The whole set of data constraints cannot be shared because of confidentiality agreements.

6.8.1 Equivalent Data Constraint Identification

To answer the first question, we evaluate EQDAC upon 30,801 data constraints by verifying the data constraint equivalence pairwise. Specifically, each pair of data constraints is fed to EQDAC to determine whether they are equivalent.

Result. We find that 11,538 data constraints (37.5%) have one or more equivalent variants, forming 26,789 equivalent pairs and 3,696 equivalence clusters. Particularly, we can leave one data constraint in each equivalence cluster and eliminate 7,842 data constraints without compromising the validity of the data validation. Due to our limited permission, we sample a subset of the data constraints and measure the CPU time reduction when

avoiding checking redundant ones. The result shows that the CPU time reduction ratio reaches 15.48%. According to the feedback of the experts, any reduction can bring a drastic benefit to the overall system in the long run, as data constraints are frequently checked online during a long development cycle.

We also count the data constraints in each cluster, in which data constraints are equivalent to each other. Figure 6.8 shows that the size of a cluster ranges from 2 to 48. Specifically, the number of clusters with a size of 2 is 2,233. For the largest cluster with the size of 48, a violation of any data constraint will generate 48 alerts. Therefore, identifying equivalent data constraints can provide practical guidance in reusing the checking results and support the redundant alert elimination.

Answer to RQ1: EQDAC identifies 26,789 equivalent pairs from 30,801 data constraints, which indicates that 7,842 data constraints can be safely removed.

6.8.2 Performance Evaluation

We investigate the time consumption and memory usage of EQDAC in the equivalence clustering and searching. The experimental configurations are set up as follows.

- *Equivalence clustering*: To quantify the cost of clustering different sizes of data constraint sets, we construct eight sets of the data constraints, of which the sizes range from 100 to 30,801, and measure the time and memory usage of the clustering. All the data constraints are selected randomly.
- *Equivalence searching*: We select 1,000 data constraints from 30,801 data constraints as the recently-submitted ones and regard the remaining as the existing ones. Specifically, half of the selected ones are equivalent to at least one data constraint in the remaining set to quantify the cost of the equivalence searching in the worst case.

Result. As shown by Figure 6.9, EQDAC finishes analyzing 30,801 data constraints in 2.89 hours within 5.01 GB of peak memory. We perform the regression analysis to quantify the scalability, choosing the quadratic and linear functions as the templates of the

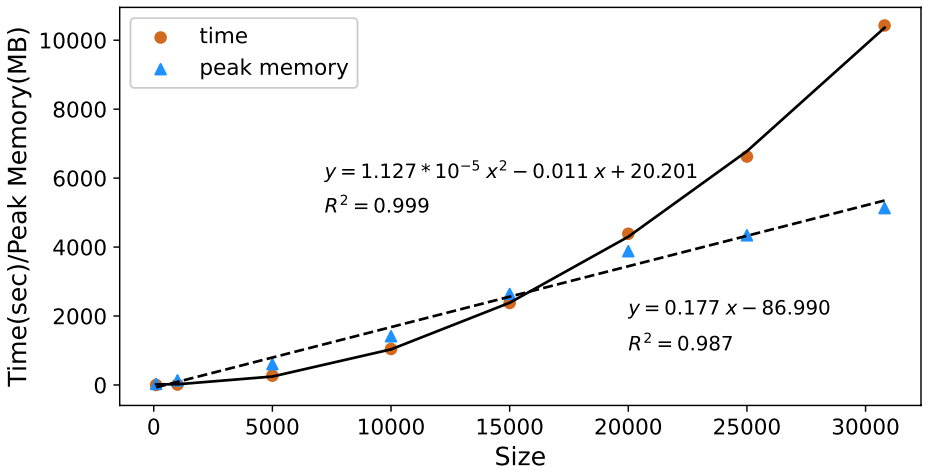


Figure 6.9: Time and memory cost of equivalence clustering

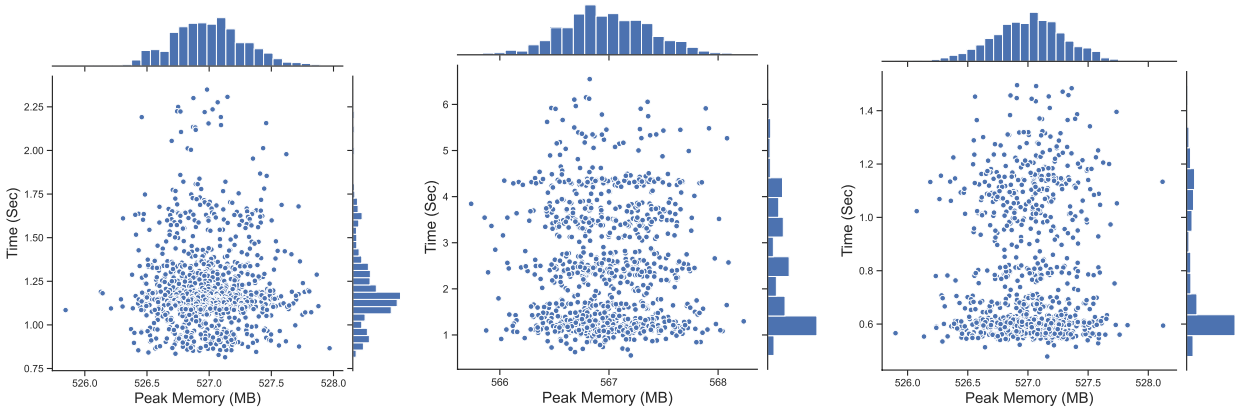
regression analyses for the time and memory cost, respectively, as we construct a symbolic representation for each data constraint only once and invoke the decision procedure in a pairwise manner. The R-squared values for memory and time are 0.987 and 0.999, respectively. Also, the coefficients in the quadratic and linear terms are quite small, indicating that the overhead increases gently. In summary, EQDAC supports the scalable equivalence clustering.

Figure 6.10a shows the cost of the equivalence searching. All the analyses finish in 2.5 seconds within 528 MB of peak memory. Specifically, there is little difference in memory cost, ranging from 525.85 MB to 527.87 MB, while the time cost has a relatively large variance. The analyses of several data constraints demand SMT solving, which introduces more time costs. Typically, most of the cases can be analyzed in 1.5 seconds, and the average time cost is only 1.22 seconds. Thus, EQDAC supports searching equivalent data constraints efficiently, which is essential for maintenance.

Answer to RQ2: EQDAC supports the equivalence clustering of 30,801 data constraints in 2.89 hours within 6 GB peak memory, and the equivalence searching in 1.22 seconds within 527.1 MB peak memory on average, promoting it to be adopted in the practical use.

Table 6.1: The statistics of the equivalence clustering

Variant	Time(h)	Mem(GB)	#Eq Pair	#Redundant
EQDAC-ND	OOT	7.27	141	53
EQDAC-NI	4.48	6.80	26,789	7,842
EQDAC-NS	2.13	3.94	25,952	7,296
EQDAC	2.89	5.01	26,789	7,842



(a) (b) (c)

Figure 6.10: Time and memory cost of EQDAC, EQDAC-NI, and EQDAC-NS

6.8.3 Ablation Study

We set three ablations, namely EQDAC-ND, EQDAC-NI, and EQDAC-NS, which skip the divergence analysis, the isomorphism analysis, and SMT solving, respectively. The first two ablations are sound and complete, while EQDAC-NS can return *unknown* due to its incompleteness

Result. Table 6.1 shows the results of the ablation study in the equivalence clustering. As we can see, EQDAC-ND does not finish analyzing 30,801 data constraints in 12 hours, and its peak memory reaches nearly 7.27 GB. Specifically, EQDAC-ND only finishes comparing seven data constraints with the remaining data constraints pairwise, discovering 141 equivalent pairs and 53 redundant data constraints. EQDAC-NI discovers the same equivalent pairs as EQDAC. However, it has to perform the SMT solving for all the data constraint pairs of which the equivalence is not refuted by the divergence analysis, increasing the time cost to 4.48 hours. EQDAC-NS skips the SMT solving and consumes less time and memory than EQDAC, spending 1.78 and 0.35 hours on the divergence analysis and the isomorphism analysis, respectively. It does not discover 837

```
/* Data contraint 1 */
assert(t.id != t.pid);
assert(ut.oid != ut.iid);
if (t.id == ut.oid){
  assert(t.pid == ut.iid);
} else {
  assert(t.id == ut.iid);
  assert(t.pid == ut.oid);
}
```

```
/* Data contraint 2 */
if (t.id == ut.iid){
  assert(ut.oid == t.pid);
} else {
  assert(t.id == ut.oid);
  assert(t.pid == ut.iid);
}
assert(ut.iid != ut.oid);
assert(t.pid != t.id);
```

Figure 6.11: An example of case study

equivalent pairs, and thus, misses 546 redundant constraints. Particularly, our divergence analysis identifies 38,964 non-equivalent pairs even if their symbolic representations have the same sets of data variables, literals, and operators. Thus, the divergence analysis not only refutes the equivalence soundly but also provides the possibility of refuting more non-equivalent pairs.

Figure 6.10 shows the cost of the equivalence searching. EQDAC-NI costs more in each equivalence searching task, as the SMT solver consumes more resources to prove the equivalence. Specifically, its average time cost is 2.53 seconds, and its peak memory reaches 566.98 MB. In the worst case, it takes 6.56 seconds to finish the equivalence searching of a data constraint, degrading its usability in real-world production. EQDAC-NS consumes less time because it does not invoke SMT solvers in all the cases. However, it can not identify the equivalent variants for 37 data constraints due to incompleteness. EQDAC-ND does not finish the equivalence searching of 1,000 data constraints in the given time budget. It has to invoke the SMT solver to prove the non-equivalence, making the overall time cost unacceptable.

Case Study. Figure 6.11 shows an equivalent pair discovered via the SMT solving. The data constraints both examine whether the IDs of the income and expense accounts match with the ones in the transaction. Unfortunately, we can not deduce the equivalence from the parse trees of their symbolic representations. Instead, we have to reason multiple assertions in a relational manner. The two assertions in the sequencing are the premise of the equivalence of two $\textbf{ite}_s$ statements. Determining their equivalence is beyond the ability of the isomorphism analysis.

Answer to RQ3: All the three stages in EQDAC are necessary for making the decision procedure practical in the real-world scenario.

6.8.4 Discussion

In what follows, we demonstrate the discussions on the feedback from the users, threats to validity, limitations, and future work.

Feedback from the Users. EQDAC has been integrated into the production line of Ant Group, serving as the core building block of two bots in the CI/CD workflow. To obtain the feedback of users, we assigned the questionnaires to the developers and the quality assurance managers in the forum of the company, which received rave reviews from users. For example, a developer comments the search bot in the forum as follows, showing his appreciation for the instant response and useful results.

> *"The search is so smooth! I had been expecting such an assistant for data constraint maintenance. The results are mostly fetched in just one or two seconds, assisting in merging data constraints."*

Threats to Validity. A threat to validity is whether the way of producing data constraints affects the evaluation result of EQDAC. As introduced in Section 6.2.1, ineffective communication between developers could increase the number of equivalent data constraints, as they are unaware of the data constraints submitted by others. For a small FinTech system with only a few data constraints, the benefit of resolving redundancy could be less significant. EQDAC mainly targets systems with thousands of data constraints and shows excellent potential to optimize data validation process.

Limitations and Future Work. First, our syntax excludes several string operations. Theoretically, solving general string constraints is undecidable [195, 140, 196], while the first two stages of EQDAC can still work in the presence of advanced string operations. It would be interesting to reason more string operations even though, according to our experience, they do not widely exist. Second, EQDAC focuses on equivalence relations in this work. It is meaningful to examine whether a data constraint subsumes others for consolidation [197]. Third, data constraints are widely utilized in general data-centric systems that take databases as their backend storages. They can also be instantiated in SQL and other domain-specific languages. Hence, it would be promising to extend EQDAC to support lightweight equivalence checking for the data constraints in other languages,

such as SQL queries [75, 198] and data validation scripts.

6.9 Conclusion

We have presented EQDAC, an efficient, sound, and complete decision procedure for verifying the data constraint equivalence in FinTech systems. It supports two typical clients, namely equivalence clustering and searching, in the production line of a global FinTech company. EQDAC scales to a large number of data constraints with high efficiency and promotes the optimization of data validation in the systems. We believe that the insight behind EQDAC can further promote equivalence checking in other domains.

CHAPTER 7

CONCLUSION AND FUTURE WORKS

7.1 Conclusion

Data-centric systems are crucial for real-world production. As the backbone of industrial infrastructure, their reliability and performance have a significant impact on the overall productivity and efficiency of the industrial sector. In this thesis, we thoroughly investigate data-centric systems from three typical perspectives: the application side, the library side, and the database side. Our works offer a systematic solution that enables us to understand how developers organize, propagate, manipulate, and validate data in these systems, ultimately improving their reliability and performance systematically. Our research fills the gap in existing static analysis techniques and will significantly impact future research on data-centric systems.

Specifically, our first work proposes CRES, a synthesis framework for optimizing data organization with efficient container types in the application code. CRES leverages insights from container semantic abstraction and complexity abstraction to discover more efficient container types for the replacement. We state and prove the soundness and complexity of our synthesis algorithm. Our empirical evaluation demonstrates CRES can successfully reduce the execution time of application code by 8.1% on average, showing its value in optimizing the performance of data-centric systems from the application side.

Our second work presents a value-flow analysis framework, ANCHOR, which targets the container semantic reasoning to understand the data propagation in the application code. Conducting strong updates on the memory layouts of anchored containers, ANCHOR improves the precision of identifying value-flows through containers, which benefits various downstream clients, including thin slicing and value-flow bug detection. Our evaluation demonstrates that ANCHOR successfully detects 20 null pointer exceptions with a 9.1% false-positive ratio, showing its great potential to improve system reliability.

Our third work, DAINFER, concentrates on the data manipulation conduced by library APIs and infers the API aliasing specifications from library documentation. Utilizing a tagging model and a large language model, DAINFER effectively interprets the informal semantic information in the documentation and efficiently infers the API aliasing specifications with neurosymbolic optimization. The inferred specifications can promote understanding indirect value flows in the data-centric applications even when the library implementations are unavailable. Our evaluation offers strong empirical evidence that DAINFER can achieve the API aliasing specification inference with high precision and recall, further offering valuable program facts for bug detection and program optimization.

Finally, our fourth work presents a decision procedure, namely EQDAC, for identifying equivalent data constraints to avoid unnecessary data validation in the database. EQDAC supports equivalence searching and equivalence clustering efficiently, enabling developers to resolve equivalent data constraints and improve system performance. It utilizes the lexical difference and isomorphic structures in the data constraints and invokes an SMT solver to ensure the completeness of the decision procedure for a decidable fragment. Our experiments over a large dataset from a real-world FinTech system demonstrate the high efficiency of EQDAC in supporting equivalence clustering and searching. The elimination of redundant data constraints significantly improve the efficiency of data validation process, reducing the CPU time by 15.48%, which promote the system performance from the database side.

7.2 Future Works

ORM Usage Optimization. As a typical data-centric system, database-backed applications manipulate the database tables to achieve specific application logic. Developers often use object-relational mapping (ORM) libraries to establish mappings between records in tables and objects. The APIs provided by ORM libraries enable developers to construct SQL queries for database table manipulation, such as CRUD operations, which significantly simplifies the development of database-backed applications, decoupling SQL statement construction from application logic implementation. Unfortunately, it is widely observed that developers can introduce inefficient ORM usage in application code. This is

because developers may choose different ORM APIs and compose them together to implement the same functionality, while those implementations significantly differ in efficiency. To address the efficiency issue, existing techniques mainly focus on specific patterns of inefficient ORM usage. For example, Yang et al. [32] collect a set of anti-patterns from Stack Overflow and design a series of detectors for each inefficient pattern. Alexi et al. [199] concentrate on the *N+1* problem and avoid unnecessary database queries. However, previous efforts do not propose a general and fully automatic solution to discover anti-patterns for general ORM libraries. In the future, it would be meaningful and promising to explore this direction and design a general mechanism to optimize the usage of any ORM library.

Blockchain Application Analysis. This thesis focuses on Web2 applications that store application-specific data in the database on a single server. However, recent years have witnessed the increasing popularity of Web3 applications, which utilize blockchain technology for decentralization and privacy. They are built on smart contracts as backends, which enable data storage in a distributed manner, enhancing security and tamper resistance. Such new computation architecture has led to new types of applications, e.g., decentralized finance and non-fungible tokens. Various kinds of bugs, such as reentrancy, integer overflow/underflow, and logic errors, can have significant security impacts [200], potentially leading to the loss of assets stored in the contract. In addition, inefficient smart contract implementation can result in high gas costs [6] and even lead to the gas-out-of-bound vulnerability [201]. In the future, it is promising to ensure the reliability and performance of blockchain applications with domain-specific static analysis techniques.

Networking Systems Analysis. As critical systems transmitting data, networking systems have gained many attentions from the system research community, especially when the network virtualization techniques like NFV [202, 203] have brought interesting ideas to distributed systems. Nowadays, networks can be programmed just like a software system [204, 205, 206]. Many research efforts have lied on the performance aspects of systems [207, 208, 209, 210, 211]. Applying software engineering techniques, such as automated testing and program analysis, can improve not only the efficiency of a distributed system's implementation [212], but also the correctness of such software artifacts. Thus, we also plan to explore new opportunities in this field.

PUBLICATIONS

Thesis related publications

- **Chengpeng Wang**, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang, Complexity-Guided Container Replacement Synthesis, In **OOPSLA 2022** : The ACM SIGPLAN Conference on Objected Oriented Programming, Systems, Languages and Applications, Dec, 2022. **(SIGPLAN Distinguished Paper Award)**

- **Chengpeng Wang**, Wenyang Wang, Peisen Yao, Qingkai Shi, Jinguo Zhou, Xiao Xiao, and Charles Zhang, Anchor: Fast and Precise Value-Flow Analysis for Containers via Memory Orientation, In **TOSEM**: The ACM Transactions on Software Engineering and Methodology, Sept, 2022.

- **Chengpeng Wang**, Gang Fan, Peisen Yao, Fuxiong Pan, and Charles Zhang, Verifying Data Constraint Equivalence in FinTech Systems, In **ICSE 2023**: The IEEE/ACM International Conference on Software Engineering, May, 2023.

- **Chengpeng Wang**, Jipeng Zhang, Rongxin Wu, and Charles Zhang, DAInfer: Inferring API Aliasing Specifications from Library Documentation via Neurosymbolic Optimization, In **FSE 2024**: The ACM International Conference on the Foundations of Software Engineering, July, 2024.

Other publications

- Hao Ling, Heqing Huang, **Chengpeng Wang**, Yuandao Cai, and Charles Zhang, GiantSan: Efficient Memory Sanitization with Segment Folding, In **ASPLOS 2024**: the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, April, 2024

- Wensheng Tang, Dejun Dong, Shijie Li, **Chengpeng Wang***, Peisen Yao, Jinguo Zhou, and Charles Zhang, Octopus: Scaling Value-Flow Analysis via Parallel Collection of Realizable Path Conditions, In **TOSEM**: The ACM Transactions on Software Engineering and Methodology, Oct, 2023.

- Rongxin Wu, Yuxuan He, Jiafeng Huang, **Chengpeng Wang***, Wensheng Tang, Qingkai Shi, Xiao Xiao, and Charles Zhang, LibAlchemy: A Two-Layer Persistent Summary Design for Taming Third-Party Libraries in Static Bug-Finding Systems, In **ICSE 2024**: The IEEE/ACM International Conference on Software Engineering, April, 2024.

- Wensheng Tang†, **Chengpeng Wang**†, Peisen Yao, Rongxin Wu, Xianjin Fu, Gang Fan, and Charles Zhang, DCLink: Bridging Data Constraint Changes and Implementations in FinTech Systems, In **ASE 2023**: IEEE/ACM International Conference on Automated Software Engineering, Sept, 2023.

- **Chengpeng Wang**, Peisen Yao, Wensheng Tang, Gang Fan, and Charles Zhang, Synthesizing Conjunctive Queries for Code Search, In **ECOOP 2023**: European Conference on Object-Oriented Programming, July, 2023.

- Zongyin Hao, Quanfeng Huang, **Chengpeng Wang**, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang, Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis, In **ATC 2023**: USENIX Annual Technical Conference, July, 2023.

- **Chengpeng Wang**, CodeSpider: Automatic Code Querying with Multi-modal Conjunctive Query Synthesis, In **SPLASH SRC 2022**: The ACM SIGPLAN conference on Systems, Programming, Languages, and Applications: Software for Humanity, Student Research Competition, Dec, 2022.

- Rongxin Wu, Minglei Chen, **Chengpeng Wang***, Gang Fan, Jiguang Qiu, and Charles Zhang, Accelerating Build Dependency Error Detection via Virtual Build, In **ASE 2022**: The IEEE/ACM International Conference on Automated Software Engineering, Oct, 2022.

- Gang Fan, **Chengpeng Wang**, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang, Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph, In **ISSTA 2020**: The ACM SIGSOFT International Symposium on Software Testing and Analysis, July, 2020.

† means equal contribution. * means corresponding author.

REFERENCES

[1] Donglin Liang and Mary Jean Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In Patrick Cousot, editor, *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*, volume 2126 of *Lecture Notes in Computer Science*, pages 279–298. Springer, 2001.

[2] Yulei Sui, Sen Ye, Jingling Xue, and Jie Zhang. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Softw. Pract. Exp.*, 44(12):1485–1510, 2014.

[3] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 693–706. ACM, 2018.

[4] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 72–82. IEEE / ACM, 2019.

[5] Thomas W. Reps. Program analysis via graph reachability. In Jan Maluszynski, editor, *Logic Programming, Proceedings of the 1997 International Symposium, Port Jefferson, Long Island, NY, USA, October 13-16, 1997*, pages 5–19. MIT Press, 1997.

[6] Yanju Chen, Yuepeng Wang, Maruth Goyal, James Dong, Yu Feng, and Isil Dillig. Synthesis-powered optimization of smart contracts via data type refactoring. *Proc. ACM Program. Lang.*, 6(OOPSLA2):560–588, 2022.

[7] Shankara Pailoor, Yuepeng Wang, Xinyu Wang, and Isil Dillig. Synthesizing data structure refinements from integrity constraints. In Stephen N. Freund and Eran

Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, pages 574–587. ACM, 2021.

[8] Malavika Samak, Deokhwan Kim, and Martin C. Rinard. Synthesizing replacement classes. *Proc. ACM Program. Lang.*, 4(POPL):52:1–52:33, 2020.

[9] Microsoft. Containers-C++ Reference. `https://www.cplusplus.com/reference/stl/`, 2022. [Online; accessed 7-Sept-2022].

[10] Oracle. Collections Framework Overview. `https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html`, 2023. [Online; accessed 1-March-2023].

[11] Apache. Apache Commons Collections. `https://commons.apache.org/proper/commons-collections/`, 2023. [Online; accessed 1-March-2023].

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

[13] Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. Scaling static taint analysis to industrial SOA applications: a case study at alibaba. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *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*, pages 1477–1486. ACM, 2020.

[14] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sicheng Pan, Ge Li, Siddharth Jha, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. Leveraging application data constraints to optimize database-backed web applications. *Proc. VLDB Endow.*, 16(6):1208–1221, 2023.

[15] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. Managing data constraints in database-backed web applications. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1098–1109. ACM, 2020.

[16] Tianxiao Wang, Chen Zhi, Xiaoqun Zhou, Jinjie Wu, Jianwei Yin, and Shuiguang Deng. Data constraint mining for automatic reconciliation scripts generation. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1119–1130. ACM, 2023.

[17] Eclipse. Eclipse Collections Main Library. `https://mvnrepository.com/artifact/org.eclipse.collections/eclipse-collections/`, 2023. [Online; accessed 1-March-2023].

[18] Fastutil. Fastutil. `https://mvnrepository.com/artifact/it.unimi.dsi/fastutil`, 2023. [Online; accessed 1-March-2023].

[19] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T Barr. Darwinian data structure selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 118–128, 2018.

[20] Isil Dillig, Thomas Dillig, and Alex Aiken. Symbolic heap abstraction with demand-driven axiomatization of memory invariants. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 397–410. ACM, 2010.

[21] Ayse Isil Dillig. *Precise and Automatic Verification of Container-Manipulating Programs*. Citeseer, 2011.

[22] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Active learning of points-to specifications. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 678–692. ACM, 2018.

[23] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin T. Vechev. Unsupervised learning of API alias specifications. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 745–759. ACM, 2019.

[24] Android. Android Platform Documentation. `https://developer.android.com/docs/`, 2023. [Online; accessed 1-March-2023].

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

[26] Khushboo Chitre, Piyus Kedia, and Rahul Purandare. The road not taken: exploring alias analysis based optimizations missed by the compiler. *Proc. ACM Program. Lang.*, 6(OOPSLA2):786–810, 2022.

[27] Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. Complexity-guided container replacement synthesis. *Proc. ACM Program. Lang.*, 6(OOPSLA):1–31, 2022.

[28] Chengpeng Wang, Wenyang Wang, Peisen Yao, Qingkai Shi, Jinguo Zhou, Xiao Xiao, and Charles Zhang. Anchor: Fast and precise value-flow analysis for containers via memory orientation. *ACM Transactions on Software Engineering and Methodology*, 2022.

[29] Chengpeng Wang, Gang Fan, Peisen Yao, Fuxiong Pan, and Charles Zhang. Verifying data constraint equivalence in fintech systems. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1329–1341. IEEE, 2023.

[30] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. Conquering the extensional scalability problem for value-flow analysis frameworks. In Gregg Rothermel

and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 812–823. ACM, 2020.

[31] JavaEE. Java(TM) EE 8 Specification APIs. `https://javaee.github.io/javaee-spec/javadocs/`, 2023. [Online; accessed 1-March-2023].

[32] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. How *not* to structure your database-backed web applications: a study of performance bugs in the wild. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 800–810. ACM, 2018.

[33] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 160–173. ACM, 2010.

[34] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 174–186. ACM, 2010.

[35] Juan Manuel Florez, Jonathan Perry, Shiyi Wei, and Andrian Marcus. Retrieving data constraint implementations using fine-grained code patterns. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1893–1905. ACM, 2022.

[36] Juan Manuel Florez, Laura Moreno, Zenong Zhang, Shiyi Wei, and Andrian Marcus. An empirical study of data constraint implementations in java. *Empir. Softw. Eng.*, 27(5):119, 2022.

[37] Haochen Huang, Bingyu Shen, Li Zhong, and Yuanyuan Zhou. Protecting data integrity of web applications with database constraints inferred from application code. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors,

Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023, pages 632–645. ACM, 2023.

[38] Google. A high performance expression evaluator for java. `https://code.google.com/archive/p/aviator/`, 2022. [Online; accessed 7-Sept-2022].

[39] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, pages 503–514, 2014.

[40] Changhee Jung, Silvius Rus, Brian P Railing, Nathan Clark, and Santosh Pande. Brainy: Effective selection of data structures. *ACM SIGPLAN Notices*, 46(6):86–97, 2011.

[41] Ohad Shacham, Martin T. Vechev, and Eran Yahav. Chameleon: adaptive selection of collections. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 408–418. ACM, 2009.

[42] Wellington Oliveira, Renato Oliveira, Fernando Castor, Benito Fernandes, and Gustavo Pinto. Recommending energy-efficient java collections. In Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 160–170. IEEE / ACM, 2019.

[43] Wellington Oliveira, Renato Oliveira, Fernando Castor, Gustavo Pinto, and João Paulo Fernandes. Improving energy-efficiency by recommending java collections. *Empir. Softw. Eng.*, 26(3):55, 2021.

[44] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 355–368. ACM, 2016.

[45] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Data representation synthesis. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 38–49. ACM, 2011.

[46] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 417–428. ACM, 2012.

[47] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 535–550. ACM, 2018.

[48] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1&2):131–170, 1996.

[49] Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and soot. In Eric Bodden, Laurie J. Hendren, Patrick Lam, and Elena Sherman, editors, *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*, pages 3–8. ACM, 2012.

[50] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 22:1–22:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[51] Johannes Späth, Karim Ali, and Eric Bodden. Ide^{al}: efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):99:1–99:27, 2017.

[52] Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis. *ACM Comput. Surv.*, 49(2):29:1–29:47, 2016.

[53] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 59–76. ACM, 2005.

[54] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 265–266. ACM, 2016.

[55] Pratik Fegade and Christian Wimmer. Scalable pointer analysis of data structures using semantic models. In Louis-Noël Pouchet and Alexandra Jimborean, editors, *CC '20: 29th International Conference on Compiler Construction, San Diego, CA, USA, February 22-23, 2020*, pages 39–50. ACM, 2020.

[56] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[57] Bertrand Jeannet, Alexey Loginov, Thomas W. Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. *ACM Trans. Program. Lang. Syst.*, 32(2):5:1–5:52, 2010.

[58] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 134–150. Springer, 2014.

[59] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Desynchronized multi-state abstractions for open programs in dynamic languages. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software,*

ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, volume 9032 of *Lecture Notes in Computer Science*, pages 483–509. Springer, 2015.

[60] Jiangchao Liu, Liqian Chen, and Xavier Rival. Automatic verification of embedded system code manipulating dynamic structures stored in contiguous regions. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 37(11):2311–2322, 2018.

[61] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979.

[62] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 246–266. Springer, 2010.

[63] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 187–200. ACM, 2011.

[64] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin T. Vechev. Scalable taint specification inference with big code. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 760–774. ACM, 2019.

[65] Timon Gehr, Dimitar K. Dimitrov, and Martin T. Vechev. Learning commutativity specifications. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 2015.

[66] Atanas Rountev, Mariana Sharp, and Guoqing Xu. IDE dataflow analysis in the presence of large object-oriented libraries. In Laurie J. Hendren, editor, *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4959 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2008.

[67] Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the android framework. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 725–735. ACM, 2016.

[68] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011.

[69] Bor-Yuh Evan Chang, Cezara Dragoi, Roman Manevich, Noam Rinetzky, and Xavier Rival. Shape analysis. *Found. Trends Program. Lang.*, 6(1-2):1–158, 2020.

[70] George C. Necula. Translation validation for an optimizing compiler. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 83–94. ACM, 2000.

[71] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 471–482. ACM, 2013.

[72] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 305–316. ACM, 2013.

[73] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured CAD mod-

els with equality saturation and inverse transformations. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 31–44. ACM, 2020.

[74] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. SPES: A two-stage query equivalence verifier. *CoRR*, abs/2004.00481, 2020.

[75] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Dong Xu. Automated verification of query equivalence using satisfiability modulo theories. *Proc. VLDB Endow.*, 12(11):1276–1288, 2019.

[76] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. Client-specific equivalence checking. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 441–451. ACM, 2018.

[77] Anna Trostanetski, Orna Grumberg, and Daniel Kroening. Modular demand-driven analysis of semantic difference for program versions. In *International Static Analysis Symposium*, pages 405–427. Springer, 2017.

[78] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. Constraint-based relational verification. In *International Conference on Computer Aided Verification*, pages 742–766. Springer, 2021.

[79] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 57–69. ACM, 2016.

[80] Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1027–1040. ACM, 2019.

[81] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 362–375, New York, NY, USA, 2017. Association for Computing Machinery.

[82] Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, and Corina S. Pasareanu. Differential symbolic execution. In Mary Jean Harrold and Gail C. Murphy, editors, *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 226–237. ACM, 2008.

[83] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification*, pages 712–717. Springer, 2012.

[84] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 13–24, 2020.

[85] Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. Relational verification using reinforcement learning. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[86] Benjamin Goldberg, Lenore D. Zuck, and Clark W. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electron. Notes Theor. Comput. Sci.*, 132(1):53–71, 2005.

[87] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. Hottsql: proving query rewrites with univalent SQL semantics. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 510–524. ACM, 2017.

[88] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *Proc. VLDB Endow.*, 11(11):1482–1495, 2018.

[89] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–28, 2021.

[90] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[91] Simon Guilloud and Viktor Kuncak. Equivalence checking for orthocomplemented bisemilattices in log-linear time. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*, volume 13244 of *Lecture Notes in Computer Science*, pages 196–214. Springer, 2022.

[92] Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. In Alberto O. Mendelzon and Jan Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 205–213. ACM Press, 1998.

[93] Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. Demonstration of the cosette automated SQL prover. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1591–1594. ACM, 2017.

[94] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Speeding up symbolic reasoning for relational queries. *Proc. ACM Program. Lang.*, 2(OOPSLA):157:1–157:25, 2018.

[95] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. A symbolic approach to proving query equivalence under bag semantics. *arXiv preprint arXiv:2004.00481*, 2020.

[96] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*, pages 207–224. Springer, 2003.

[97] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. SIA: optimizing queries using learned predicates. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2169–2181. ACM, 2021.

[98] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 225–236. ACM, 2016.

[99] Rocco De Nicola. Behavioral equivalences. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 120–127. Springer, 2011.

[100] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 127–139. ACM, 2009.

[101] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 375–385.

ACM, 2009.

[102] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.

[103] Oliver Kennedy and Lukasz Ziarek. Just-in-time data structures. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[104] Guoqing Xu. Coco: Sound and adaptive replacement of java collections. In Giuseppe Castagna, editor, *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2013.

[105] Cres. Report of container replacement synthesis, 2021.

[106] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73. ACM, 2011.

[107] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 136–148. ACM, 2008.

[108] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013.

[109] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. Proteus: computing disjunctive loop summary via path dependency analysis. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 61–72. ACM, 2016.

[110] Akhilesh Srikanth, Burak Sahin, and William R. Harris. Complexity verification using guided theorem enumeration. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 639–652. ACM, 2017.

[111] Tomás Fiedor, Lukás Holík, Adam Rogalewicz, Moritz Sinn, Tomás Vojnar, and Florian Zuleger. From shapes to amortized complexity. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, volume 10747 of *Lecture Notes in Computer Science*, pages 205–225. Springer, 2018.

[112] Tianhan Lu, Bor-Yuh Evan Chang, and Ashutosh Trivedi. Selectively-amortized resource bounding. In Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi, editors, *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings*, volume 12913 of *Lecture Notes in Computer Science*, pages 286–307. Springer, 2021.

[113] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14. ACM, 2013.

[114] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA):63:1–63:26, 2017.

[115] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Path-sensitive sparse

analysis without path conditions. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 930–943. ACM, 2021.

[116] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[117] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 435–446. ACM, 2013.

[118] Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 343–353. ACM, 2011.

[119] Michael P Fay and Michael A Proschan. Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics surveys*, 4:1, 2010.

[120] Andrea Arcuri and Lionel C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 1–10. ACM, 2011.

[121] Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient flow profiling for detecting performance bugs. In Andreas Zeller and Abhik Roychoudhury, editors,

Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016, pages 413–424. ACM, 2016.

[122] Björn Franke, Zhibo Li, John Magnus Morton, and Michel Steuwer. Collection skeletons: Declarative abstractions for data collections. In Bernd Fischer, Lola Burgueño, and Walter Cazzola, editors, *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*, pages 189–201. ACM, 2022.

[123] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. Thin slicing. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 112–122. ACM, 2007.

[124] Susan Horwitz, Thomas W. Reps, and David W. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[125] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: explaining program failures via postmortem static analysis. In Richard N. Taylor and Matthew B. Dwyer, editors, *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*, pages 63–72. ACM, 2004.

[126] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In Patrick D. McDaniel, editor, *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. USENIX Association, 2005.

[127] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 87–97. ACM, 2009.

[128] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: precise refutations for heap reachability. In Hans-Juergen Boehm and Cormac Flanagan, edi-

tors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 275–286. ACM, 2013.

[129] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. Grapple: A graph system for static finite-state property checking of large-scale systems code. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 38:1–38:17. ACM, 2019.

[130] Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. Chianina: an evolving graph system for flow- and context-sensitive analyses of million lines of C code. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, pages 914–929. ACM, 2021.

[131] Mark Marron, Cesar Sanchez, and Zhendong Su. High-level heap abstractions for debugging programs, 2010.

[132] Hiralal Agrawal. *Towards automatic debugging of computer programs*. PhD thesis, Purdue University, 1991.

[133] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. Bottom-up context-sensitive pointer analysis for java. In Xinyu Feng and Sungwoo Park, editors, *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, volume 9458 of *Lecture Notes in Computer Science*, pages 465–484. Springer, 2015.

[134] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.*, 2(OOPSLA):140:1–140:29, 2018.

[135] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. A principled approach to selective context sensitivity for pointer analysis. *ACM Trans. Program. Lang. Syst.*, 42(2):10:1–10:40, 2020.

[136] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[137] Bill McCloskey, Thomas W. Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 71–99. Springer, 2010.

[138] Jiangchao Liu and Xavier Rival. Abstraction of optional numerical values. In Xinyu Feng and Sungwoo Park, editors, *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, volume 9458 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2015.

[139] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2003.

[140] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25:1–25:28, 2012.

[141] Pieter Hooimeijer and Margus Veanes. An evaluation of automata algorithms for string analysis. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2011.

[142] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. S-looper: automatic

summarization for multipath string loops. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 188–198. ACM, 2015.

[143] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[144] David L. Heine and Monica S. Lam. Static detection of leaks in polymorphic containers. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 252–261. ACM, 2006.

[145] Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. Semantic-directed clumping of disjunctive abstract states. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 32–45. ACM, 2017.

[146] Isil Dillig, Thomas Dillig, and Alex Aiken. Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2009.

[147] OWASP. Open Web Application Security Project. `http://www.owasp.org/`, 2023. [Online; accessed 1-March-2023].

[148] Anchor. Bug reports of Anchor. `https://containeranalyzer.github.io/`, 2023. [Online; accessed 1-March-2023].

[149] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[150] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. Computing summaries of string loops in C for better testing and refactoring.

In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 874–888. ACM, 2019.

[151] Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. A combination framework for tracking partition sizes. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 239–251. ACM, 2009.

[152] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. *ACM SIGPLAN Notices*, 41(6):376–386, 2006.

[153] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[154] Peisen Yao, Jinguo Zhou, Xiao Xiao, Qingkai Shi, Rongxin Wu, and Charles Zhang. Efficient path-sensitive data-dependence analysis. *arXiv preprint arXiv:2109.07923*, 2021.

[155] Thomas W. Reps. Program analysis via graph reachability. In Jan Maluszynski, editor, *Logic Programming, Proceedings of the 1997 International Symposium, Port Jefferson, Long Island, NY, USA, October 13-16, 1997*, pages 5–19. MIT Press, 1997.

[156] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of alias. In Lori L. Pollock and Mauro Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 133–144. ACM, 2006.

[157] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming*

Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, pages 480–491. ACM, 2007.

[158] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: semantics-based detection of android malware through static analysis. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 576–587. ACM, 2014.

[159] Ondrej Lhoták and Laurie J. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53, 2008.

[160] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 485–495. ACM, 2014.

[161] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[162] Rong Gu, Zhiqiang Zuo, Xi Jiang, Han Yin, Zhaokang Wang, Linzhang Wang, Xuandong Li, and Yihua Huang. Towards efficient large-scale interprocedural program static analysis on distributed data-parallel computation. *IEEE Trans. Parallel Distributed Syst.*, 32(4):867–883, 2021.

[163] Yichen Xie and Alexander Aiken. Scalable error detection using boolean satisfiability. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 351–363. ACM, 2005.

[164] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. Program tailoring: Slicing by sequential criteria. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 15:1–15:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[165] Facebook. Infer Static Analyzer. `https://fbinfer.com/`, 2022. [Online; accessed 7-Sept-2022].

[166] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70, 2019.

[167] David A. Tomassi and Cindy Rubio-González. On the real-world effectiveness of static bug detectors at finding null pointer exceptions. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 292–303. IEEE, 2021.

[168] David Mitchel Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. Accelerating array constraints in symbolic execution. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 68–78. ACM, 2017.

[169] Trove. High Speed Object and Primitive Collections for Java. `http://trove.starlight-systems.com/`, 2023. [Online; accessed 1-March-2023].

[170] Guava. Google Core Libraries for Java. `https://github.com/google/guava`, 2023. [Online; accessed 1-March-2023].

[171] John Toman and Dan Grossman. Taming the static analysis beast. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*, pages 18:1–18:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[172] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. Do the dependency conflicts in my project

matter? In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *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*, pages 319–330. ACM, 2018.

[173] Simon Butler, Michel Wermelinger, and Yijun Yu. A survey of the forms of java reference names. In Andrea De Lucia, Christian Bird, and Rocco Oliveto, editors, *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, pages 196–206. IEEE Computer Society, 2015.

[174] W Nelson Francis and Henry Kucera. Computational analysis of present-day american english. *Providence, RI: Brown University Press. Kuperman, V., Estes, Z., Brysbaert, M., & Warriner, AB (2014). Emotion and language: Valence and arousal affect word recognition. Journal of Experimental Psychology: General*, 143:1065–1081, 1967.

[175] OpenAI. Introducing chatgpt. 2022.

[176] OpenAI. Gpt-4 technical report, 2023.

[177] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[178] Nikolaj S. Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νz - an optimizing SMT solver. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as*

Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, volume 9035 of *Lecture Notes in Computer Science*, pages 194–199. Springer, 2015.

[179] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[180] Yunlong Lyu, Yi Fang, Yiwei Zhang, Qibin Sun, Siqi Ma, Elisa Bertino, Kangjie Lu, and Juanru Li. Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2096–2113. IEEE, 2022.

[181] NLTK. Natural Language Toolkit. `https://www.nltk.org/index.html`, 2023. [Online; accessed 7-Sep-2023].

[182] OpenAI. GPT-3.5. `https://platform.openai.com/docs/models/gpt-3-5`, 2023. [Online; accessed 7-Sep-2023].

[183] DAInfer. Source code and Data of DAInfer. `https://github.com/DAInfer/DAInferTool`, 2023. [Online; accessed 13-Sept-2023].

[184] ATLAS. Soure code of ATLAS. `https://github.com/obastani/atlas`, 2023. [Online; accessed 13-Sept-2023].

[185] F-Droid. F-Droid. `https://f-droid.org/`, 2023. [Online; accessed 1-Sept-2023].

[186] Feifei Li. Cloud native database systems at alibaba: Opportunities and challenges. *Proc. VLDB Endow.*, 12(12):2263–2272, 2019.

[187] Neil Ernst, Rick Kazman, and Julien Delange. *Technical Debt in Practice: How to Find It and Fix It*. MIT Press, 2021.

[188] Andrés Letelier, Jorge Pérez, Reinhard Pichler, and Sebastian Skritek. Static analysis and optimization of semantic web queries. *ACM Trans. Database Syst.*, 38(4):25:1–25:45, 2013.

[189] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *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*, pages 13–24. ACM, 2020.

[190] Malik Bouchet, Byron Cook, Bryant Cutler, Anna Druzkina, Andrew Gacek, Liana Hadarean, Ranjit Jhala, Brad Marshall, Daniel Peebles, Neha Rungta, Cole Schlesinger, Chriss Stephens, Carsten Varming, and Andy Warfield. Block public access: trust safety verification of access control policies. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *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*, pages 281–291. ACM, 2020.

[191] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[192] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. CI/CD pipelines evolution and restructuring: A qualitative and quantitative study. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*, pages 471–482. IEEE, 2021.

[193] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[194] EqDAC. Equivalence Verification of Data Constraints. `https://github.com/EqDAC/EqDACTool`, 2022. [Online; accessed 7-Sept-2022].

[195] Taolue Chen, Yan Chen, Matthew Hague, Anthony W Lin, and Zhilin Wu. What is decidable about string constraints with the replaceall function. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017.

[196] Taolue Chen, Matthew Hague, Anthony W Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[197] Marcelo Sousa, Isil Dillig, Dimitrios Vytiniotis, Thomas Dillig, and Christos Gkantsidis. Consolidation of queries with user-defined functions. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 554–564. ACM, 2014.

[198] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. Extracting equivalent SQL from imperative code in database applications. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1781–1796. ACM, 2016.

[199] Alexi Turcotte, Mark W. Aldrich, and Frank Tip. reformulator: Automated refactoring of the N+1 problem in database-backed applications. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 84:1–84:12. ACM, 2022.

[200] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying exploitable bugs in smart contracts. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 615–627. IEEE, 2023.

[201] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. etainter: detecting gas-related vulnerabilities in smart contracts. In Sukyoung Ryu and Yannis Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 728–739. ACM, 2022.

[202] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. SNF: Serverless Network Functions. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 296–310, New York, NY, USA, 2020. Association for Computing Machinery.

[203] Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. Quadrant: A cloud-deployable nf virtualization platform. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pages 493–509, New York, NY, USA, 2022. Association for Computing Machinery.

[204] Jane Yen, Jianfeng Wang, Sucha Supittayapornpong, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. Meeting slos in cross-platform nfv. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, pages 509–523, New York, NY, USA, 2020. Association for Computing Machinery.

[205] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100gbps intrusion prevention on a single server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1083–1100. USENIX Association, November 2020.

[206] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*, pages 490–503, 2018.

[207] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gabor Retvari. Batchy: Batch-scheduling data flow graphs with service-level objectives. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 633–649, Santa Clara, CA, February 2020. USENIX Association.

[208] Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos AM Vieira, Ramesh Govindan, and Barath Raghavan. Galleon: Reshaping the square peg of nfv. *arXiv preprint arXiv:2101.06466*, 2021.

[209] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. Packet order matters! improving application performance by deliberately delaying packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 807–827, Renton, WA, April 2022. USENIX Association.

[210] Barath Raghavan, Ramesh Govindan, Zhuojin Li, and Jianfeng Wang. Methods and systems for efficient and secure network function execution, June 2023. US Patent App. 18/082,873.

[211] Jianfeng Wang, Siddhant Gupta, Marcos A. M. Vieira, Barath Raghavan, and Ramesh Govindan. Scheduling network function chains under sub-millisecond latency slos. 2023.

[212] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 49–54, 2012.