Data Lineage Analysis for Enterprise Applications by Manta: The Story of Java and C# Scanners

Pavel Parízek
Charles University
Prague, Czech Republic
parizek@d3s.mff.cuni.cz

Lukáš Hermann
Manta, an IBM company
Czech Republic
lukashermann@ibm.com

ABSTRACT

Data lineage is a view over the whole data environment of a business company or government institution, which represents the flow of data values through the system. It helps people to navigate through all the data storages and data transformations, find the origin of a specific data value, or to ensure data consistency after updates. Manta Flow is an automated data lineage platform that supports many different technologies, including dialects of SQL and programs code written in general-purpose languages.

In this paper, we focus on scanners that analyze programs in Java or C# and generate data flow graphs as output. We describe the process of their development and present the main concepts of the modular symbolic data flow analysis that we designed for this purpose. Then we also discuss technical challenges related to static analysis of real-world enterprise applications that we have faced, explain the key ideas of our current solutions, and share the main lessons learned within this project.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*; **Software post-development issues**.

KEYWORDS

data lineage, symbolic data flow analysis, enterprise applications

ACM Reference Format:
Pavel Parízek and Lukáš Hermann. 2024. Data Lineage Analysis for Enterprise Applications by Manta: The Story of Java and C# Scanners. In *46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3639477.3639739

1 INTRODUCTION

In the modern digital world based on software, practically every business company and government agency uses electronic databases to store their vital data and software to manipulate the data. This is true in particular for large enterprises, which have collected huge amounts of data stored in thousands of databases and tables. Data models and data storage environments used in most companies are therefore getting very complex, and they are also evolving over time when developers (database architects) come and go. Because of that, it may be very difficult and tedious for people to manually navigate through all the databases and tables, find the origin of a specific invalid or suspicious data value, reason about dependencies between data stored in different tables, or to ensure consistency of data when doing some updates. Problems with data management and quality may lead, for example, to wrong business decisions or to leakage of sensitive private data.

These challenges and possible issues can be addressed with the help of *data lineage* information [4, 20], a view over the whole data environment that shows the origin of each data value and how data flow through the system. While such a view could be constructed manually, it would take a very long time and great effort, so automated data lineage analysis of the whole environment (software with data together) and visualization of data flow are needed. Benefits include much better understanding of the organization's data environment and trust in data values [19, 21].

Manta Flow is an automated data lineage platform developed by the software company Manta [1]. The platform consists mainly of the following parts: (1) *scanners* that analyze input programs, SQL, ETL (extract-transform-load) scripts, data analytics scripts, and definitions of reports, (2) the metadata repository used to store the resulting data flow graphs (enabling also integration with other systems), and (3) the web application that presents nice visualizations of data flow graphs, enabling users to inspect them. Currently, Manta Flow supports over 40 scanners for many different technologies, including various dialects of SQL, many popular ETL and reporting tools, and program code written in general-purpose languages such as Java, C#, and Python. In this paper, however, we focus on the scanners for Java and C# programs.

Both scanners have three main components: extractor, the actual data-flow analysis to be run on input programs, and the scanner-specific generator of a partial flow graph.

The extractor is responsible for preparing the input for data flow analysis by retrieving the relevant entities (e.g., Java classes and libraries) from various locations and sources provided by the customer. During this process, the extractor also collects important metadata about the input program and its configuration, and identifies all entry points. Multiple distribution formats of Java and C# programs are supported, in particular Java classes (bytecode), a set of JAR files (Java libraries), Spring Boot executable JAR/WAR files, .NET assemblies, and C# source code files.

The partial flow graphs produced by individual scanners for all the relevant technologies are then saved into the metadata repository, where they are merged into the final output Manta graph. Note that the ability to produce a single output flow graph covering

[1]https://manta.io/

multiple technologies is really essential, because most enterprise applications for data management and processing combine at least SQL with some ETL jobs or programs written in general-purpose languages. This was also our original motivation behind developing scanners for Java and C#, because many enterprise business applications include some parts implemented in Java or C#, typically complex business logic.

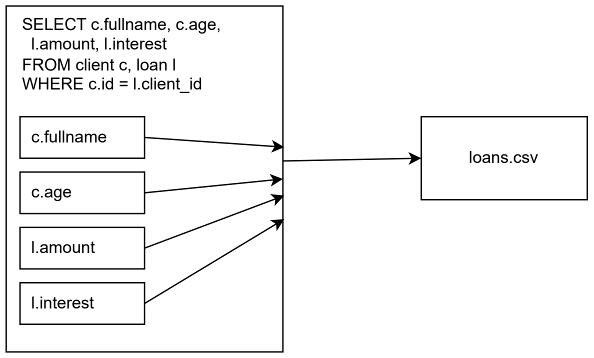
We illustrate the data lineage information computed by Manta Flow on the example of a simplified Java program in Figure 1. It reads data from a database using an SQL SELECT query and writes into a CSV file. The Java scanner would analyze the program and create an intermediate graph that captures possible flow of data from the SELECT query, defined at lines 5–8 and executed at line 9, to a CSV file named loans.csv. This file is opened at line 1 and new content (a line of text) is appended at lines 15-16 within the loop over the result (a list of rows) of the SELECT query.

```
1   FileWriter fw = new FileWriter("loans.csv");
2   BufferedWriter bufw = new BufferedWriter(fw);
3   Connection con = getConnection("192.168.0.1");
4   Statement stmt = con.createStatement();
5   String query = "SELECT c.fullname, c.age, " +
6                  "l.amount, l.interest " +
7                  "FROM client c, loan l " +
8                  "WHERE c.id = l.client_id";
9   ResultSet rs = stmt.executeQuery(query);
10  while (rs.next()) {
11    String loanStr = rs.getString(1) + "," +
12                     rs.getInt(2) + "," +
13                     rs.getInt("amount") + "," +
14                     rs.getDouble("interest");
15    bufw.append(loanStr);
16    bufw.newLine();
17  }
18  bufw.close();
19  con.close();
```

Figure 1: Java program that reads data from an SQL database and writes into a CSV file

As the next step, a scanner for the particular dialect of SQL will parse the SELECT query and definition of the source database, recognize individual columns in respective database tables, and extend the flow graph with edges from nodes that represent columns to the node representing the destination CSV file loans.csv. Both the source SELECT query and the destination CSV file represent data lineage endpoints. Figure 2 shows the essential data lineage information captured by the flow graph. Note that the Manta Flow platform actually creates graphs that are visually much more appealing than our schematic diagram and contain a lot of additional metadata, such as information about the connection to database.

Contribution. The whole data lineage analysis specifically for Java and C# programs has been designed and implemented within the scope of a project between software engineers working at the Manta company, faculty members (researchers) at Charles University and many students. In this paper, we present (1) the story



```
SELECT c.fullname, c.age,
  l.amount, l.interest
FROM client c, loan l
WHERE c.id = l.client_id
```

Figure 2: Data flow graph for the example Java program

behind development of Java and C# scanners, (2) the main technical challenges that we have faced and how we addressed them, and (3) lessons that we have learned (often the hard way) and that may be valuable for the readers. We describe especially the challenges, solutions, our experience, and lessons learned that are related to the development of static data lineage analysis for large enterprise applications. We focus just on technical and software engineering aspects of the whole project, neglecting issues related to operations (DevOps) and relations with customers, but highlighting software engineering challenges associated with the development of tools and products based on static program code analysis. In this focus on the challenges and experience related to static program analysis of large enterprise applications, our work complements recently published studies and techniques by Harman et al. [9], Antoniadis et al. [1], Wang et al. [18], and Chen et al. [3].

2 DESIRED FEATURES AND INITIAL STEPS

In this section we discuss the originally desired features of the data lineage analysis, main characteristics of applications that we aimed to process with our analysis (i.e., the subject domain), and the initial exploratory steps leading towards our choice of the principal approach to be implemented.

The original motivation (goal) was to use some kind of static program analysis to compute data lineage information for enterprise business applications that satisfy the following conditions:

- a large part of the application's business logic is written in Java or C#, and
- heavy usage of relational databases (accessed typically via SQL queries and updated via SQL commands), data warehouses, and other data storage technologies (Excel spreadsheets, plain-text files in a CSV format).

When considering a typical business application, the program code in Java/C# loads data from a specific source (e.g., from a database by an SQL SELECT query), processes and transforms the data in some way, and finally writes the result into a target data storage space (e.g., to another database using an SQL INSERT statement).

However, very quickly we discovered and realized that applications in our target domain use many different enterprise frameworks, not only for the application construction (e.g., Spring [32]), but in particular for data management and processing. The set of popular frameworks includes various object-relational mapping (ORM) libraries such as Hibernate [28] and MyBatis [30] for Java or Entity Framework Core [27] for C#/.NET, frameworks that enable to do streaming- and pipeline-based big data processing for the purpose of data analytics (e.g., Apache Spark [26]). The number of popular data-processing frameworks is really high, and they are quite diverse, as we have found out. Support for many of these frameworks by Java and C# scanners has been always our top priority from the beginning.

Another very important functional requirement for both scanners was the need to accept binary executable programs as input. That means, in our context, Java bytecode in the form of Java class files or JAR libraries, and .NET assemblies compiled from C# source code. Most customers do not want to share the source code of their business applications with Manta, or upload it to the Manta Flow platform. Actually, some customers have always been reluctant to share even just the binaries for testing and debugging purposes.

Given all these requirements, desired features of the scanners, and goals set by Manta, right at the beginning of the project we decided to use techniques of static program code analysis [12] to process the input Java/C# programs and compute the data flow graphs. We neglected other possible approaches, like some kind of dynamic runtime analysis. More importantly, though, in the initial phases of the project, we experimented with two principal approaches to static program analysis: (1) the classic data-flow [13] and pointer analyses [11, 17] implemented in libraries such as Soot [31] and WALA [34], and (2) modular analysis based on symbolic linear interpretation of bytecode in each method of the subject application [14]. We have considered maturity of the available libraries, desirable precision of the data lineage analysis, potential scalability, and performance. Based on our initial experiments and prior academic research experience in the field of static program analysis, we have chosen the second option, that is the modular symbolic analysis, as more promising for the large enterprise applications that we wanted to tackle.

One reason behind our decision was that modular static analysis has been widely considered as scalable to large programs with many classes and methods. In particular, there was already evidence indicating that this approach is useful in practice and can be successfully applied on a large industry scale [9].

We describe the core principles of our modular symbolic data flow analysis in the next section.

3 MODULAR SYMBOLIC DATA FLOW ANALYSIS FOR JAVA/C# BYTECODE

We call the analysis *symbolic* mainly because it uses names and values in the form that appears in the source code or, more precisely, in the compiled bytecode and in the associated debugging information (symbol tables). This includes symbolic names of variables and object fields (attributes), constant string values and integer constants. Our motivation behind this particular approach was to make the analysis results easily readable by users that have access to source code of their applications, meaning that elements of the produced flow graphs refer to input program entities by the same name as in the source code.

The symbolic data flow analysis processes each method (procedure) in the input program separately, one at a time in a modular fashion, and performs symbolic interpretation of the effects of methods' bytecode instructions. For each variable in a given method, the analysis determines the set of all possible sources (e.g., database table columns) for runtime values of the variable during execution of the method. In addition, it also determines the set of possible values for string variables. This is really important and necessary, because string variables are used to store textual representation of SQL statements (both constant and dynamically created), file names, and other information relevant for data lineage.

A curious reader may wonder if the analysis based on symbolic interpretation of bytecode instructions computes useful results that are sound and precise enough. This design decision, like several others, has been influenced by our desire to handle really large enterprise applications by the data lineage scanners. We have believed from the start that our symbolic bytecode interpretation is really sufficient for the purpose of data lineage, and we still believe that our decision was the right one, despite many technical challenges along the way. There is no need for a more complex and precise static analysis (that would be less scalable) in our case, as the data lineage analysis does not have to precisely model control-flow in each procedure — it just needs to process all statements (bytecode instructions) in a linear fashion and collect information relevant for data lineage.

Next, we describe key steps of the whole analysis, including the core algorithm, explain how it really works, and emphasize few important aspects that may not be obvious immediately. The basic principle of the symbolic analysis based on linear bytecode interpretation has been first presented in [14], but the original ideas had to be extended quite significantly for the purpose of data lineage analysis targeting complex enterprise applications, whose implementation uses all the features of Java and C#.

The whole data lineage analysis takes as input the program (binary) code in the form prepared by the extractor, that means a set of classes and libraries, and produces data flow summary for each method in the program. All those flow summaries are then transformed into a flow graph by other modules of the Manta platform. The data lineage analysis itself consists of the following main steps:

(1) preparation of the analysis scope and class hierarchy,
(2) building the call graph of statically reachable methods,
(3) computing the alias analysis for all methods, and
(4) run of the actual modular symbolic analysis that computes a data flow summary for each application method.

After that follows post-processing of method flow summaries and construction of the flow graph by a transformer component (visualizer), but that is not covered in this paper. We describe each of the four steps in more detail below.

Class hierarchy. The analysis scope is just a list of application classes and libraries that the analysis has to process, while the class hierarchy is an internal representation of the analysis scope that provided also important metadata about loaded classes and methods (in addition to their code).

Call graph. In the second step, the program call graph is constructed with a given entrypoint method as the root node, which represents the starting point of program execution. The entrypoint method is designated by the user. For example, it can be the static method main in case of standalone Java applications, or some method defined by the public interface of a service run and managed by an enterprise application framework like Spring. We have tried several approaches to call graph construction within this project. Initially we used the approach that builds the call graph together with pointer analysis data [7, 8, 10], specifically the algorithm 0-CFA implemented by the WALA library, but later we have switched to a simpler approach based on Class Hierarchy Analysis (CHA) [5] and Rapid Type Analysis (RTA) [2]. The primary motivation behind this decision was the need to support dependency injection (autowiring), that means classes and methods added to the analysis scope via mechanisms not fully resolvable just from the bytecode. One popular mechanism, used also by the Spring framework [32], is to specify the actual class names in a declarative way using configuration XML files. When the program is run, such classes and methods are loaded at some point during program execution. More information about handling of dependency injection is provided at the end of Section 4.

Aliasing. We have designed also the auxiliary analysis that computes aliasing information as symbolic and modular. For every application method and for each program code location within the method, the analysis determines all pairs (sets) of symbolic variables and expressions that may be possibly aliased at the code location. It works by processing assignment statements in one or more iterations, propagating aliases transitively over assignments, until it reaches a fixed point. Note that method calls also have to be considered; in particular, the analysis must record the "is-aliased" relation between the return value in a callee method and the variable in a caller used to store the result.

Modular symbolic computation of method flow summaries. For the purpose of computing the data flow summaries in an efficient and scalable manner, we have chosen a variant of the classic worklist algorithm used in many static program analyses [13]. It processes the application methods one-by-one until a fixed point over the flow summaries is reached. Figure 3 shows the top-level iterative worklist-based algorithm for computing method flow summaries.

A key characteristic of the algorithm is that only application methods are ever added to the worklist represented by the variable *queue* (lines 1, 9, 10, and 13) because only application methods have to be processed by the symbolic interpreter of bytecode. The code of library methods is not processed at all by the symbolic analysis. Just the calls of library methods from application code are evaluated, using specific handlers that capture their effects on data flow information.

The algorithm involves usage of two other data structures, method flow summaries (variables *oldSumm* and *newSumm*) and invocation contexts (variable *ctx*). A method flow summary contains three items: (1) a mapping from symbolic variables (expressions) accessed in the method to sets of possible data sources of their values, (2) a set of possible constant string values for each string variable, and (3) a set of possible constant values for each integer variable.

```
1  $queue$ = {each application method in the scope}
2  while not empty( $queue$ ) do
3     $mth$ = removeHead( $queue$ )
4     for $ctx \in$ getInvocationContexts( $mth$ ) do
5        $oldSumm$ = retrieveFlowSummary( $mth, ctx$ )
6        ( $newSumm, cm2newCtxs$ ) = analyzeMethod( $mth, ctx$ )
7        updateFlowSummary( $mth, ctx, newSumm$ )
8        if not equal( $oldSumm, newSumm$ ) then
9           for $ce \in$ getAppCallees( $mth$ ) do $queue = queue \oplus ce$
10          for $cr \in$ getAppCallers( $mth$ ) do $queue = queue \oplus cr$
11       end if
12       for $ce \in$ getAppCallees( $mth$ ) do
13          if not cm2newCtxs[ $ce$ ] = $\emptyset$ then $queue = queue \oplus ce$
14       end for
15    end for
16 end while
```

Figure 3: Iterative worklist-based algorithm for computing method flow summaries

An invocation context for a method captures flow data for its arguments, including the method call receiver object and static variables (fields). Usage of invocation contexts enables the modular analysis to simulate propagation of data values (and the associated flow information) over method call boundaries. Note also that the analysis keeps a set of invocation contexts for each method, in order to capture flow data for distinct argument values, and computes a method flow summary for each context separately. Due to all this, our modular symbolic analysis is partially context-sensitive. Invocation contexts effectively represent call-strings of the length 1. Our motivation behind this limited form of context-sensitivity is to achieve greater analysis precision in the common case of a method called at multiple locations (call sites) and with different sets of argument values (coming from different sources).

In each iteration of the algorithm, the procedure analyzeMethod (line 6) computes the flow summary for a given method and a specific invocation context, and does that in a way that reflects the currently known data flow information (summaries) about all methods. We provide much more details about how this procedure works later in this section, and in the subsequent sections too.

When the newly computed flow summary for a given method *mth* is different from the previous one (see the check at line 8), i.e. when the flow data for *mth* changed possibly due to recently updated information about other methods that are callees or callers of *mth*, then all the callers and callees of the method have to be added (again) to the queue (lines 9 and 10) in order to ensure that their flow summaries eventually reflect this new summary of *mth*. Here, the key principle is that, after update of data flow information for some method, the analysis has to recompute flow summary for all other possibly affected methods to produce sound results. Finally, if some new invocation contexts for some callees of *mth* were recorded during the analysis of *mth*, then all the respective callees are added to the queue for another round of processing.

Symbolic interpretation of bytecode. We have already indicated that our analysis computes a data flow summary of each method using symbolic interpretation of its bytecode (within the procedure analyzeMethod). The interpreter performs linear traversal of the sequence of bytecode instructions for the given method and evaluates the effects of each bytecode instruction on the data flow information associated with symbolic expressions. During the process of analyzing the bytecode of a given method, the interpreter manipulates with a stack of symbolic expressions, which contains operands and results of bytecode instructions. We could say that our symbolic interpreter simulates the execution of method's bytecode with respect to data flow information.

The symbolic analysis of method's bytecode has been designed as partially flow-sensitive; it distinguishes between different program code locations and computes flow data specific for every code location. Even though the symbolic interpreter just traverses the sequence of bytecode instructions once and in a linear fashion, it has a limited support for control-flow branches and loops. More specifically, the interpreter maintains association of each bytecode instruction with a control-flow branch into which it belongs, keeps the flow data for branches separately, and merges the flow data only at join points. In the case of loops, recognized by the presence of a backward jump, a solution that we have decided to use is to compute an over-approximation by taking flow data that exist at the end of the loop body and merge them to all relevant program code locations within the loop body. This approach quite precisely approximates the possible effects (behavior) of multiple loop iterations. Flow data summary for a method is computed by merging the data over all control-flow paths, and thus reflects all possible executions of the method.

```
count = ...
  // [push count to expression stack]
threshold = ...
  // [push threshold to expression stack]

if (count > threshold) {
  // [pop count and threshold from the stack]
  // [new control-flow branch registered]

  data = executeQuery("SELECT * FROM orders");
    // [data: "SELECT * FROM orders"]

} else {
  // [another control-flow branch registered]

  data = loadFile("/tmp/neworders.csv");
    // [data: "/tmp/neworders.csv"]

} // [join point]
// [data: "SELECT * FROM orders",
//        "/tmp/neworders.csv"]
processOrders(data);
```

Figure 4: A fragment of program code with flow data

Figure 4 illustrates the process of interpreting bytecode and computing flow data on a small code fragment that involves an if-then-else statement. All the statements are processed in a sequence from top to bottom. Comments in square brackets describe the effects of bytecode instructions, in particular updates of the symbolic expression stack, flow data information, and control-flow branches.

Flow data propagation. The most important bytecode instructions (program statements) with respect to possible effects on data lineage, which deserve special attention, are these: assignment statements and method calls. We describe our approach to handling these statements (and modeling their effects) in a generalized unified way, as propagation of flow data from source expressions to respective target expressions. In the case of an assignment v := o, the primary source expression is the value o specified at the right-hand side and the primary target is the destination variable v on the left-hand side. The situation is a bit more complicated in the case of method calls, where propagation of flow data over the method call boundaries has to be simulated and evaluated with respect to actual arguments (in the caller), formal parameters (in the callee) and return values. It is done in two separate steps, processing of invoke and return. First, the invocation of a callee method is evaluated, where flow data for actual arguments of the call (source) are propagated to formal parameters (target). The actual receiver object and the formal parameter this are considered too. When the declared target of the call statement is an application method, a new invocation context for the callee is created, filled with arguments' flow data, and then recorded (but only if the new context is different from all the already observed contexts for the callee method). Second, upon processing the return from callee, flow data associated with return expressions in the callee (source) are propagated to the variable in the caller scope (target) where the result of the method call is stored.

While propagation from source to target expressions may seem as quite straightforward, here we need the emphasize the additional really essential aspect of the propagation procedure. It has to consider also field access paths defined over the primary source and target, array access expressions, and even aliasing information. For example, when processing the statement v := o, flow data for the field access expression o.f.g are propagated to v.f.g. In general, we designed the general unified handler for assignment such that it collects the set AE_{src} of all access expressions (to object fields or array elements) over the primary source, including their possible aliases, and for each element of the set AE_{src} propagates flow data to the corresponding expression over the primary target.

Handling of method calls. We want to discuss two more issues related to simulation of method calls that we had to consider hen designing our handler: (1) interfaces and (2) library methods.

The declared target of a method call statement, that means a target specified in the bytecode, may be an interface method or a virtual method. When the declared target method belongs to an interface defined by the application (i.e., not by some library), or when it is an interface possibly implemented by some application class, the handler must consider all the concrete target methods defined by application classes that implement the respective interface. Similarly, in the case of virtual or abstract methods, all concrete implementations defined in subclasses must be considered, We extended our basic handler procedure to accommodate these situations. Flow data are propagated from the caller to every possible concrete target callee method, specifically to the new invocation context specific to each concrete callee. The effects of return are simulated such that flow data in the caller are updated based on the currently available method flow summaries for all the callees together, ensuring that no information about data flow is lost.

Library methods are not analyzed by the symbolic bytecode interpreter, as we already mentioned. Our approach is to model the effects of calls to library methods on data flow information by the means of *semantic descriptions* that are predefined (configured) in advance by Manta engineers. The semantic description of a library method can be either (1) specified in a declarative way using our simple DSL implemented in Java or (2) encoded in the form of Java program code (using the programmatic approach) when the library method's behavior is more complicated. Examples of semantic descriptions are provided in the following sections, for Java core standard libraries in Section 4 and for enterprise data-manipulation frameworks in Section 5. For practical purposes, we have explicitly defined semantic descriptions just for a rather small number of library methods, those deemed by Manta engineers to have effects on data flow information and frequently used in customer applications. Calls of other library methods are processed by so-called identity handler that is applied by default when no explicit semantic description exists for the library method. The identity handler just propagates merged flow data for the receiver object and each argument of the library call to the return value.

Evaluation and remarks. Our experience with the core modular symbolic data lineage analysis, described above in this section, gathered mostly through usage of the Java and C# scanners on applications provided by customers, indicates that the symbolic analysis works well in principle and computes results that are sufficiently precise. But the real technical and software engineering challenges, encountered and observed within this project, were mostly related to precise and scalable handling of program constructs, features, and software development approaches typical for large enterprise business applications. We provide details in the following two sections. First, in the next section, we focus on the program constructs and software development approaches used in the case of large real-world Java/C# programs, and thoroughly describe how they are handled by the scanners. Then, in Section 5, we provide an overview of relevant features specific to large enterprise applications, point out the associated technical challenges, and present our current solutions together with discussion of their benefits and limitations. Although we present the challenges in the context of Manta and its Java/C# scanners, we believe the challenges are more general and especially relevant to the development of any static analysis tool aiming at large enterprise systems written in Java or C#. Some of the challenges are similar to those discussed in recently published studies [1, 3, 18].

4 CHALLENGES RELATED TO ANALYSIS OF REAL-WORLD APPLICATIONS

An obvious implicit functional requirement on the Java and C# scanners has been to ensure they can process real-world applications written in those languages and compute valid data lineage for them. During the course of this project, when trying to fulfill this overall goal, we have encountered numerous technical challenges (kind of) specific to real-world applications that are much more complex than simple (toy) programs that we used for our initial experiments with static data lineage analysis. In particular, those members of the project team coming from the academic background could not really imagine, at the beginning, how many challenges and of what kind specifically we would have to face.

Programming language features. The major challenge that we discuss as the first is the need to properly support all the features of Java [22] and C# [24] programming languages, including those features that are rather obscure, and to support also the Java/JVM [23] and .NET/CLR [25] bytecode sequences (patterns) that compilers may produce. Here, by the word "properly" we mean correctly, without failures, and with useful precision. The list of relevant features includes the basic program constructs, such as object (instance) fields, arrays, control-flow structures (if-then-else, loops), static fields, and so on. But, the list contains also features and programming constructs that turned out to be more tricky to handle, specifically the following:

- abstract classes and interfaces;
- inheritance, with all its trickier aspects like calls of superclass constructors and fields declared in superclasses;
- static constructors (initializers);
- inner classes (which have synthetic references to the outer class and may access its fields);
- lambda methods, some of which implement functional interfaces (e.g., java.util.function.Predicate), and possibly with captured arguments (local variables defined in the enclosing syntactic scope).

We provide more details about the way some of these language features and constructs are handled by the symbolic analysis.

One of the difficulties related to abstract classes and interfaces with multiple concrete implementations is to make sure that flow data are not propagated between methods in an overly imprecise way. Flow data associated with an interface (abstract) method, either in its summary or invocation context, should be computed as union of the flow data for all the concrete implementing methods. But, the content of the flow summary or invocation context associated with the interface method cannot be simply propagated to a particular concrete implementing method (defined in a subclass), because it may refer to entities that belong to another subclass. For illustration, consider the program in Figure 5. It contains interface A with two subclasses B and C, where each of the subclasses declares its own field and implements the method load. The flow data computed for A.load should refer to data sources for both B.f and C.g, set in the respective concrete implementing methods. But it would be overly imprecise to simply propagate all flow data from A.load to B.load, for example, because then flow data for B.load would refer to the unrelated field C.g. Our solution has been to apply carefully designed filters, which omit flow data according to well-defined criteria during propagation in specific directions. The criteria reflect the syntactic scopes in which expressions are visible.

Similarly, flow data computed for the return expression in one concrete implementing method *cm* cannot be simply propagated to

```
interface A {
  void load();
}

class B implements A {
  int f;
  void load() {
    f = executeQuery("SELECT ... ");
  }
}

class C implements A {
  int g;
  void load() {
    g = loadFile("/tmp/data.csv");
  }
}
```

Figure 5: A program that illustrates flow data propagation for an interface method with several concrete implementations

the respective abstract method *am*, because the return expression in *cm* may involve, e.g., some local variable that would not make sense in flow data for the return expression of *am*. The carefully designed filters, applied in propagation, help to avoid many spurious entries in the computed flow data.

In the case of lambda methods in Java, the main challenge was to determine a correct mapping between functional interface methods (e.g., Supplier.get and Function.apply) and synthetic compiler-generated methods that represent implementations of lambda expressions at the bytecode level. For that purpose we had to create an auxiliary modular symbolic analysis that computes this mapping for variables of function object types in advance. This auxiliary analysis follows the same principles as the main symbolic data flow analysis, that means usage of worklist, iteration until a fixed point is reached, and propagation of information over assignment statements and method call boundaries. Captured arguments of lambda methods are supported by flow data propagation from the enclosed syntactic scope to their invocation contexts, like for standard methods. Note that delegates in C#/.NET are handled in a very similar way to lambda methods in Java/JVM.

We also want to highlight one important aspect of symbolic evaluation of array access expressions. The analysis maintains flow data just for those individual elements accessed through indexes explicitly specified in the bytecode. But the problem is that many different expressions used as an index may, in fact, refer to the same element. Our solution was to use the *covering relation*, which for every observed array index expression determines all the affected elements with respect to flow data propagation. For example, the array expression a[i] covers a[0] and a[1], so any change to flow data for a[0], e.g. through explicit assignment, therefore affects also flow data for a[i] and vice versa.

Basic libraries. Another major challenge has been support for basic libraries, including strings, collections, file I/O, databases (the JDBC API [29] in case of Java), and few other.

Precise modeling of string operations, such as concatenation, and their possible results is especially very important for data lineage analysis, because data flow endpoints used in Java/C# programs are identified by strings. Consider, for example, text of SQL queries and commands, names of database tables, and paths to files. An SQL query may be constructed like this (in Java): String query = "SELECT * FROM orders" + year + " WHERE price > " + minPrice. String constants are concatenated together with values of several variables. To achieve high precision and reduce over-approximation as much as possible, and to make space for non-trivial optimizations, we have decided to implement handlers for string operations directly in the scanners in the form of Java code. String concatenation is evaluated by generating all the possible sequences (combinations) from input string fragments. Figure 6 shows an example that involves two string variables that together encode the name of a database table. Each of these two variables has several (2-3) possible string constant values tracked in their associated flow data. The result of concatenation is stored into the flow data for the variable query.

```
String tabNameBase = ...
  // ["orders", "invoices"]
String tabNameYear = ...
  // ["2020", "2021", "2022"]

String query = "SELECT * FROM " + tabNameBase
               + "_" + tabNameYear +
               " WHERE customer_id > " + cId;
  // ["SELECT * FROM orders_2020
             WHERE customer_id = UNDEF",
        "SELECT * FROM orders_2021
             WHERE customer_id = UNDEF",
        ...
        "SELECT * FROM invoices_2022
             WHERE customer_id = UNDEF"]
```

Figure 6: Evaluation of string concatenation

The example in Figure 6 shows one additional feature of our handler for string concatenations. Note the variable cId without any flow data, especially without any known possible constant value. In such cases, the handler uses the special constant __UNDEF__ as the default value. When the user observes this constant in the data lineage graph, it serves as an indicator that the analysis could not determine any possible string constant value for the respective variable. The reason may be that the actual value of the variable is not explicitly defined anywhere in the bytecode (source code), but provided as input at runtime.

Sound and reasonably precise modeling of collections is also needed for data lineage analysis, simply because programs that access databases or files typically use collections to store data in memory and pass them around. Taking into account common usage patterns of collections that we observed in enterprise applications, we have determined that, besides the content of collections, our model has to support also iterators.

Our basic approach to modeling collections uses an abstraction that represents all the concrete elements (stored values) by a single abstract summary node (element). It does not distinguish individual elements. Therefore, flow data for the summary element are equal to the union of flow data for all objects ever added to the collection, over-approximating its possible runtime content. This model for collections is expressed using the mechanism of semantic descriptions for individual operations. A semantic description for a particular operation with collections, that means a handler for calls of the corresponding library method, specifies how the flow data both for the collection variable (a method call receiver) and for the operation's return value should be updated to reflect flow data of the call arguments.

For illustration, the basic variant of the semantic description for the call v = Map.get(k) in a human-readable form looks like this:

$$\texttt{propagation} : \texttt{from} - \texttt{collection} - \texttt{to} - \texttt{returnvalue}$$

Since only the union of elements' flow data is maintained in the basic model, the description reflects the abstract non-deterministic mapping of any possible key to any possible value, and therefore flow data for the whole collection are propagated to the return value of Map.get.

The semantic description for List.add(o) looks like this:

$$\texttt{propagation} : \texttt{from} - \texttt{argument} - \texttt{to} - \texttt{receiver}$$

Flow data of the newly added element are merged into flow data of the whole collection (receiver object).

Later during the course of this project, we have extended the basic approach to modeling collections by precise tracking of flow data for elements that are only ever accessed through a constant key or index. This was motivated by the need to improve precision of analysis results for some data-processing frameworks that take method call arguments (or provide results) in the form of maps with specific constants as keys. The extended model for collections has these main features:

- it stores flow data separately for keys and values in maps,
- precisely tracks the association of constant keys (in case of maps) and numeric indexes (in the case of lists) to flow data for values, and
- keeps information about iteration order for collections that are initialized just with constants in a specific order (using some form of constant initialization block).

Specifically, flow data for values associated with constant keys are not represented by the summary node.

Our extended model also tracks flow data for field access expressions over the collection elements.

To increase the analysis precision even further, based on typical usage of collections and strings together observed in customer applications, we have added support for precise evaluation of string concatenation in the case of collections (i) that contain only string constants as elements and (ii) for which there is just a single known exact iteration order.

Dependency injection and external configuration. Modern real-world applications use also many other advanced features and constructs, like dependency injection and loading configuration from external sources (e.g., XML documents), that make data lineage analysis more difficult.

Here we focus on dependency injection mechanisms provided by the very popular Spring framework [32]. The most relevant aspect is the definition of implementation classes for interfaces through autowiring. Consider the small example program in Figure 7. The field svc in the class Application is annotated with @Autowired, so there is no explicit assignment of an implementing object in the source code, but the dependency injection framework would search for some implementation (HttpService in this example), create an instance, and then assign it to the field using reflection. All that is done automatically behind the scenes by the framework.

```
interface Service {
  int handleRequest(String payload);
}

class Application {
  @Autowired
  Service svc;

  void runWorker() {
    String data = waitForRequest();
    int res = svc.handleRequest(data);
  }
}

class HttpService implements Service {
  int handleRequest(String payload) { ... }
}
```

Figure 7: Dependency injection through autowiring

For the purpose of computing sound data lineage for such applications, it is necessary to (1) collect all fields of interface types with "autowired" values, (2) for each of the respective interfaces find all possible concrete implementing classes in the analysis scope, and (3) then add concrete methods of every implementing class to the call graph and worklist. Our symbolic analysis determines the set of all possible implementing classes for a given interface based on the class (type) hierarchy. The currently implemented support for dependency-injection frameworks within the data lineage scanner for Java is fully covered in [15].

5 CHALLENGES RELATED TO ANALYSIS OF LARGE ENTERPRISE APPLICATIONS

We have already mentioned that another group of technical challenges that we had to face (in our work on data lineage scanners) is related specifically to large enterprise applications that involve business logic written in Java or C#. Such applications manipulate with large amounts of complex data in order to automate business processes [6], and thus represent one of the main target application domains for data lineage analysis.

Frameworks. A really big never-ending challenge (still ongoing) is the need to support many data processing and manipulation frameworks (libraries) used in enterprise applications. The list of popular frameworks includes JDBC API [29], Spring JdbcTemplate [33], MyBatis [30], Hibernate [28], Apache Spark [26], and Entity Framework Core [27] (for C#/.NET). Like for the basic libraries, we use semantic descriptions that capture the effects of calls to framework methods on the data flow information.

For illustration, we take as examples two methods provided by the JDBC API. Our semantic description for the method Statement.executeQuery(String sql) just specifies that (1) a new data flow source is created for every possible concrete string value of the argument sql, (2) each of the data sources is linked to information about the database connection, and (3) then everything is propagated to the return value of type ResultSet. The semantic description for ResultSet.getInt takes the data source associated with a method call receiver (that represents a set of SQL queries) and makes its fresh copy, refines this copy with identification of a database table column (index or name), and propagates the refined information about data sources to the return value.

In general, we strive to define the semantic description (handler) of data flow effects of each framework method in a way that corresponds as closely and precisely as possible to its API documentation, resorting to imprecise over-approximation just when needed to express the respective behavior in a feasible way for the purpose of static analysis.

Semantic descriptions for public methods (API) of every single framework are grouped together in the form of a *data lineage analysis plugin* that is responsible for handling and processing just everything related to the particular framework. Note, however, that for each supported framework we have explicitly defined handlers just for a subset of its API, focusing on the API operations that are really used in customer applications. In the case of most plugins that we have developed so far, semantic descriptions combine a declarative part (written in a simple DSL) with rather complicated program code that implements the DSL and performs actual modifications of respective flow data objects. The plugins also have to load and process relevant metadata, such as definitions in XML files and configurations in Java property files.

Probably the most challenging aspect from a high-level software engineering perspective has been the need to support many frameworks, each based on a different approach (paradigm) to data manipulation. Consider, for example, the MyBatis persistence framework where object-relational mapping (ORM) definitions are written in rather complicated XML documents, and the Apache Spark framework centered around pipeline processing of data frames.

Callbacks. Next, we focus on a common feature of many enterprise data-manipulation frameworks, usage of callback methods for application-specific operations with data. An example of typical usage of callbacks in the context of Spring JdbcTemplate is presented in Figure 8. The call of the framework API method query at line 6 gets also the application-specific function object that encapsulates the callback method processRow. When this callback is invoked by the framework, it receives a ResultSet object that represents one row of the SQL query result.

Very early we have found out that precise modeling of the effects of callbacks from libraries (frameworks), within the context of symbolic data lineage analysis, is hard for the following reasons.

```
1  void testQueryWithCallback() {
2    DataSource ds = new OracleDataSource();
3    // configuring the data source (URL, ...)
4    JdbcTemplate jt = new JdbcTemplate(ds);
5
6    res = jt.query("SELECT * FROM orders",
7              new OrdersRowCallbackHandler());
8  }
9
10 class OrdersRowCallbackHandler
11         implements RowCallbackHandler {
12   void processRow(ResultSet rs) {
13     ... = resultSet.getString(1);
14   }
15 }
```

Figure 8: Usage of callbacks by Spring JdbcTemplate

- There is no edge in the call graph that directly connects (i) application method *am* that invokes framework method *fm* with (ii) callback method *cm* possibly invoked by *fm*. In our example, there is no call graph edge from testQueryWithCallback to processRow.
- It is not possible to say where exactly in the scope of the library (framework) method the application-defined callback method is really invoked, since we do not inspect the code (implementation) of library methods. In addition, we cannot tell in which order are the callbacks invoked (when there is more than one).

Therefore, we have decided to use a solution that computes a relatively coarse over-approximation both for the callback method and for the library (framework) method that invokes the callback. We present the key ideas of our solution.

- Input flow data for the callback include flow data of the receiver object and every argument of the "invoker" framework method.
- Flow data associated with the result of the framework method are propagated into flow data for the receiver object and every argument of every callback possibly invoked from the framework method.
- Flow data for the receiver object and for every call argument of the framework method (including field access paths over these expressions) are augmented with flow data that model the result of callback's invocation.
- For every callback possibly invoked within execution of a framework method, flow data representing the result of this callback are propagated (i) to the result of the whole framework method and (2) to arguments of every other callback invoked by the framework method.
- Finally, input flow data for a specific callback are computed by merging the data over all invocations of library (framework) methods that may invoke the callback.

We have implemented all these ideas within plugins (handlers) for the respective frameworks and in the core symbolic analysis.

Correct symbolic data flow analysis of the actual callback methods is ensured too, in a way described here. For each application method *am*, the analysis keeps a set of possible callbacks from all library and framework methods called by *am*. In addition, for each application method used as a callback somewhere, the analysis keeps a set of application methods that may "trigger" it via some library call. This enables proper updates of the worklist and other data structures:

- When the flow summary for application method *am* changes, meaning that input flow data for library methods called by *am* may have changed too, every relevant callback method is added to the worklist.
- Also when the flow summary of method *cm* used as a callback is updated, then every application method that may "trigger" execution of *cm* via some library call is added to the worklist.

In hindsight, correct, efficient, and reasonably precise handling of callbacks has been one of the most difficult challenges from the algorithmic point of view. Both researchers and engineers have spent really a lot of time tweaking the algorithms, design and implementation (when trying to get it right).

Performance and scalability. While the modular symbolic data flow analysis works well in principle, it does not scale well enough for really large enterprise applications. Here we are talking about applications that consist of thousands of classes and tens of thousands methods. The size of an input application, in terms of the total number of Java/C# methods, greatly influences the running time of data lineage analysis, because most of the application methods are processed several times (even 20-30 times in some cases) before the top-level algorithm reaches a fixed point over the method flow summaries.

Therefore, we have tried many algorithmic optimizations during our work on this project, and evaluated their benefits for performance and scalability of the data lineage analysis on customer applications. We discuss few of the optimizations that we designed and implemented, in particular those with a really big impact, below in this section and in the next one.

The total running time and memory consumption of the symbolic data lineage analysis depends very much on the number of symbolic expressions for which it tracks and propagates flow data. Note that flow data include also possible string values and other information, they contain more than just identification of data sources (text of SQL statements, file names, etc). So an obvious idea was to reduce the number of such "tracked' expressions, but the real challenge here was the design and especially implementation of a procedure that would compute the set of relevant symbolic expressions, i.e. those for which data flow information needs to be tracked, in a way that is (1) fully automated, (2) correct, (3) efficient, and (4) sufficiently precise. After much thought and many experiments (research), we have decided to use the following approach:

(1) Tracking flow data for symbolic expressions used as arguments for operations with data sources and endpoints, arguments to calls of data-processing frameworks, and expressions used to store the results of such operations.
(2) Then also tracking flow data for all symbolic expressions (A) that those in the first group depend on transitively through assignments or (B) into which some from the first group may be propagated.

Our approach has been greatly inspired by the algorithms for program slicing [16], both forward and backward. Note also that the final set of tracked symbolic expressions is computed gradually over the run of symbolic analysis. It may be extended in each iteration of the top-level worklist algorithm. This way, the analysis can still track all the important data lineage information that the scanners should report, capturing in the graph all the data endpoints and the flow of data between them, while ignoring possible values of the irrelevant program expressions.

Another optimization that has a really great benefit is usage of the *subsuming relation* over invocation contexts. The key principle is this: when the analysis gets to the state in which there would be two different invocation contexts, c_1 and c_2 for method m, and c_2 subsumes c_1, meaning that c_2 contains all the information stored in c_1 and possibly more, then it suffices to keep just c_2 and merge all the data flow information associated with c_1 into that for c_2.

6 IMPLEMENTATION

Manta engineers and university students have implemented the whole symbolic data lineage analysis, including all the algorithmic optimizations described above, within the proprietary closed-source Manta Flow platform. The current implementation features also many low-level performance optimizations that proved to be very useful — for example, a copy-on-write mechanism for data structures that capture flow information (summaries, invocation contexts), and extensive caching of intermediate results of various computations performed as steps of the analysis.

All the developers together have created a large test suite, which includes (1) unit tests for small code fragments (individual methods and classes, small pieces of functionality) but also (2) many integration tests. The integration tests are designed to validate the complete run of a scanner on small input programs manually prepared by the developers. The input programs used by tests have around 30-50 lines of code each, covering all the different features of Java/C#, supported core libraries (strings, collections), and calls of the supported operations of data-processing frameworks. Validation of the scanner output is based on comparing the computed flow graph against manually defined expectations in the form of assertions over the flow graph. The assertions check the overall flow graph structure, existence of nodes with specific content (that represents specific data endpoints), and existence of paths (sequences of edges) between specific pairs of nodes.

7 EXPERIENCE AND LESSONS LEARNED

In this final section, we discuss the key points of our experience gathered so far during the whole process of developing the scanners for Java and C#, and share the most important ("take-away") lessons that we have learned and that may be interesting for others.

During our work on the scanners, a really large amount of work (lot of time and effort) has been dedicated to development of performance and scalability optimizations, many times sacrificing precision for performance and vice versa, in order to make sure that scanners are able to compute precise and useful data lineage graphs within practical time bounds. Especially testing and debugging of the performance issues, optimizations and precision improvements on large applications requires lot of time and effort. Also the process of defining the semantic descriptions (handlers) for selected individual methods (API) of data manipulation and processing frameworks is very time consuming.

Precision of the computed data flow graphs inherently depends on the level of approximation that is applied during a run of static analyses (caused, e.g., by merging flow data over all control-flow paths within a single method). While many features of the static analyses used in Manta scanners have some effect on the precision of the result, here we want to highlight especially the following two as very important in practice: (1) approximation introduced by models (semantic descriptions) of library procedures for string manipulation and (2) merging flow data (summaries) computed for different invocations (call contexts) of individual application methods. Note that, without high level of over-approximation and quite aggressive performance and scalability optimizations that affect precision of the resulting flow graphs, the static analyses performed by scanners would be absolutely infeasible.

The whole endeavour of trying to process large enterprise applications, provided by customers, has shown the limitations of modular static analysis in practice. For some of the large applications, our scanners run for tens of hours (few days), need really a lot of memory (tens of GBs), and produce just partial results. Our experience with the design and implementation of data lineage scanners (analysis) shows that, while the design of some analysis optimizations may be quite straightforward, the real challenge is to create a both correct and efficient implementation that provides observable benefits when scanners are used on large applications.

ACKNOWLEDGMENTS

This work was supported by Manta through a contractual research project with Charles University. We would also like to thank all the employees of Manta and students who have been involved in this project for their contributions.

REFERENCES

[1] A. Antoniadis, N. Filippakis, P. Krishnan, R. Ramesh, N. Allen, and Y. Smaragdakis. Static Analysis of Java Enterprise Applications: Frameworks and Caches, the Elephants in the Room. In Proceedings of PLDI 2020, ACM.

[2] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In Proceedings of OOPSLA 1996, ACM.

[3] M. Chen, T. Tu, H. Zhang, Q. Wen, and W. Wang. Jasmine: A Static Analysis Framework for Spring Core Technologies. In Proceedings of ASE 2022, ACM.

[4] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. The VLDB Journal, volume 12, number 1, 2003.

[5] J.Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In Proceedings of ECOOP 1995, LNCS 952, Springer.

[6] M. Fowler. Patterns of Enterprise Application Architecture, 1st edition. Addison Wesley, 2002.

[7] D. Grove and C. Chambers. A framework for call graph construction algorithms. ACM Transactions on Programming Languages and Systems (TOPLAS), volume 23, number 6, 2001.

[8] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object-Oriented Languages. In Proceedings of OOPSLA 97, ACM.

[9] M. Harman and P.W. O'Hearn. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In Proceedings of SCAM 2018, IEEE CS.

[10] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In Proceedings of CC 2003, LNCS 2622, Springer.

[11] O. Lhoták, Y. Smaragdakis, and M. Sridharan. Pointer Analysis (Dagstuhl Seminar 13162). Dagstuhl Reports, volume 3, number 4, 2013.

[12] A. Møller and M.I. Schwartzbach. Lecture notes on Static Program analysis. Department of Computer Science, Aarhus University. https://cs.au.dk/~amoeller/spa/ (accessed in October 2023)

[13] F. Nielson, H.R. Nielson, and C. Hankin. Principles of Program Analysis, corrected edition. Springer, 1999.

[14] P. Parízek. BUBEN: Automated Library Abstractions Enabling Scalable Bug Detection for Large Programs with I/O and Complex Environment. In Proceedings of ATVA 2019, LNCS 11781, Springer.

[15] L. Riedel. Extending Data Lineage Analysis Platform with Support for Dependency Injection Frameworks. Master thesis, Charles University, 2021. https://dspace.cuni.cz/handle/20.500.11956/127818?locale-attribute=en (accessed in October 2023)

[16] J. Silva. A vocabulary of program slicing-based techniques. ACM Computing Surveys, volume 44, issue 3, ACM, 2012.

[17] M. Sridharan, S. Chandra, J. Dolby, S.J. Fink, and E. Yahav. Alias Analysis for Object-Oriented Programs. LNCS 7850, Springer, 2013.

[18] J. Wang, Y. Wu, G. Zhou, Y. Yu, Z. Guo, and Y. Xiong. Scaling Static Taint Analysis to Industrial SOA Applications: A Case Study at Alibaba. In Proceedings of ESEC/FSE 2020, ACM.

[19] 6 Benefits of Data Lineage for Financial Services. https://manta.io/blog/6-benefits-of-data-lineage-for-financial-services-manta (accessed in January 2024).

[20] Data Lineage. https://www.techopedia.com/definition/28040/data-lineage (accessed in January 2024).

[21] Why Data Lineage is Crucial for Data Accuracy. https://manta.io/blog/why-data-lineage-is-crucial-for-maintaining-data-accuracy (accessed in January 2024).

[22] The Java Language Specification, Java SE 11 Edition. https://docs.oracle.com/javase/specs/ (accessed in October 2023).

[23] The Java Virtual Machine Specification, Java SE 11 Edition. https://docs.oracle.com/javase/specs/ (accessed in October 2023).

[24] The C# Language Specification, 6th edition. https://www.ecma-international.org/publications-and-standards/standards/ecma-334/ https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction (accessed in October 2023).

[25] The Common Language Infrastructure, 6th edition. https://www.ecma-international.org/publications-and-standards/standards/ecma-335/ https://learn.microsoft.com/en-us/dotnet/standard/clr (accessed in October 2023).

[26] Apache Spark: Unified engine for large-scale data analytics. https://spark.apache.org/ (accessed in October 2023).

[27] Entity Framework Core. https://learn.microsoft.com/en-us/ef/core/ (accessed in October 2023).

[28] Hibernate ORM (Object/Relational Mapping). https://hibernate.org/orm/ (accessed in October 2023).

[29] The Java JDBC API. https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/ (accessed in October 2023).

[30] MyBatis persistence framework. https://mybatis.org/mybatis-3/ (accessed in October 2023).

[31] Soot: A framework for analyzing and transforming Java and Android applications. http://soot-oss.github.io/soot/ (accessed in October 2023).

[32] Spring Framework. https://spring.io/projects/spring-framework (accessed in October 2023).

[33] Spring JDBC package. https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html (accessed in October 2023).

[34] The T. J. Watson Libraries for Analysis (WALA). https://github.com/wala/WALA (accessed in October 2023).