ArCode: Facilitating the Use of Application Frameworks to Implement Tactics and Patterns

Ali Shokri
Rochester Institute of Technology
Rochester, NY, United States
as8308@rit.edu

Joanna C. S. Santos
Rochester Institute of Technology
Rochester, NY, United States
jds5109@rit.edu

Mehdi Mirakhorli
Rochester Institute of Technology
Rochester, NY, United States
mxmvse@rit.edu

***Abstract*—Software designers and developers are increasingly relying on application frameworks as first-class design concepts. They instantiate the services that frameworks provide to implement various architectural tactics and patterns. One of the challenges in employing frameworks for such tasks is the difficulty of learning and correctly using the APIs of the frameworks. This paper introduces a learning-based approach called ArCode to help novice programmers correctly use frameworks' APIs to implement architectural tactics and patterns. ArCode has several novel components: a graph-based approach for learning specification of a framework from a limited number of training software, a program analysis algorithm to eliminate erroneous training data, and a recommender module to help programmers use APIs correctly and identify API misuses in their program. We evaluated our technique across two popular frameworks: JAAS security framework used for authentication and authorization tactic and Java RMI framework used to enable remote method invocation between client and server and other object oriented patterns. Our results demonstrate (i) the feasibility of using ArCode to learn the specification of a framework; (ii) ArCode generates accurate recommendations for finding the next API call to implement an architectural tactic/pattern based on the context of the programmer's code; (iii) it accurately detects API misuses in the code that implements a tactic/pattern and provides fix recommendations. We also demonstrate that ArCode outperforms two famous techniques, MAPO and GrouMiner, on API recommendation and misuse detection tasks.**

***Index Terms*—Software Framework, Architectural Tactics, API Specification, API Usage Model, API Recommendation, API Misuse Detection**

I. Introduction

To satisfy performance, security, reliability and other quality concerns, architects need to compare and carefully choose a combination of architectural patterns, styles or tactics. In the subsequent development phase, these architectural choices must be implemented completely and correctly in order to avoid a drift from envisioned design. Cervantes et al. [1] confirms that software designers and developers are increasingly relying on application frameworks as first-class design concepts to facilitate implementation of architectural tactics and patterns. *Software frameworks* are reusable software elements that provide key functionalities, addressing recurring concerns across a range of applications. They incorporate many architectural patterns and tactics to prevent software designers and developers from implementing software from scratch [1], [2]. For instance, the architecture of most contemporary enterprise applications relies on the *Spring Framework* that provides pre-packaged solutions to implement various architectural concepts ranging from *Model-View-Controller* (MVC) patterns to *authentication* and *authorization* security tactics [3], [4].

Developers use Application Programming Interfaces (APIs) to import and use the frameworks' functionalities [2], [5]. Therefore, programs' quality largely depends on using these APIs correctly [6]. Multiple studies have shown that proper use of a framework's API requires an in-depth understanding of its internal implementation details including underlying architectural patterns and tactics, class structure, and set of tacit sequence calls, data-flows as well as interfaces that need to be implemented [1], [6]–[8]. Recent qualitative and quantitative studies have reported that implementing architectural tactics is more complex compared to delivering software functionalities, and novice and non-architecture savvy developers struggle in implementing architectural tactics and patterns [4], [6]. Soliman et al. [9] indicate that developers rely on sources such as Q&A websites (e.g. StackOverflow) to find information on how to use frameworks, implement tactics and patterns for specific quality attributes [10]. However, prior research have also shed light on the fact that even code snippets from accepted answers of Q&A websites can contain design flaws, bugs or vulnerabilities that get reproduced across multiple software systems that reused that code snippet as-is [11], [12].

The prior work on framework API recommendations [13]–[31] focus on low level, local data structure related concerns and basic utility frameworks used to implement various data structures. This line of work falls short of addressing the challenges of implementing tactics and patterns and has not fully studied frameworks used to bring a new architectural tactic or pattern into a given system design. There are some recommender systems developed by researchers to assist programmers in implementing architectural tactics and patterns [4], [32], [33]. These approaches, however, do not support implementing architectural tactics and patterns using frameworks.

In this paper, we aim to study frameworks with architectural intents that address tactics or patterns. We propose the ArCode approach, aiming to help programmers implement an architectural tactic or pattern using API recommendations. ArCode leverages a novel learning technique to infer an *accurate* API specification model which will be used for generating recommendations. Artifacts of this paper are available at ArCode's repository [34].

The significance of contributions made by this paper is briefly described below:

- To the best of our knowledge, this is the first study focusing on inferring and using API specification of a software framework with architectural implications (e.g., frameworks implement architectural tactics and patterns).
- We present a program analysis approach to reverse engineer a novel **G**RAPH-BASED **F**R**A**MEWORK **API** USAGE **M**ODEL (**GRAAM**), which is an *abstract and semi-formal representation* of how a framework's API is being used in a given program to implement tactics and patterns.
- An automated approach to analyzes the byte code of frameworks and extracts **I**NTER-**F**RAMEWORK **D**EPENDENCY (**IFD**) model. Later the IFD can be used to identify programs that violate a framework's implicit API order constraints.
- An inter-procedural context-, and flow- sensitive static analysis approach to automatically infer a specification model of frameworks from a repository of **GRAAMs**. This inference is performed based on two sources, limited sample programs and a framework byte code.
- An empirical investigation of the usefulness of the proposed approach to recommend APIs and detect API misuses for tactics' implementation on two popular Java-based frameworks, Java Authentication and Authorization Services (JAAS) and Remote Method Invocation (RMI) frameworks.

The remainder of this paper is organized as follows. Section II provides an overview of our approach. Section III briefly describes data collection phase. Section IV presents our graph-based API usage model (GRAAM) for a program. Section V describes ARCODE, our automated approach for inferring a framework API specification model from extracted GRAAMs. An experimental evaluation of the approach is presented in Section VII. Section VIII discusses threats to validity of the approach. Section IX reviews related work. Lastly, Section X concludes this paper.

II. OVERVIEW

ARCODE is designed to perform API recommendation and misuse detection for application frameworks used to implement architectural tactics and patterns. In particular, these frameworks exhibit a great degree of inter-process communication and API interactions beyond a single class, module or process [4]. ARCODE aims to help novice developers, and non-architecture savvy developers to use frameworks. ARCODE learns the API specification of frameworks from limited *sound* program examples. A program is considered **sound** if it uses the framework API correctly to implement the tactic. As shown in Figure 1, the approach has four phases.

❶ **Data Collection and Preparation:** We create a code repository of programs that incorporate the framework of interest. Section VII use *learning saturation* as a measure of indicating how many training projects are required.

❷ **Training Data Pre-Processing and Validation:** Given the training data, we perform a *context-*, *path-*, and *flow-sensitive* static analysis to create an *inter-procedural* graph-based representation of API usages for each program. In this phase, we ensure that only sound training programs will be included in our training data. To guarantee the syntax correctness, we first compile programs and generate their jar files. Moreover, to verify that a program is semantically error free (w.r.t. API usages), we use a novel technique to extract **I**NTER-**F**RAMEWORK **D**EPENDENCY (**IFD**) model from the framework's internal source code. This model encompasses mandatory order constraints of APIs that are enforced by the framework and must be preserved in any given program. This approach identifies and rules out incorrect API usages and API violations. Leveraging the IFD model, we separate sound programs from those that violate implicit rules of API usage required by the framework.

❸ **Training:** For each sound program, we create its **G**raph-based **Fr**amework **A**PI Us**a**ge **M**odel (GRAAM). This model demonstrates how the framework API was used in a given validated training data. Finally, via a recursive learning method the framework API specification model is created from a set of generated GRAAMs.

❹ **Recommendation:** The learnt model is used in this phase to implement a recommender system guiding programmers in using APIs. Specifically, this system analyzes the program under development and recommends what APIs should be considered and how they should be used in that program to correctly implement a tactic or a pattern. Furthermore, if there are API misuses, ArCode will detect and recommend fixes.

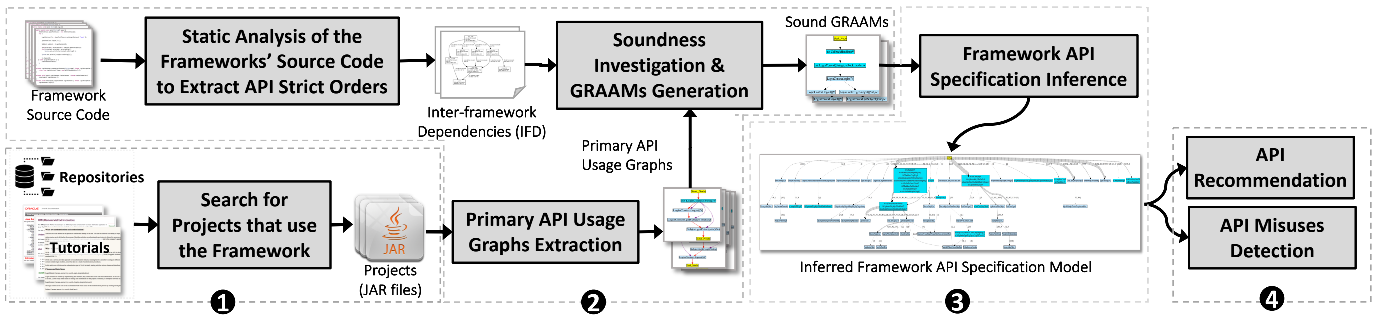
III. DATA COLLECTION AND PREPARATION PHASE

The first phase of this approach is focused on collecting a set of projects that uses a framework of interest. These projects should match the following criteria: *(i)* it imports and uses the framework API, and *(ii)* it is syntactically correct. For checking the first condition, we scan the source code and check for API statements in the code. To guarantee the syntax correctness (second condition), we compile the programs to generate their JAR files.

IV. TRAINING DATA PROCESSING AND VALIDATION PHASE

In the previous phase, we obtained collected projects that use a framework API and are compilable. However, these two characteristics do not guarantee these projects to be *sound*. Therefore, this second phase focuses on finding sound projects that can be used for training.

Distinguishing sound and unsound projects require a sophisticated ***program representation model*** that can capture fundamental information regarding the usage of the framework APIs including but not limited to object instantiation, static and non-static method calls, as well as static and non-static field accesses. These constructs can be scattered over multiple methods, which requires an inter-procedural analysis of the



Framework Source Code
Static Analysis of the Frameworks' Source Code to Extract API Strict Orders
Inter-framework Dependencies (IFD)
Soundness Investigation & GRAAMs Generation
Sound GRAAMs
Framework API Specification Inference
Repositories
Tutorials
Search for Projects that use the Framework
JAR
Projects (JAR files)
Primary API Usage Graphs Extraction
Primary API Usage Graphs
Inferred Framework API Specification Model
API Recommendation
API Misuses Detection
1
2
3
4

Fig. 1: Overview of our approach (ARCODE)

program. Such a representation must also capture implementation of interfaces as well as inheritances of abstract classes of the framework. Moreover, this model shall be able to represent relationships between the framework APIs, allowable data dependencies between these methods, the logical and a usecase-driven order of method calls, and semantically equivalent sequences of API calls.

In this phase, we present an approach to pre-process the training data to (1) extract the above elements, and (2) validate the training data so projects which incorrectly implement tactics/patterns can be eliminated from the training process.

A. *Motivating Examples*

Listing 1 and Listing 2 show two real-world code snippets of the *authentication* tactic [3] implemented using the JAAS framework. The correct implementation of this security tactic requires a careful sequence of API calls and manipulations of data objects. Although these two code snippets implement authentication tactic in a different code structure, they both: create a Subject object and an object that implements the CallbackHandler interface and pass them to the constructor of the instantiated LoginContext object; and then call the login() method from LoginContext.

In addition to the above statements, the code in Listing 2 goes further and calls the getSubject() method from a LoginContext object (line 11) and calls the getPrincipal() method from a Subject object (line 12).

From these two examples, we observe the followings:
- **Observation #1:** Only a subset of statements in a program (the highlighted lines) are related to the framework of interest and should be part of a framework API specification model.
- **Observation #2:** The framework-related statements could be scattered across multiple methods. Thus, the framework API specification model shall be able to interconnect these statements that are within different scopes.
- **Observation #3:** Multiple (sub) programs with the same behavior might be written slightly different. For instance, both code snippets in Listing 1 and Listing 2 create Subject and CallbackHandler objects to be passed to the constructor of LoginContext. Although the order of object instantiation in two code snippets is different, it does not change the behavior of the program.

```
1.  public class TestJaasAuthentication {
2.    public static void main(String[] args) {
3.       String user = System.getProperty("user");
4.       String pass = System.getProperty("pass");
5.       boolean loginStatus = true;
6.       try {
8.           LoginContext loginContext = getLoginContext(user,pass);
9.           loginContext.login();
10.      } catch (LoginException e) { loginStatus = false; }
11.       if(loginStatus) System.out.println("Login Successful.");
12.       else System.out.println("Login Failed.");
13.   }
14.   private static LoginContext getLoginContext(String u, String p) throws
        LoginException{
15.       CallbackHandler handler = new RanchCallbackHandler(u, p);
16.       Subject subject = new Subject();
17.       LoginContext lc = new LoginContext("RanchLogin", subject, handler)};
18.       return lc;
19.   }
20. }
```

Listing 1: *Sample #1: A code snippet that implements the Authentication tactic using JAAS framework in multiple methods*

```
1.  public class LoginUsecase {
2.    private static Logger LOGGER = Logger.getLogger(LoginUsecase.class);
3.    public static void main(String[] args){
4.       BasicConfigurator.configure();
5.       LoginContext lc = null;
6.       System.setProperty("java.security.auth.login.config", "jaas2.config");
7.       try{
8.           Subject subject = new Subject();
9.           lc = new LoginContext("rainyDay2", subject, new JAASCallbackHandler(
              "user1", "pass1"));
10.          lc.login();
11.          Subject subject = lc.getSubject();
12.          subject.getPrincipals();
13.          LOGGER.info("established new logincontext");
14.      }
15.      catch (LoginException e){
16.          LOGGER.error("Authentication failed " + e);
17.      }
18.   }
19. }
```

Listing 2: *Sample #2: A code snippet that implements the Authentication tactic using JAAS framework in a single method*

Therefore, it is necessary to develop an API usage representation model that adequately captures correct usages of frameworks' APIs while taking these concerns into account. We developed a novel **Graph-based Framework API Usage Model (GRAAM)** to address these concerns.

B. *Pre-Processing and Validation*

This pre-processing step identifies APIs that are used to implement tactics and patterns in a program, validates their usage based on several ground-truths obtained from the framework's source code, and creates *Graph-based Framework API Usage Models (GRAAMs)* for correct API usages.

We follow a four-step process: **(1)** System Dependence Graph (SDG) extraction, **(2)** Slicing of the SDG, **(3)** Removal of API usage violations, and **(4)** Generation of Graph-based Framework API Usage Models (GRAAMs).

1) System Dependence Graph (SDG) Extraction: First, we perform an *inter-procedural* static analysis on a program and extract its context-sensitive *Call Graph* using 1-CFA algorithm. A call graph is a directed graph representing relationships between caller and callee methods in a program [35]. A 1-CFA context-sensitive callgraph [36] distinguishes between situations where a method m_3 is being called from m_1 or m_2. This type of call graph considers the possibility of different behaviors of the program in callee method (e.g. m_3) based on different caller methods (e.g. m_1 or m_2). For the sake of scalability, we did not choose a higher sensitivity of context (i.e., n-CFA for $n > 1$).

Next, we compute the System Dependence Graph (SDG) of the program under analysis. An SDG is a directed graph representing a whole program [37]. The nodes in this graph are *statements* in the program and the edges are either *data* or *control* dependencies between nodes.

Since we incorporate a context-sensitive call graph, the constructed SDG holds the following characteristics:

– *flow-sensitive*: it accounts the order of execution of statements in the program being analyzed;
– *context-sensitive*: it distinguishes different call sites. The same method *m* can be invoked by different methods (call sites). Hence, *m* is analyzed differently based on its corresponding call site.
– *inter-procedural*: it represents the system as a whole, interconnecting statements within different methods based on the caller-callee relationships;

We use T. J. Watson Libraries for Analysis (WALA)[1] to construct SDGs with the aforementioned attributes. We chose WALA over other tools (e.g. Soot [38]) because it provides built-in supports for extracting SDGs from a 1-CFA call graph as well as different versions of Java language (e.g. Java 8).

2) Slicing the Extracted SDG: In this second step, we compute a slice[2] of the program *p* under analysis that includes all statements *s* in the SDG that are either (i) a *framework-related* statement; or (ii) that may be *affected by* a framework-related statement or (iii) that *affects* a framework-related statement. **Framework-related Statements** are the statements s_f that match the conditions (a)-(f) listed below. Each statement s_f can be related with the framework either ***directly*** (cases *a*, *b*, and *c*) or ***indirectly*** via inheritance (cases *d*, *e*, and *f*):

(a) it invokes a method from a *framework data type* (classes or interfaces declared in the framework);
(b) it instantiates an object from a *framework data type*;
(c) it accesses a field from a *framework data type*;
(d) it invokes a method implemented by an application class that inherits or implements a *framework data type*;
(e) it instantiates an object from an application class that inherits or implements a *framework data type*;
(f) it accesses a field from an application class that inherits or implements a *framework data type*;

After computing a slice of the SDG, we remove all the nodes (statements) in the remained graph that are not framework-related yet keeping the direct and indirect dependencies between framework statements. The outcome of this process is the program's **Primary API Usage Graph** $g = (V, E)$, which is a directed labelled sub-graph of the SDG, with nodes $v \in V$ and edges $e \in E$ where $E \subseteq V \times V$. The set of nodes V in a primary API usage graph is partitioned into three types: start node V_{start}, end nodes V_{end} and framework API usage nodes V_f:

(1) A ***start node*** $v_{start} \in V_{start}$ represents the begining of the framework usage in a program. Each primary API usage graph starts with a single *start node*;
(2) Each ***end node*** $v_{end} \in V_{end}$ indicates the termination of the framework usage. Each primary API usage graph ends with one or more *end node(s)*;
(3) Each ***framework API usage node*** $v_f \in V_f$ denotes a framework-related statement in a program (i.e., $V_f = S_f$). Each v_f has an associated ***instruction type*** $type(v_f)$ (i.e. *object instantiation*, *field access* or *method invocation*). Furthermore, each v_f has a ***target framework data type*** $target(v_f)$, that depends on the instruction type. In case of object instantiations, $target(v_f)$ is the framework class type of the object; for method invocations, $target(v_f)$ is the target of the call; and for field accesses, the $target(v_f)$ is the framework type that contains the field.

The edge set E in a primary API usage graph has two partitions: ***sequence*** edges $E_s \subseteq (V_{start} \times V_f) \cup (V_f \times V_{end}) \cup (V_f \times V_f)$ and ***data dependency*** edges $E_d \subseteq V_f \times V_f$:

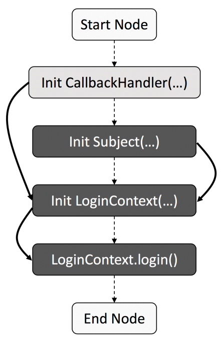
(1) A data dependency edge $e_{data} = v_{src} \xrightarrow{data} v_{dst}$ indicates that v_{dst} uses data that has been defined by v_{src}.
(2) A sequence dependency $e_{seq} = v_{src} \xrightarrow{seq.} v_{dst}$ indicates that v_{dst} is used after v_{src} in the program.

Figure 2a and Figure 2b show the primary API usage graphs for the code snippets in Listing 1 and 2. We use solid edges to show data dependencies and dashed edges to indicate sequence dependencies. For example, since the CallbackHandler object is used in the LoginContext's constructor, there is a data dependency edge between init CallbackHandler and init LoginContext nodes in the primary API usage graphs. Nodes with a lighter background represent statements that are related with the framework indirectly (via inheritance).

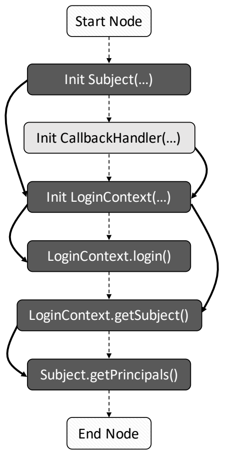
3) Removing API Usage Violations: We explained so far how we analyze a project to extract its API usage information and represent it as a primary API usage graph. However, we want to eliminate any erroneous API usages in our training corpus. Therefore, we need a reliable ground truth to identify such incorrect usages in a code repository and filter them out from the training data.

ArCode analyzes the framework's source codes and captures the API usage rules that are not visible to the developers, but are implicitly reflected in the source code of frameworks. To do so, it performs a static analysis of the framework's

[1]http://wala.sourceforge.net
[2]A program slice includes only the set of statements that may affect a point of interest of the program (referred as the slice criterion) [39].



(a) Primary API usage graph for Listing 1



(b) Primary API usage graph for Listing 2

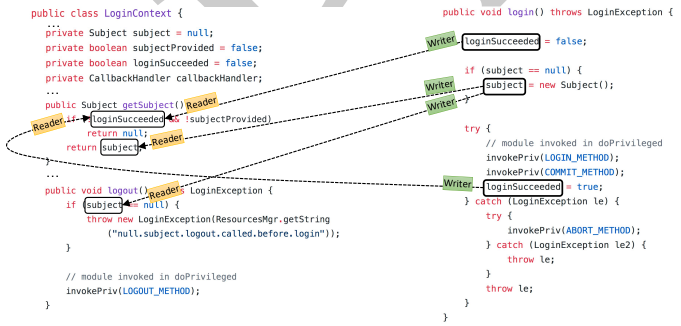
Fig. 2: Extracted primary API usage graphs from sample code snippets

source code to capture *implicit* data dependencies between APIs of a framework. Since this information is obtained directly from the framework, not from programs that use the framework, it could be considered as a ground truth for identifying API misuses.

The main idea is to leverage **reader-writer** roles of API methods inside a framework to find dependencies between them. Using these dependencies, one can find partial API strict orders that must be followed in a program. Therefore, API misuses in a program could be identified with confidence by finding violations from these strict orders. *Writer* and *Reader* methods are defined as:

– ***Writer method:*** A method is considered as a writer regarding a specific class *field* if it changes the value of that specific field somewhere in its body.

– ***Reader method:*** A method is considered as a reader regarding a specific class *field* if it uses the value of that specific field somewhere in its body.



```
public class LoginContext {
  ...
  private Subject subject = null;
  private boolean subjectProvided = false;
  private boolean loginSucceeded = false;
  private CallbackHandler callbackHandler;
  ...
  public Subject getSubject() {
    if (!loginSucceeded && !subjectProvided)
        return null;
      return subject;
  }
  ...
  public void logout() throws LoginException {
      if (subject == null) {
          throw new LoginException(ResourcesMgr.getString
              ("null.subject.logout.called.before.login"));
      }

      // module invoked in doPrivileged
      invokePriv(LOGOUT_METHOD);
  }
```

```
public void login() throws LoginException {

    loginSucceeded = false;

    if (subject == null) {
        subject = new Subject();
    }

    try {
        // module invoked in doPrivileged
        invokePriv(LOGIN_METHOD);
        invokePriv(COMMIT_METHOD);
        loginSucceeded = true;
    } catch (LoginException le) {
        try {
            invokePriv(ABORT_METHOD);
        } catch (LoginException le2) {
            throw le;
        }
        throw le;
    }
}
```

Fig. 3: Extracting Inter-framework Dependencies between getSubject(), logout(), and login() methods which are three API methods inside LoginContext class (JAAS framework)

As an example, Figure 3 shows three methods from LoginContext class in the JAAS framework. This class has various fields including subject and loginSucceeded. While method

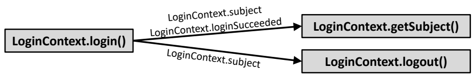


Fig. 4: IFD Model based on Fig. 3

login() assigns values (*writes*) to subject and loginSucceeded fields, methods getSubject() and logout() use (*read*) these fields' values. Thus, getSubject() and logout() methods are dependent to some data generated in login() method. We extract these information and create our **Inter-Framework Dependency (IFD)** model. The IFD built for the framework's code shown in Figure 3 is depicted in Figure 4.

We use IFD model for identifying incorrect programs w.r.t. a framework usage. If a programmer has a framework-related statement s_2 in a program before another framework-related statement s_1 while based on IFD model s_2 reads data generated by s_1 ($s_1 \xrightarrow{\text{data}} s_2$), then, that program is considered as an incorrect program.

4) Generating Graph-based Framework API Usage Models (GRAAM): The last step of pre-processing and data validation phase focuses on creating a representation of each training data, a Graph-based Framework API Usage Model (GRAAM), which can be used by ARCODE's learning algorithm. We provide a definition for a GRAAM further in this section.

As discussed earlier, it is possible that two programs with different sequences of framework-related statements implement the same tactic. For example, although Listing 1 and Listing 2 implement the same use case, the instantiation order of Subject and CallbackHandler in their programs and so, in their corresponding primary API usage graphs (Figure 2a and Figure 2b) is different. We call these sequences of API usage nodes as **Semantically Equivalent API Sequences**. Two sequences of framework usage nodes, seq_1 and seq_2, are semantically equivalent if both conditions are true:

- there is a *bijection* between the nodes in seq_1 and seq_2. Each paired nodes $s_1 \in seq_1$ and $s_2 \in seq_2$ have the same *type* and *target framework type*, i.e. $type(s_1) = type(s_2)$ and $target(s_1) = target(s_2)$; and
- the two sequences are isomorphic considering only the data dependency between APIs in each sequence.

As a result, a **Graph-based Framework API Usage Model** (GRAAM) is a directed graph g which has the same set of nodes as a *primary API usage graph*, but a different edge type: **API Order Constraint** edges. Semantically equivalent API sequences have a single representative in a GRAAM. In other words, two different programs with the same behavior (w.r.t. APIs of the same framework) have isomorphic GRAAMs.

For example, parts of the programs in Listing 1 and Listing 2 including instantiation of Subject, CallbackHandler, LoginContext, and calling login() method, implement the same tactic (authenticate actors). Thus, their corresponding sub-GRAAMs are the same. This property of GRAAM enables us to have the same representation for semantically equivalent API usages in different programs. Hence, when we look at a

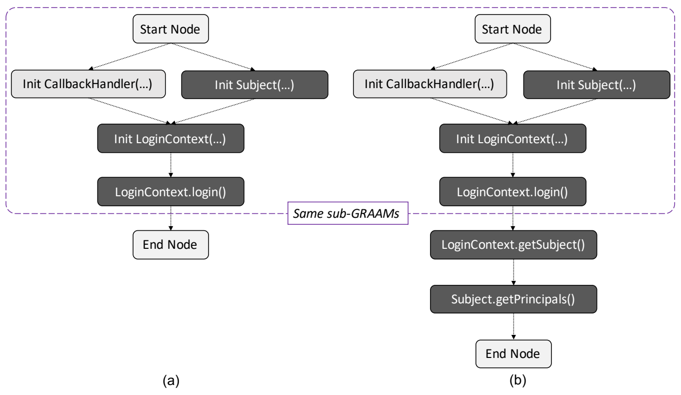


Fig. 5: Built GRAAMs for programs in (a) Listing 1 and (b) Listing 2. Since both programs use *semantically the same* API sequences for creating a LoginContext object and calling login() method, their corresponding Sub-GRAAMs confined by the purple dashed line are isomorphically the same.

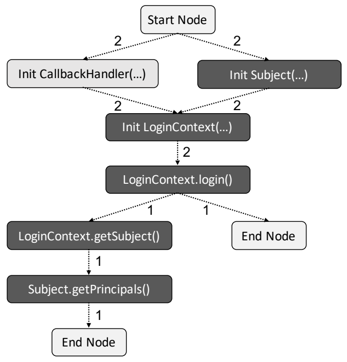


Fig. 6: Built Framework API Specification Model (FSpec) from depicted GRAAMs in Figure 5a and Figure 5b

code repository, we can correctly identify the same API usages and compute their frequencies.

To create a GRAAM, we **(i)** start with a validated primary API usage graph, **(ii)** remove all the sequence edges except edges from start node and edges to end node(s), and **(iii)** add relevant edges from IFD model to the graph. To be more specific, this graph is built based on framework-related statements identified in the program, their *data dependencies* captured from the program, plus data dependencies mined from the framework's source code. The reasoning behind removing sequence edges and keeping data edges is that if the data produced in an API call a_1 is not being used by an API call a_2, it means that calling a_1 before a_2 in the program is not required, i.e., the order of a_1 and a_2 does not affect the program's behavior. We reflect this non-restriction situation in our GRAAM by eliminating this order constraint between a_1 and a_2.

Figure 5 shows the created GRAAMs for the examples in Listing 1 and 2. These GRAAMs are created based on the primary API usage graphs depicted in Figure 2 and the IFD model shown in Figure 4. For instance, the edge between LoginContext.login() and LoginContext.getSubject() in Figure 5b comes from data dependency between LoginContext.login() and LoginContext.getSubject() captured in IFD model depicted in Figure 4. As shown in Figure 5, parts of programs in Listing 1 and Listing 2 that implement the same thing have the same sub-GRAAMs.

V. Training Phase: Inferring a Framework API Specification Model (FSpec)

ArCode uses the repository of created GRAAMs to infer the Framework API Specification Model (FSpec).

A. Framework API Specification Model (FSpec)

We use the collected sound GRAAMs to build a unified graph-based Framework API Specification Model (FSpec) which represents **correct** ways to use the framework. This model aims to (a) reflect only possible correct combinations of API calls, (b) to contain only paths that represent a correct framework's API call sequence, and (c) to create one representative for all the semantically equivalent API usages.

Figure 6 shows a Framework API Specification model (FSpec) built from the GRAAMs shown in Figure 5a and Figure 5b. Similar to a GRAAM, an FSpec encompasses three types of nodes: *start node*, *end node*, and *framework-related node*. An FSpec has the same edge type as a GRAAM, *API order constraint* edge. This type of edge represents the strict orders of framework APIs one should follow to correctly incorporate that framework in a program. However, FSpec has a new label on each edge: *frequency*. Frequency of a sub-graph of FSpec represents the number of times that the corresponding API usage was observed in the code repository of sound programs.

For instance, all edges between the start node and login() node in Figure 6 are labeled with frequency of 2. That means, there were two (semantically) equivalent API usages observed in the code repository that both **(i)** instantiated objects of Subject and CallbackHandler, **(ii)** used those objects to instantiate an object of LoginContext, and then, **(iii)** called login() method of LoginContext. After the login() node, however, the frequencies are changed to 1. It means that there was a program that has not used any other APIs of JAAS framework after login() method. In addition to that, the model shows that there was a program that continued calling framework APIs and had two more API calls.

FSpec represents *semantically equivalent* API usages in a single representation. This property shows the **generalizability** of ArCode in the sense that when it visits an instance of correct API usage in a program, a representative of all the semantically equivalent API usages to the originally visited one would be added to the FSpec under construction.

B. Inferring a FSpec Model from Sound GRAAMs

To infer a framework's API specification model (FSpec), ArCode finds mergeable parts of created *sound* GRAAMs. We will define mergeable parts of GRAAMs later in this section. Through the inference process, if ArCode finds

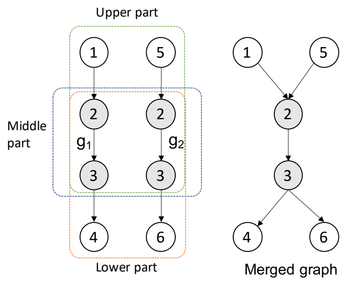


Fig. 7: Emergence of unexpected and unsound sequences by merging g_1 and g_2 from their middle parts

meargable part of a GRAAM similar to one that was previously added to the FSpec model, then, it increases the frequencies of the corresponding edges in FSpec. Otherwise, it adds the corresponding new nodes to the model and sets their edge frequency to 1.

To guarantee that all paths from start to end nodes represents a correct framework usage, we have provided inference rules to identify **Mergeable sub-GRAAMs**. Two sub-GRAAMs are mergeable if **(i)** both include the start node, and **(ii)** their corresponding sequences are *semantically equivalent*. Assuming that there are two GRAAMs g_1 and g_2, we explain how the merging algorithm works such that g_2 merges into g_1.

- We first identify all the mergeable sub-GRAAM pairs from g_1 and g_2;
- Amongst the identified merging candidates, the pair with the highest number of nodes is selected;
- The frequency of edges from $sub - g1$ increments by the frequency of corresponding edges from $sub - g2$;
- The remained parts of $g2$ which are not included in $sub-g2$ will be added to g_1;
- We repeat this process until there are no more mergeable sub-GRAAM pairs from g_1 and g_2. This process guarantees that in the end, no semantically equivalent sequences exists in the start node's children list.

To clarify this algorithm, there are three different parts in Figure 7 that could be a candidate for merging purposes. The *upper part* starts with the root (e.g. node 1 from g_1 and node 5 from g_2) of the graph and may (or may not) contain some successors of the root. The *middle part* does not contain root nor any end nodes. The *lower part* contains at least one end node and may (or may not) include predecessors of the end node. To merge two GRAAMs g_1 and g_2, the algorithm only considers their *upper parts* to avoid the emergence of incorrect paths. Figure 7 provides an example of this situation. Assume that $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ and $5 \rightarrow 2 \rightarrow 3 \rightarrow 6$ are two *correct* API sequences. Upon merging g_1 and g_2 from their middle part, two *incorrect* paths ($1 \rightarrow 2 \rightarrow 3 \rightarrow 6$ and $5 \rightarrow 2 \rightarrow 3 \rightarrow 4$) are appeared in the merged graph.

VI. Recommendation Phase

Once ArCode is trained and a FSpec model is built, it can be used to help programmers correctly implement architectural patterns and tactics through providing correct API recommendations. The recommendation system has the following steps:

1) **Process Partial Program:** it takes in a partial program written by a programmer and creates its GRAAM;
2) **Context Based Recommendation:** the recommender engine finds the most similar *semantically equivalent* API usages inside the FSpect to the given GRAAM. Then it finds changes needed to be performed on the GRAAM to make it a correct implementation of a tactic or pattern.
3) **Ranked List**: the outcome is provided in the form of ranked list of API recommendations (e.g. remove, add, replace). The rank of each recommendation in the list is determined based on the frequency of its corresponding edges in the FSpec.

ArCode can identify the next APIs required to be called in a program to make it a complete and correct implementation. It also is able to detect misuses of APIs in a program and recommend fixes for it. Some API misuses are not detectable in the compile time since it does not violate syntax of the language. However, these are serious semantic bugs which can compromise the entire objective of a tactic/pattern.

VII. Experimental Study

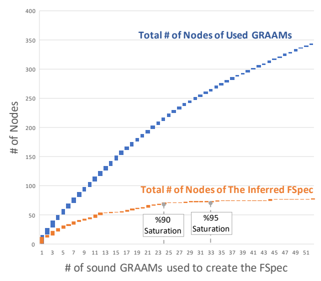
To show the practical usefulness of our approach for complex software systems, we used it in experimental studies of JAAS (Java Authentication and Authorization Services) and RMI (Remote Method Invocation) frameworks to generate recommendations for projects that incorporate them to implement tactic and patterns. JAAS addresses security architectural concerns in applications, and RMI supports interactions between different modules and methods in distributed systems. In these two studied frameworks **(i)** we investigate whether ArCode can learn correct ways framework APIs are invoked to implement tactics/patterns. We rely on a learning saturation experiment; **(ii)** we evaluate the accuracy of *API recommendation* to help programmers implement tactics and patterns, and **(iii)** we analyze ArCode's performance in *API misuse detection*.

Additionally, we compare ArCode's performance with MAPO [21] and GrouMiner [40], two approaches which were developed for API recommendation and misuse detection, but not evaluated on frameworks with architectural implications [3].

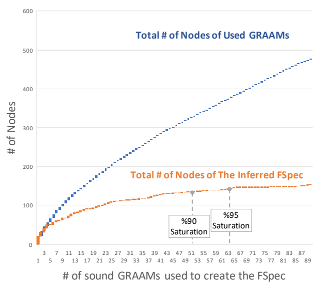
A. Data Collection

To create the training data for ArCode, we identified a large number of popular open-source projects using JAAS and RMI frameworks. These projects were collected from different public and open-source code repositories including GitHub, BitBucket and Maven. A team of three members peer reviewed programs and compiled them to generate their bytecode. The purpose of reviewing programs was to make sure that each selected program implemented authentication and authorization tactics and inter-process communication patterns and had more than one API call in its code. The final repository contains **51** projects that uses JAAS, and **50** projects that uses RMI. This repository has a total **1,106,886** lines of code (Java files), **8,831** classes and **9,600** methods.

[3]Frameworks used to implement tactics or patterns

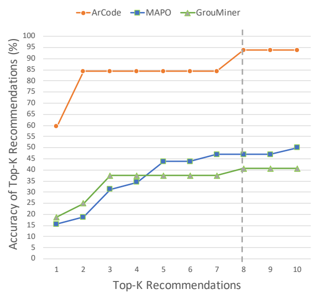


(a) ArCode learning curve while visiting 51 JAAS-based sound GRAAMs

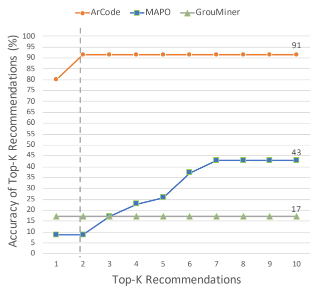


(b) ArCode learning curve while visiting 89 RMI-based sound GRAAMs

Fig. 8: Model saturation while creating FSpecs of JAAS and RMI frameworks



(a) JAAS-based projects



(b) RMI-based projects

Fig. 9: Accuracy of ArCode, GrouMiner, and MAPO for next API recommendations on JAAS and RMI experiments

B. Finding API Usages in Programs

We created GRAAMs for projects in the code repository based on the approach discussed in this paper (Section IV). We also created API usages for GrouMiner, and MAPO approaches accordingly. It is worth mentioning that a program may have more than one entrypoint [4]. Therefore, it is possible that more than one API usage can be found per program.

One issue we encountered in our experiments was failure of GrouMiner when the number of nodes in its API usage graph (i.e. graph-based object usage model) precedes 19. Since authors of GrouMiner provided their source code, and to stay faithful to the code, we did not make any changes to its source code. Therefore, we exclude API usage graphs with more than 19 nodes from our experiment.

C. ArCode *Learning Saturation*

To investigate the learning capability of ArCode, we create its ***learning curve***. This curve represents the rate of new knowledge added to FSpec comparing to the size of new GRAAMs it visits. In the case that ArCode has a good learning ability, the rate of growth of FSpec should decrease by visiting more GRAAMs.

Figure 8a and Figure 8b show learning curve of ArCode while creating FSpec for JAAS and RMI frameworks in the training phase. Blue curves represent the cumulative size of k visited GRAAMs by ArCode. Orange curves show the size of FSpec after visiting k GRAAMs. To conduct this experiment, we first sort the GRAAMs based on the number of their nodes. Then, we feed ArCode starting with bigger chunks of information. While there is an increase in the number of FSpec nodes in the initial steps, after visiting 24 GRAAMs of JAAS-based programs, we observe reaching 90% completion of the final FSpec model. It means that after this point, there would be only 10% new information learnt by FSpec. Likewise, ArCode reaches 90% completion of its FSpec after visiting 51 GRAAMs of RMI-related projects. The ArCode learning curve indicates that:

[4] A program's entrypoint is the first method invoked once it starts executing (e.g. the main() method).

Finding 1: Two case studies of RMI and JAAS indicate that there are *limited correct ways* that developers incorporated these frameworks in their program; ArCode's learning technique was able to enumerate and learn all these possible sound usages of the frameworks.

D. API Recommendation to Implement Tactics/Patterns

To evaluate the quality of recommendations, we randomly select 80% of the projects of each framework in the code repository as training and the remaining 20% as testing data set. We use the same train and test data sets for ArCode, GrouMiner, and MAPO approaches to make a fair comparison between their performance.

To generate labeled test cases for this experiment, we first create GRAAMs of each program in the test data set and then, we remove the last node of each GRAAM. Since a recommendation system is expected to recommend the removed API, that API would be used as the label of that test case. Next, we ask the recommendation system to return a ranked list of recommendations for each test case. Finally, based on the position of the correct recommendation in the ranked list, we compute the top-K accuracy of the recommendations ($k : 1 \rightarrow 10$). Since we expect only one correct answer for each test case, the results for top-K precision and top-K recall would be the same as top-K accuracy in our experiments.

We also computed the top-K accuracy of recommendations for MAPO and GrouMiner approaches. Figure 9 shows the result of this experiment. In the case of JAAS framework (Figure 9a), ArCode achieved a 59% accuracy while considering only the top ranked (top-1) API recommendation. However, if we consider top-2 recommendations, the accuracy improves to 84%. Finally, if we consider top-8 APIs (or beyond), the accuracy of recommendations provided by ArCode tops 94%. Compared to ArCode, the accuracy of MAPO and GrouMiner reaches 50% and 41% for JAAS-based programs. For RMI-based programs, ArCode provides 91% accuracy for top-2 recommendations and beyond. The highest accuracy for MAPO and GrouMiner is 43% and 17%. Based on our observation, the diversity of API usages in JAAS-based programs is fewer compared to those of RMI-based programs. As a result, all approaches have their best performance in JAAS-based repository. Nevertheless, ArCode still outperforms MAPO and GrouMiner in both JAAS- and RMI-based tests.

These results bring us to the following observation:

Finding 2: ArCode's next API recommendation outperforms the prior work significantly (40% and more). In both case studies top-2 recommendations were reliable (85% in JAAS and 95% in RMI), however ArCode top-1 recommendations in RMI were more reliable than JAAS.

E. API Misuse: Detecting a Missed API Call

One of common API missuses is missing a critical API call while implementing a tactic. Authenticating users with LoginContext.login() without first calling HttpSession.invalidate() [4] or checking the role of a user before granting access are just some of many examples.

To examine the accuracy of our approach in identifying such cases of API misuse in programs, similar to the API recommendation experiment (Section VII-D), we generate labeled test cases for this experiment. To do so, we randomly remove an API call from each program in an iterative manner. We were able to generate 77 test cases for JAAS-based and 80 test cases for RMI-based experiments. These test cases are generated only from test projects. Then, we ask the system to identify the missed API and recommend a fix for it.

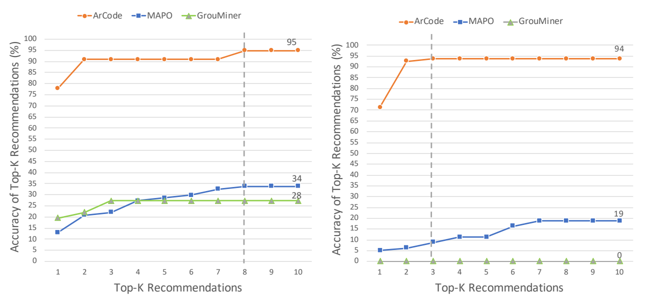
Figure 10 depicts the result of this experiment. In the case of the JAAS framework (Figure 10a), ArCode achieved a 78% accuracy considering only the top ranked (top-1) API recommendation. Top-2 recommendation shows 91% accuracy and finally, the accuracy of recommendations provided by ArCode tops 95% for top-8 recommendations and beyond. Comparing to ArCode, the accuracy of MAPO and GrouMiner reaches 34% and 28% respectively. This test scenario is more complicated compared to the previous test case (next API recommendation) because, in addition to the ancestors, the descendants of the provided recommendation and the correct answer should match as well. Therefore, the accuracy of pattern-based approaches decreases in this usecase.

Based on the results of this experiment we found:

Finding 3: ArCode can identify a missed API in implementation of tactics/patterns and provide a recommendation to fix it. ArCade's top-2 recommendation accuracy in JAAS and RMI case studies are above 90%. Furthermore, ArCode outperforms prior work with 60% and more, making it a more reliable approach for API recommendations to implement tactics and patterns.

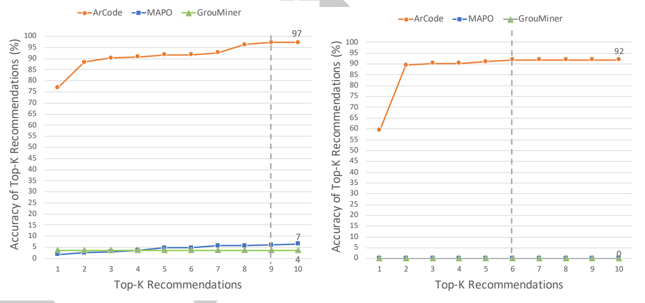
F. API Misuse: Wrong API Usage

Another type of misusing APIs in a program includes calling APIs in a wrong order in that program. Although such a program might not show a compile time error, the expected tactic is not implemented correctly. In this experiment, we create test cases that include an incorrect API usage in each project (e.g. incorrect order of APIs). Then, we ask the system to first, identify this API misuse and second, recommend a fix to make that API usage correct. To create such test cases, we go over all APIs in each program iteratively, selecting two random APIs and swapping them to generate incorrect usages of APIs. Following this approach, we were able to generate 432 test cases for JAAS and 246 test cases for RMI framework. Then, we ask the system to identify this API misuse and recommend a fix for it. Like the two other test scenarios, the system returns a ranked recommendation and we compute accuracy for top-k ($k : 1 \rightarrow 10$) recommendations.



(a) JAAS-based projects (b) RMI-based projects

Fig. 10: Accuracy of ArCode, GrouMiner, and MAPO for missed API recommendations on JAAS and RMI experiments



(a) JAAS-based projects (b) RMI-based projects

Fig. 11: Accuracy of ArCode, GrouMiner, and MAPO for fix recommendations on JAAS and RMI experiments

The result of this experiment is demonstrated in Figure 11. In the case of JAAS framework, ArCode shows 77% accuracy for top-1 recommendations. Also, if we consider 8-top returned recommendations (or beyond), the accuracy of ArCode tops 97%. The results show that MAPO and GrouMiner reach 7% and 4% accuracy respectively. This test scenario is the most complicated case compared to the previous experiments. Since we are swapping two APIs, ancestors and descendants of both the first and second APIs should be matched against those that are recommended. Thus, we observe accuracy degrading for MAPO and GrouMiner in such experiments. Based on the results of this experiment, we observed that:

Finding 4: ArCode accurately detects wrong API usages, which are more complicated compared to finding a missed API call in a tactic/pattern, and generates fix recommendations for them. It outperforms (60% to 90%) two of the prior work (GrouMiner and MAPO).

VIII. Threats to Validity

ArCode aims to filter out incorrect API usages from a code repository and then, learn correct API usages from the remained programs. While the learnt framework API specification model (FSpec) does not include incorrect API usages, we can not claim that it covers all the possible correct API usages. Another notable point is that ArCode leverages conservative merging rules while creating an FSpec. Although it can guarantee that no incorrect API usage emerges in the final FSpec, it can affect the efficiency of the training phase as well as the size of the final FSpec adversely. Moreover, the Inter-framework Dependency (IFD) model introduced in this paper is created based on static analysis over the source code of the framework to find reader-writer roles of API methods inside that framework. However, some modern frameworks use dynamic features of object oriented languages (e.g. reflections) to implement the framework. In these cases, performing dynamic analysis alongside static analysis would result in capturing more accurate dependencies between APIs.

Concerning the evaluation of ArCode provided in this paper, we used two popular Java based frameworks, JAAS and RMI. For generality purposes, conducting experiments with more frameworks would be advised.

IX. Related Work

There have been numerous works on identifying frameworks' API specifications and creating models that enable API recommendation and misuse detection systems. The aim of these systems is to help programmers correctly use API calls and prevent incorrect usage of APIs in their programs [41], [42]. Many researchers define patterns of **API co-occurrences** in the same scope of a program as an API specification. Works in this category find the frequency of co-appearance of APIs in the same context and use it to create API co-occurrence patterns. While most of the approaches in this category find such patterns within a method [26], [27], [29], [43]–[49], some go beyond and find patterns in a bigger scope (e.g. class or program) [18], [50]. We have found these works useful for simple API recommendation or misuse detection tasks. However, they become impractical when the problem gets more complex as to accurately **(i)** identify an incorrect API *order* in a code, or **(ii)** recommend the *next* API based on the order of other APIs in a *sequence*.

To address the aforementioned issues, more sophisticated approaches find **API sequence** patterns in a code repository. This is an important specification needed for a correct API recommendation or misuse detection. The developed techniques in this category range from finding partial API usage patterns and creating rule-based specifications [13], [31], [51] to mining a complete sequence of API usage patterns [14], [16], [17], [21]–[24], [52]–[54]. As an example, MAPO [21], one of the most respected API specification miners, aims to create patterns of API usage sequences and leverage that to find relevant code samples for programmers. Although sequential API specification miners take API order into account, they are not able to: **(i)** provide more API-related information in a program (e.g. data dependency) than the order of appearance in the code, and **(ii)** distinguish between two semantically equivalent sequences (Section IV).

Graph-based API pattern finders are the most advanced methods developed to cover sequence-based API pattern miners shortcomings and create more detailed API specifications (e.g. data dependency between APIs). Specifically, these works aim to track the usage of APIs related to a single object [15], [40], [55]–[57] or multiple objects [19], [25], [58]–[60] in a specific scope in programs. For instance, GrouMiner [40], a well-appreciated graph-based API pattern miner, finds relationships between APIs of the same type in a method. While the mentioned works present more detailed API specifications, they still: **(i)** do not track dependencies across different scopes (inter-procedural), **(ii)** can not represent the whole context of an API usage, and **(iii)** can not identify semantically equivalent, yet in different order, API usages in programs.

ArCode performs an inter-procedural program analysis and infers a context-sensitive graph-based API specification for frameworks. This approach produces isomorphically the same graphs for API usages that have different sequences but are semantically equivalent.

X. Conclusion & Future Work

Obtaining API specification of a framework can enable the correct implementation of architectural tactics and patterns. This paper introduces ArCode, a novel approach for inferring frameworks' API specification from limited projects. It relies on a Graph-based Framework API Usage model (GRAAM), which is an inter-procedural context-, and flow-sensitive representation of APIs in a program. Furthermore, ArCode also extracts inter-framework dependencies between APIs from framework source code and uses them to identify incorrect API usages in programs. It uses an inference algorithm to combine all extracted GRAAMs into a framework specification model. In a series of experiments, we demonstrated that it is possible to infer a framework specification model that accurately captures correct API usage to implement tactics and patterns. Moreover, recommendation systems empowered by the created framework specification model are able to provide accurate API recommendations, identify API misuses and provide a fix recommendation. Future work includes exploration of additional frameworks with regard to our technique and leveraging dynamic analysis alongside static analysis to extracting inter-framework dependencies.

Acknowledgments

This work was partially funded by the US National Science Foundation (NSF) under grant number CCF-1943300, CNS-1816845 and CNS-1823246.

References

[1] H. Cervantes, P. V. Elizondo, and R. Kazman, "A principled way to use frameworks in architecture design," *IEEE Software*, vol. 30, no. 2, pp. 46–53, 2013. [Online]. Available: https://doi.org/10.1109/MS.2012.175

[2] H. Cervantes, R. Kazman, J. Ryoo, J. Cho, G. Cho, H. Kim, and J. Kang, "Data-driven selection of security application frameworks during architectural design," in *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 2019.

[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[4] J. C. S. Santos, A. Peruma, M. Mirakhorli, M. Galstery, J. V. Vidal, and A. Sejfia, "Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of Chromium, PHP and Thunderbird," in *2017 IEEE International Conference on Software Architecture (ICSA)*, April 2017, pp. 69–78.

[5] I. J. Mujhid, J. C. S. Santos, R. Gopalakrishnan, and M. Mirakhorli, "A search engine for finding and reusing architecturally significant code," *Journal of Systems and Software*, vol. 130, pp. 81 – 93, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216302345

[6] I. Rehman, M. Mirakhorli, M. Nagappan, A. A. Uulu, and M. Thornton, "Roles and impacts of hands-on software architects in five industrial case studies," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 117–127.

[7] M. P. Robillard, "What makes APIs hard to learn? answers from developers," *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[8] R. E. Johnson, "Documenting frameworks using patterns," in *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '92. New York, NY, USA: ACM, 1992, pp. 63–76. [Online]. Available: http://doi.acm.org/10.1145/141936.141943

[9] M. Soliman, M. Galster, and M. Riebisch, "Developing an ontology for architecture knowledge from developer communities," in *2017 IEEE International Conference on Software Architecture (ICSA)*, 2017, pp. 89–92.

[10] M. Soliman, M. Galster, A. R. Salama, and M. Riebisch, "Architectural knowledge for technology decisions in developer communities: An exploratory study with StackOverflow," in *13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016, Venice, Italy, April 5-8, 2016*. IEEE Computer Society, 2016, pp. 128–133. [Online]. Available: https://doi.org/10.1109/WICSA.2016.13

[11] S. Baltes and S. Diehl, "Usage and attribution of Stack Overflow code snippets in GitHub projects," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1259–1295, 2019.

[12] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? the impact of copy&paste on android application security," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 121–136.

[13] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: from usage scenarios to specifications," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 25–34.

[14] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 651–666, 2008.

[15] S. Mover, S. Sankaranarayanan, R. B. P. Olsen, and B.-Y. E. Chang, "Mining framework usage graphs from app corpora," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 277–289.

[16] J. Fowkes and C. Sutton, "Parameter-free probabilistic API mining across GitHub," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 254–265.

[17] Y. Dang, J. Wang, K. Chen, H. Zhang, T. Xie, and D. Zhang, "Api usage pattern mining," Aug. 11 2015, uS Patent 9,104,525.

[18] Y. Lamba, M. Khattar, and A. Sureka, "Pravaaha: Mining android applications for discovering API call usage patterns and trends," in *Proceedings of the 8th India Software Engineering Conference*. ACM, 2015, pp. 10–19.

[19] S. Amann, H. A. Nguyen, S. Nadi, T. N. , Nguyen, and M. Mezini, "Investigating next steps in static API-misuse detection," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, p. (To appear).

[20] H. Zhong and H. Mei, "An empirical study on API usages," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[21] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending API usage patterns," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 318–343.

[22] S.-K. Hsu and S.-J. Lin, "Macs: Mining API code snippets for code reuse," *Expert Systems with Applications*, vol. 38, no. 6, pp. 7291–7301, 2011.

[23] M. A. Saied, H. Sahraoui, E. Batot, M. Famelis, and P.-O. Talbot, "Towards the automated recovery of complex temporal API usage patterns," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 1435–1442.

[24] M. A. Saied, E. Raelijohn, E. Batot, M. Famelis, and H. Sahraoui, "Towards assisting developers in API usage by automated recovery of complex temporal patterns," *Information and Software Technology*, vol. 119, p. 106213, 2020.

[25] X. Gu, H. Zhang, and S. Kim, "Codekernel: A graph kernel based approach to the selection of API usage examples," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 590–601.

[26] F. Thung, "Api recommendation system for software development," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 896–899.

[27] W. Zheng, Q. Zhang, and M. Lyu, "Cross-library API recommendation using web search engines," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 480–483.

[28] X. Ren, J. Sun, Z. Xing, X. Xia, and J. Sun, "Demystify official API usage directives with crowd sourced API misuse scenarios, erroneous code examples and patches," pp. 1407–1410, 2020.

[29] M. Monperrus and M. Mezini, "Detecting missing method calls as violations of the majority rule," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, p. 7, 2013.

[30] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static API-misuse detectors," *IEEE Transactions on Software Engineering*, 2018.

[31] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 35–44.

[32] M. Mirakhorli, H. Chen, and R. Kazman, "Mining big data for detecting, extracting and recommending architectural design concepts," in *1st IEEE/ACM International Workshop on Big Data Software Engineering, BIGDSE 2015, Florence, Italy, May 23, 2015*, L. Baresi, T. Menzies, A. Metzger, and T. Zimmermann, Eds. IEEE Computer Society, 2015, pp. 15–18. [Online]. Available: https://doi.org/10.1109/BIGDSE.2015.11

[33] M. Bhat, C. Tinnes, K. Shumaiev, A. Biesdorf, U. Hohenstein, and F. Matthes, "Adex: A tool for automatic curation of design decision knowledge for architectural decision recommendations," in *IEEE International Conference on Software Architecture Companion, ICSA Companion 2019, Hamburg, Germany, March 25-26, 2019*. IEEE, 2019, pp. 158–161. [Online]. Available: https://doi.org/10.1109/ICSA-C.2019.00035

[34] A. Shokri, J. C. S. Santos, and M. Mirakhorli, "Arcode-1.2.1," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4581117

[35] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1997, pp. 108–124.

[36] O. Shivers, "Control-flow analysis of higher-order languages," Ph.D. dissertation, PhD thesis, Carnegie Mellon University, 1991.

[37] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, 1990.

[38] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[39] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

[40] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 383–392.

[41] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Trans. Software Eng.*, vol. 39, no. 5, pp. 613–637, 2013.

[42] H. Zhong and H. Mei, "An empirical study on API usages," *IEEE Transactions on Software Engineering*, 2017.

[43] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 306–315.

[44] E. Moritz, M. Linares-Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers, "Export: Detecting and visualizing API usages in large source code repositories," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 646–651.

[45] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui, "Mining multi-level API usage patterns," in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 2015, pp. 23–32.

[46] M. A. Saied, H. Abdeen, O. Benomar, and H. Sahraoui, "Could we infer unordered API usage patterns only using the library source code?" in *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 2015, pp. 71–81.

[47] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "Api code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 511–522.

[48] R. Xie, X. Kong, L. Wang, Y. Zhou, and B. Li, "Hirec: API recommendation using hierarchical context," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 369–379.

[49] H. Niu, I. Keivanloo, and Y. Zou, "Api usage pattern recommendation for software development," *Journal of Systems and Software*, vol. 129, pp. 127–139, 2017.

[50] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "Focus: A recommender system for mining API function calls and usage patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1050–1060.

[51] C. Liu, E. Ye, and D. J. Richardson, "Ltrules: an automated software library usage rule extraction tool," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 823–826.

[52] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical API usage," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 826–836.

[53] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 319–328.

[54] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "CSCC: Simple, efficient, context sensitive code completion," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 71–80.

[55] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 302–321, 2010.

[56] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Grapacc: a graph-based pattern-oriented, context-sensitive code completion tool," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1407–1410.

[57] M. Ghafari and A. Heydarnoori, "Towards a visualized code recommendation for APIs enriched with specification mining," in *Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering*, 2014, pp. 26–27.

[58] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Recommending framework extension examples," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 456–466.

[59] ——, "Femir: A tool for recommending framework extension examples," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 967–972.

[60] Y. Pacheco, J. De Bleser, T. Molderez, D. Di Nucci, W. De Meuter, and C. De Roover, "Mining Scala framework extensions for recommendation patterns," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 514–523.