

IPSYNTH: Interprocedural Program Synthesis for Software Security Implementation

Ali Shokri, Ibrahim Jameel Mujhid, and Mehdi Mirakhori

Abstract—To implement important quality attributes of software such as architectural security tactics, developers incorporate API of software frameworks, as building blocks, to avoid re-inventing the wheel and improve their productivity. However, this is a challenging and error-prone task, especially for novice programmers. Despite the advances in the field of API-based program synthesis, the state-of-the-art suffers from a twofold shortcoming when it comes to architectural tactic implementation tasks. First, the specification of the desired tactic must be explicitly expressed, which is out of the knowledge of such programmers. Second, these approaches synthesize a block of code and leave the task of breaking it down into smaller pieces, adding each piece to the proper location in the code, and establishing correct dependencies between each piece and its surrounding environment as well as the other pieces, to the programmer.

To mitigate these challenges, we introduce IPSynth, a novel inter-procedural program synthesis approach that automatically learns the specification of the tactic, synthesizes the tactic as inter-related code snippets, and adds them to an existing code base. We extend our first-place award-winning extended abstract recognized at the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21) research competition track. In this paper, we provide the details of the approach, present the results of the experimental evaluation of IPSynth, and analyses and insights for a more comprehensive exploration of the research topic. Moreover, we compare the results of our approach to one of the most powerful code generator tools, ChatGPT. Our results show that our approach can accurately locate corresponding spots in the program, synthesize needed code snippets, add them to the program, and outperform ChatGPT in inter-procedural tactic synthesis tasks.

Index Terms—Program Synthesis, Architectural Tactic, Framework Specification Model, API Usage Model.

1 INTRODUCTION

Software architectural tactics are re-usable design strategies that aim to preserve important quality attributes of a software system, including security, interoperability, availability, performance, and scalability [1]. To satisfy these concerns, software architects carefully compare and choose proper architectural tactics to be implemented by software developers in the code [2], [3]. In the subsequent development phase, these architectural choices must be implemented completely and correctly in order to avoid any drifts from the envisioned design [4].

Previous studies [1], [5] emphasize the significance of correct tactic implementation. In practice, developers—especially less experienced ones—face challenges in correctly implementing chosen tactics [6], resulting in software bugs and flaws [7]. Insufficient experience, limited knowledge of architectural tactics, and unfamiliarity with the underlying framework and the code base are primary factors contributing to an incorrect implementation of these tactics [8]. In practice, these tactics are mostly implemented by incorporating APIs of third-party libraries [9]. For example, Java Authentication and Authorization Security Services (JAAS) [10] is a popular Java-based framework that provides APIs for adding such tactics to a program.

Program synthesis has been promising in providing support for programmers by automatically constructing a program in an underlying programming language, based

on a given specification of the to-be-synthesized program [11]. In the past recent years, there have been works introduced by researchers that specifically focus on synthesizing a program from a given set of API calls [12], [13], [14], [15], [16], [17]. However, despite the advances in this field, the current state-of-the-art API-based program synthesis approaches suffer from a two-fold shortcoming when it comes to architectural tactic implementation tasks. First, the specification of the desired tactic has to be explicitly expressed by the user, which comes with complexity [6] and is out of the knowledge of non-expert programmers [4]. Second, these approaches synthesize a block of code and leave the task of breaking down the synthesized code into smaller pieces, adding each piece to its proper location in the code, and establishing correct dependencies between each piece and its surrounding environment as well as the other pieces, to the programmer. This is obviously a non-trivial task, even for more savvy programmers.

To mitigate these challenges, in this paper, we introduce IPSYNTH, a novel inter-procedural program synthesis approach that automatically learns the specification of the to-be-synthesized tactic, synthesizes the tactic as inter-related code snippets, adds each piece to its corresponding location in the given code base, and establishes correct dependencies (i.e., control- and data-dependencies) between each piece and its surrounding environment. We particularly enhance and extend our prior extended abstract, titled “A program synthesis approach for adding architectural tactics to an existing code base” [18] which was awarded the first-place prize in the research competition track at ASE'21. Building on the positive feedback and encouragement received, this

• All authors are with the Department of Software Engineering, Rochester Institute of Technology, Rochester, NY, USA. (email: as8308@rit.edu; ijmvse@rit.edu; mxmvse@rit.edu)

journal paper provides a more in-depth and rigorous investigation of the research subject.

The purpose of IPSYNTH is: *given a compilable program, i.e., a syntactically correct program, it is able to automatically synthesize and add architectural tactics to that program such that the final program is syntactically and semantically (w.r.t. tactic implementation) correct.*

1.1 Contributions

In summary, the contributions of this paper are as follows.

- A novel inter-procedural program synthesis approach called IPSYNTH for adding architectural tactics to a program. This approach follows API-based program synthesis in which the specification is automatically inferred from a pre-learned API usage specification model as well as the context of the under-development program.
- A dataset of test programs to be used by other researchers for inter-procedural program synthesis tasks. The dataset consists of 20 architectural tactic implementation tasks and is publically available from its anonymized repository at <https://anonymous.4open.science/r/Anonymous-82DE>.
- An experimental study of the approach to investigate its accuracy and effectiveness. This study has been conducted in two different levels of granularity, (i) component level where we assess the functionality of each of the components of the approach separately, and (ii) the entire approach, where we evaluate the accuracy of the synthesized code and compare it against the output of one of the most powerful tools that is widely used by programmers, ChatGPT [19]. The experiment and the results are provided in Section 4.

The rest of the paper is structured as follows. Section 2 motivates our work through a tactic synthesis example. The approach is detailed in Section 3. We provide an experimental study of the approach in Section 4. Section 5 discusses the limitation of the approach sheds light on the future direction of the work. Related works are studied in Section 6. Finally, Section 7 concludes the paper.

2 MOTIVATING EXAMPLE

In order to better motivate our problem, in this section, we provide an example of an architectural tactic synthesis task. We then, use this example throughout the paper to provide a better walk-through of the technical parts of the approach. Figure 1 shows an under-development program in which the programmer aims to use *Java Authentication and Authorization Services (JAAS)*, a popular open-source java-based framework, to implement *authentication* security tactic and restrict non-user access to unwanted resources. This would not be a trivial task, especially for novice programmers. The entry point of this program is the `main` method (line 16), which retrieves the name of the authentication module from the `args` argument (line 17) and creates an instance of `JaasImplementor` class (line 18). Then, in a sequence of method calls, it is supposed to instantiate needed objects for the authentication process (line 19), perform the login process (line 20), and finally, inspect

```

1. public class JaasImplementor {
2.     String varStr;
3.     public LoginContext initializeLC(String name) throws LoginException(){
4.         //TODO: To be implemented
5.         return null;
6.     }
7.
8.     public void login(LoginContext loginContext) throws LoginException(){
9.         //TODO: To be implemented
10.    }
11.
12.    public void inspectSubject( LoginContext loginContext ){
13.        //TODO: To be implemented
14.    }
15.
16.    public static void main(String[] args) throws LoginException{
17.        String moduleName = args[0];
18.        JaasImplementor jaasImplementor = new JaasImplementor();
19.        LoginContext loginContext = jaasImplementor.initializeLC( moduleName );
20.        jaasImplementor.login(loginContext );
21.        jaasImplementor.inspectSubject( loginContext );
22.    }
23.
24. }
```

Fig. 1. Before synthesizing the *authentication* tactic

```

1. public class JaasImplementor {
2.     String varStr;
3.     public LoginContext initializeLC(String name) throws LoginException(){
4.         CallbackHandler callbackHandler = new MyCallbackHandler();
5.         LoginContext loginContext = new LoginContext( name, callbackHandler );
6.         return loginContext;
7.     }
8.
9.     public void login(LoginContext loginContext) throws LoginException(){
10.        loginContext.login();
11.    }
12.
13.    public void inspectSubject( LoginContext loginContext ){
14.        Subject subject = loginContext.getSubject();
15.        List principals = subject.getPrincipals() ;
16.    }
17.
18.    public static void main(String[] args) throws LoginException{
19.        String moduleName = args[0];
20.        JaasImplementor jaasImplementor = new JaasImplementor();
21.        LoginContext loginContext = jaasImplementor.initializeLC( moduleName );
22.        jaasImplementor.login(loginContext );
23.        jaasImplementor.inspectSubject( loginContext );
24.    }
25.
26. }
```

Fig. 2. After synthesizing the *authentication* tactic

the authenticated subject (line 21). This is a routine process one needs to follow for an authentication task. While these steps seem limited, the challenge comes from (i) creating pieces of code snippets constructed from correct APIs to be used in different parts of the program, (ii) identifying candidate locations in the program in which each piece can be added to, and (iii) correctly connecting each piece to its surrounding context, as well as to the other pieces. Figure 2 shows a correct implementation of authentication tactic using the JAAS framework for the program shown in Figure 1. To start the authentication process, an object of `LoginContext` should be instantiated (line 5). `LoginContext` is the keystone part of an authentication process implemented by the JAAS framework. However, the constructor of `LoginContext` requires an object of `CallbackHandler` as an input argument. Hence, an object of `CallbackHandler` is created

at line 4. A `CallbackHandler` handles the communication with the user, including collecting the user's `username` and `password`. Once the `LoginContext` is created and returned (line 6), it can be used to perform the actual logging-in process. This is implemented by calling the `login()` method of the previously created `LoginContext` in a separate method (line 10). Finally, to inspect the output of the authentication process, this program retrieves the `subject` object (line 14) that is populated during the login process. A `Subject` encompasses all the information around an authenticated user. In case of successful authentication, the retrieved `subject` has the user's `principals` (line 15), which can be corresponding `roles` of the user in the specified context. Otherwise, the returned set of `principals` would be empty. There are some more details around the JAAS framework which we have overlooked in this example for the sake of simplicity.

As the example demonstrates, the task of program synthesis in architectural tactic implementation is an inter-procedural task. It means that APIs might be used in different methods of different classes in a program, yet they need to interact with each other through method calls. Moreover, to guarantee the semantic correctness of the synthesized program, the synthesizer needs to have a good knowledge of correct control- and data dependencies between the used APIs in the program for implementing a specific tactic. Also, it is crucial to analyze the under-development program and spot the correct locations in the code to which each piece of the tactic should be added. Finally, for each spot, the corresponding code snippet should be synthesized such that the correct data- and control dependencies between that code snippet and its surrounding environment, as well as the other synthesized code snippets, are established.

The state-of-the-art approaches are not designed and equipped for such a task. The approach introduced in this paper aims to address the mentioned needs. Figure 2 represents the outcome of our approach for synthesizing the authentication tactic for the program shown in Figure 1.

3 INTER-PROCEDURAL PROGRAM SYNTHESIS

To address the gaps mentioned in the current state-of-the-art approaches for synthesizing architectural tactics in a program, we introduce our **INTER-PROCEDURAL PROGRAM SYNTHESIS (IPSYNTH)** approach. Figure 3 provides an overview of IPSYNTH which consists of four main steps. Before providing a birds-eye view of the steps, we briefly provide background about our prior work, ARCODE [8], [20], that produces specification models of APIs that programmers use in their tactic implementation tasks. We use this model during the first step of IPSYNTH (Section 3.1).

ARCODE is a learning-by-example approach designed for learning API specifications related to architectural tactic implementation tasks. ArCode is able to generate API specification models (*FSpec*) for the frameworks that developers incorporate to implement tactics (e.g., Java Authentication and Authorization Services - JAAS). *FSpec* is a directed graph that the nodes are API calls and the edges are dependencies between API calls (e.g., control- and data-dependencies) that one should consider in order to *correctly* implement a tactic using those API calls. An *FSpec* represents the correct ways of using API calls in order to *correctly*

implement a tactic. Figure 4 demonstrates an example of an *FSpec* generated by ARCODE that represents two ways to correctly use API calls provided by the JAAS framework for implementing the authentication tactic. Each correct API usage starts with the *start* node and ends in an *end* node. The numbers on the edges of this model represent the frequency of times that ARCODE observed the same tactic in different programs in its training set.

In the first step of IPSYNTH (Section 3.1), we perform clustering on API calls inside the *FSpec* model such that each cluster corresponds to a tactic-related sub-task —conceptual sub-tasks in the realization of a tactic. For example, in the case of the *authentication* tactic implementation by incorporating the JAAS framework, there will be API clusters related to *object instantiation*, *logging in*, and *subject inspection*. We also annotate the clusters with meaningful labels, indicating which part of the *FSpec* model implements what sub-task of the tactic implementation. Next, we find a mapping between the annotated clusters in the *FSpec* and the correct locations in the code that tactic pieces should be synthesized and added to (Section 3.2). In the third step (Section 3.3), sketches of to-be-synthesized pieces of the tactic will be generated. A *sketch* is a skeleton of a code snippet that consists of some unimplemented parts, i.e. *holes*. Lastly, by resolving the holes in sketches (Section 3.4), we generate an actual tactic implementation from the previously created code structure. This approach follows the concept of *correct by construction*, meaning that we make sure that the synthesis process does not generate an incorrect (semantically and syntactically) tactic. The details of the mentioned steps are provided as follows.

3.1 Step #1 - FSpec Annotation

Since we follow the API-based program synthesis approach, we need to learn how to correctly use the API of a framework for correctly implementing a tactic in a program. In that regard, we leverage ARCODE [8], [20], our prior work which creates the *framework API specification model (FSpec)*, a probabilistic model that represents only the correct ways of incorporating API of that framework in a program to implement a tactic. To better guide our program synthesizer, we cluster API calls inside the *FSpec* such that all APIs inside a cluster contribute to the implementation of a (sub)task. Then, we **annotate** (i.e., label) each cluster with a meaningful name that represents the task that APIs inside the cluster contribute to. An annotation can be a simple or a compound name. For instance, in the code snippet provided in Figure 2, creating instances of `CallbackHandler` and `LoginContext` (line 4-5) would be considered as the task of *initializing* objects needed for the authentication process. Therefore, one would expect to see these two API calls in the same cluster inside the *FSpec* with an annotation such as `#Initialization`. In general, it would be quite possible for API calls of different clusters to collaborate and contribute to more coarse-grained tasks. Therefore, we will end up in a hierarchical clustering/annotation structure. For example, one would expect to see a couple of API calls in a cluster annotated as `#Authentication`, while there are more fine-grained clusters annotated as `#Initialization`, `#Logging_In`, and `#Subject_Inspection` inside the `#Authentication`. Ideally,

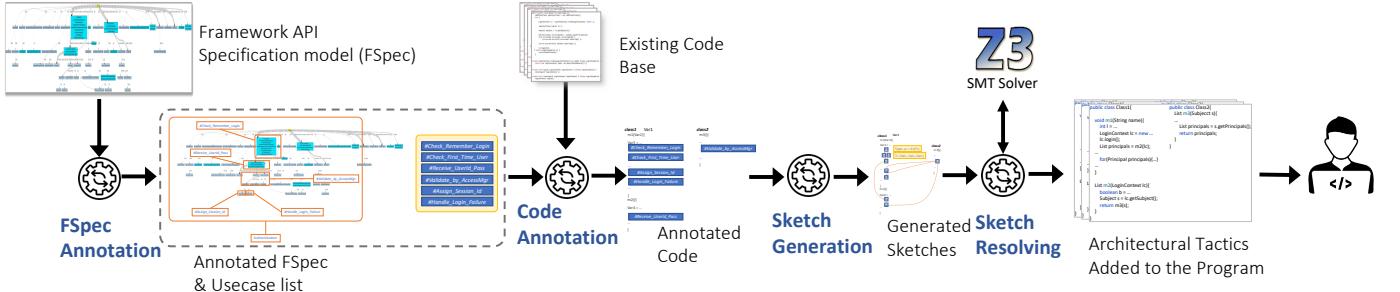


Fig. 3. An overview of our IPSYNTH approach

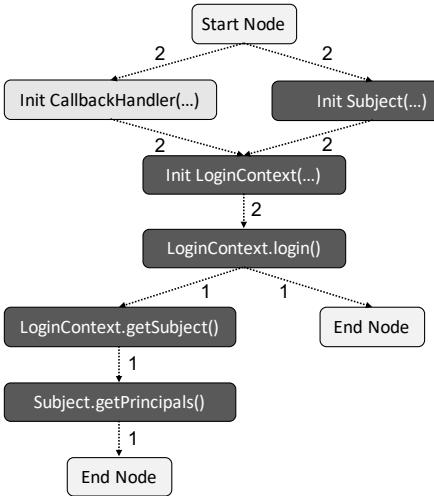


Fig. 4. An FSpec constructed by ARCODE that demonstrates two correct ways of incorporating JAAS API for implementing the authentication tactic.

generated code for each fine-grained cluster will be put in a separate method, yet, methods will be collaborating to implement the entire tactic.

To automatically generate clusters and their corresponding annotations, we follow an iterative process that starts with each API call inside the FSpec. The process is explained as follows.

- 1) We extend the ARCODE approach such that during the FSpec creation, we generate an annotation for each API call inside the FSpec. The annotation of an API call would represent the finest-grained (sub)task that the API call contributes to. For this purpose, let's say that API A_1 in the FSpec was found from method m of class c of program p during the FSpec creating process. We aim to generate a short summary of the task that method m contributes to and annotate API A_1 inside the FSpec with that summary.

To achieve this goal, we leverage state-of-the-art *method naming suggestion* approaches that find meaningful names for a given method. In particular, we use Code2Vec [21] which is an approach that given an implementation of a method (i.e., the body of the method), finds semantically similar methods in its code repository and returns a ranked list of suggested methods names for the given method implementation. In other words, based on a given method implementation, it returns a ranked list of meaningful name suggestions that

aim to describe the semantics of the given implementation. We use this technique to annotate each API call inside the FSpec model. For example, if ARCODE finds **CallbackHandler** initialization in method m of class c of program p in a code repository that is used for creating the FSpec, we pass the body of method m to Code2Vec and receive a ranked list of corresponding names. Then, we use the 1st ranked name as the annotation for **CallbackHandler** initialization API inside the FSpec. Please note that since an API could be found from multiple methods, classes, or programs, there might be different annotation candidates per API call inside the FSpec. In such a situation, we choose the annotation with the highest frequency as the annotation of the API.

- 2) Next, using Levenshtein distance metric [22], we calculate dissimilarities between the selected annotations for APIs (Equation (1)) and create the first level of clusters.

$$lev(a, b) = \begin{cases} |a| & |b| == 0 \\ |b| & |a| == 0 \\ lev(tail(a), tail(b)) & a[0] == b[0] \\ 1 + \min \begin{cases} lev(tail(a), b) \\ lev(a, tail(b)) \\ lev(tail(a), tail(b)) \end{cases} & Otherwise \end{cases} \quad (1)$$

In particular, for each branch (i.e., API usage) in FSpec, we first create a list of pairs of APIs, compute the Levenshtein distance score for each pair, and sort the list ascending based on the computed scores. Then, starting from the first element of the list, we create a cluster per pair. Please note that to avoid adding an API in more than one cluster, whenever we select a pair to create a cluster for its corresponding APIs, we remove all the remaining pairs in the list that contain either of the APIs of the selected pair.

- 3) Finally, we iteratively create hierarchical clusters for APIs in each branch of FSpec based on the distances of clusters' annotations. While forming a cluster, the annotation with the minimum distance to all other annotations in that cluster will be selected as the annotation for the newly created cluster. Due to its hierarchical nature, one would reasonably expect to see the top-most level cluster be annotated as **#Authentication**.

3.2 Step #2 - Code Annotation

Next, we leverage the under-development program as a context, to identify candidate clusters of API calls inside the FSpec that need to be incorporated into the synthesis

task. To that extent, we find mappings between API clusters inside the FSpec and candidate locations in the program such that a code snippet synthesized from the cluster can be added to the location. Each mapping basically represents *what* part of the tactic should be synthesized, and *where* in the program it should be placed. We consider four criteria for creating these mappings, namely, **(i)** method name similarity, **(ii)** required variable availability, **(iii)** correct data-and control-dependencies, and **(iv)** the quality of the final code. The details of each criterion will be provided later in this section. For each criterion, we compute a score to quantify the likelihood of adding a cluster to that location in the program. We use this score to create a ranked list of candidate pairs of $\langle \text{annotation}, \text{location} \rangle$ while synthesizing the tactic.

3.2.1 Criteria 1: Method Name Similarity

The given program might have many methods to which the to-be-synthesized code snippets can be added. Each to-be-synthesized code snippet will be created from API calls inside a cluster. During the FSpec annotation phase (Section 3.1), we generate annotations for clusters such that each annotation describes the (sub)task that API calls inside the cluster contribute to. Hence, one heuristic for identifying the proper method of the program for adding a to-be-synthesized code snippet would be to consider methods with a similar name to the cluster’s annotation. For the purpose of method name similarity computation, we use Equation (2) which basically is the inverse of the Levenshtein distance metric that we used before (Equation (1)). Please note that in order to avoid division by zero in case of complete similarity (i.e., $distance = 0$), we added a constant (i.e., 1) to the denominator of Equation (2). A *method name similarity score* (MNS) is a double number in the range of $(0, 1]$. The higher the MNS score is, the more desirable the method is for adding the to-be-synthesized code snippet to it.

$$MNS(a, b) = \frac{1}{Lev(a, b) + 1} \quad (2)$$

For instance, in the created FSpec, **CallbackHandler** instantiation and **LoginContext** instantiation are annotated as **#Initialization**. As shown in the under-development program in Figure 1, there is a method named **initializeLC(...)**. The name of this method has the most similarity to **#Initialization** annotation. The reason is that the MNS score between the “Initialization” annotation and the method names “initializeLC”, “login”, “inspectSubject”, and “main” is 0.14, 0.09, 0.07, and 0.09, respectively. Therefore, **initializeLC(...)** would be at the top of the list of candidate methods for adding the synthesized code snippet from API calls of the **#Initialization** cluster.

3.2.2 Criteria 2: Available Variables

Each FSpec cluster consists of multiple API calls and their inter-dependencies. If an API call A_1 produces data that is used by another API call A_2 (i.e., $A_1 \xrightarrow{\text{data}} A_2$), it means that A_2 is data-dependent to A_1 . This dependency could be either as a target object o_1 (e.g., $o_1 = A_1(\dots); o_1.A_2(\dots);$), or as an input argument arg_1 (e.g., $arg_1 = A_1(\dots); A_2(arg_1, arg_2, \dots)$). However, there is a

possibility that A_2 takes another input argument arg_2 of type t_1 that would not be generated by A_1 or any other API calls inside the cluster. Therefore, at least one pre-declared, visible, and initialized variable from type t_1 should be *available* in the scope to which the synthesized code snippet from that cluster will be added. This will add another constraint for finding a proper location in the program to synthesize and add each part of the tactic. For instance, annotation **#Initialization** consists of **CallbackHandler** and **LoginContext** instantiation. While **CallbackHandler** does not need any specific input arguments, **LoginContext** requires an object of type **CallbackHandler** and an object of type **String**. The instantiated **CallbackHandler** would satisfy the need for an argument of type **CallbackHandler**. However, still, a variable of type **String** should be introduced to the constructor of **LoginContext**. Based on this constraint, in the case of the tactic synthesis task demonstrated in Figure 1, there are two possible locations in the program to add API calls related to **#Initialization**: **(1)** inside the method **initializeLC(...)**, and **(2)** inside the method **main(...)** after line 17. Please note that although there is a variable named **varStr** of type **String** which is declared at line 2, our approach does not consider this variable since it is not initialized anywhere in the program.

We use **JavaParser** [23], which is a popular open-source Java parser tool, for finding visible variables from a queried location. Moreover, to ensure that each of the found variables is initialized before reaching that location, we use the **WALA** [24] program analysis tool to create a context-sensitive data flow for the program. We use Equation (3) to calculate *variable availability score* (VAS), which quantifies the suitability of a location L in the program for adding a to-be-synthesized code snippet related to cluster C , from a variable availability point of view.

$$VAS(C, L) = \begin{cases} 1 & M = 0 \\ \prod_{i:1 \rightarrow M} \min\left(\frac{Avail(L, T_i)}{Args(C, T_i)}, 1\right) & M > 0 \end{cases} \quad (3)$$

In this equation, C is the cluster, L is the location in the code, and M is the number of different types T that API calls in the cluster C need to receive as unfilled arguments or target objects. If there are no unfilled arguments or target objects in the cluster (i.e., $M = 0$), then, the VAS score will be 1 since the to-be-synthesized code from the cluster C can be added to the location L without a problem (w.r.t., variable availability). Otherwise (i.e., $M > 0$) we compute the VAS as explained. For example, in the case of the cluster **#Initialization**, since the only unfilled argument is of type **String**, then, M would be 1. However, if the argument of type **CallbackHandler** was not filled by another API call, M should be considered 2. Moreover, T_i represents the i -th type in the cluster, $Avail(L, T_i)$ counts the number of available variables of type T_i at location L in the program, and $Args(C, T_i)$ counts the number of unfilled arguments or target objects in the to-be-synthesized code from cluster C that are of type T_i . In case there is at least one distinct available variable of type T_i per needed argument of type T_i , the score related to that type will be 1. Otherwise, it will be a fraction number in the range of $[0, 1]$. If there is no available variable of type T_i , then it means that the location L is not suitable for adding the to-be-synthesized

snippet from the cluster C . The reason is that we can not fill the unfilled arguments of type T_i with a proper variable in this location of the program. The multiplication in the equation guarantees that in such cases the overall score will be 0. In general, the higher the MNS score is, the more appropriate the location is for adding the to-be-synthesized code snippet.

3.2.3 Criteria 3: Control- and Data-dependencies

From the previous two criteria, we were able to identify and rank candidate locations in the program for synthesizing code snippets related to a cluster. Those two criteria focus on API calls inside each cluster. However, in addition to control- and data-flow dependencies between API calls inside a cluster (i.e., intra-cluster dependencies), there are some dependencies between API calls of different clusters (i.e., inter-cluster dependencies) that need to be considered while searching for proper locations in a program. In other words, the synthesizer should also preserve the correct control- and data-flow dependencies between API elements of different clusters during the synthesis process. For instance, let's assume that the synthesizer adds the API `Login()` of class `LoginContext` which belongs to cluster C_2 to a location L_2 in the program. In such a case, the synthesizer has to add the *initialization* of `LoginContext` which belongs to cluster C_1 to a location L_1 such that L_1 is visited before L_2 while running the program.

In order to check the above condition, we first perform an inter-procedural context-sensitive static analysis over the program to find data- and control- dependencies between different *scopes* of the program. We consider the basic blocks of the control-flow analysis of the program as the scope in our approach. Next, based on the performed analysis, we create a scope dependency graph in which the nodes are identified scopes, and the edges are data- and control dependencies between scopes. Please note that each to-be-synthesized code snippet (created from API calls in a cluster) will be placed in a scope. In fact, by selecting a candidate location for a cluster, we basically annotate a node in the scope dependency graph with the annotation of that cluster. If we keep the annotated nodes and remove the remained nodes from the graph (and yet preserve the direct and indirect edges between the remained nodes), the final graph would represent a program-wide control- and data-dependencies between to-be-synthesized code snippets. In order to have a correct tactic synthesis, this graph should look the same as the sub-graph of the FSpec where the selected clusters belong to. If this constraint is satisfied, then, it means that the selected mapping between clusters and locations will result in a correct tactic implementation. Otherwise, it means that the combination of selected locations for clusters was not correct and thus, the selected mapping will be disregarded. In such a case, the next best candidate mapping from the ranked list of mappings will be considered for the next round of control- and data-dependency checking. The result of this analysis will be the *Control-and Data-dependency Score (CDS)* for the entire mapping, which is a binary number 0 or 1. If correct dependencies are observed, then the score will be 1. Otherwise, it will be 0.

3.2.4 Criteria 4: Synthesized Code Quality

Lastly, we would like to have the tactic implemented in a way that the final code has a high quality. There are a variety of techniques that quantify the quality of the code from different perspectives, e.g., high-cohesion and loose-coupling as two fundamental quality metrics for code [25]. In this paper, we focus on *high-cohesion*, but we will discuss how *loose-coupling* could also be incorporated as future work.

High-cohesion and loose coupling can be measured based on considering different scope granularities, including method-level, class-level, package-level, and module-level. In this paper, without affecting the generalizability of the approach, we focus on class-level granularity, meaning that we intend to have classes where single functionalities are implemented in separate methods to improve the *cohesion*, while functional methods (i.e., methods that implement functionalities) have low dependencies to each other to reduce the *coupling*. However, one can easily change the granularity of the scope, e.g., consider package-level or module-level scopes.

The general idea is to give a higher score to the implementation that adds functionality-relevant code (i.e., lines of code that collaboratively implement a single functionality) to the same method and places those that implement different functionalities in different methods (i.e., high-cohesion). Also, to keep a loose coupling, one would keep the (direct) interaction between these methods as minimum as possible.

As we discussed before in Section 3.1, IPSYNTH considers API calls that collaborate to implement a single functionality in a cluster. Also, IPSYNTH always generates the corresponding code to a cluster of API calls as a block of code, keeping the code inside this block always together, and putting them in the same method. Therefore, it always generates and adds the lines of code that implement a single functionality in the same method. However, to guide the synthesizer to put the synthesized code for different clusters in separate methods (to avoid violation of the single responsibility rule and improve the code cohesion), we use Code Quality Score (CQS) shown in Equation (4) that measures the cohesion level of the code.

$$CQS = \frac{1}{n} \sum_{i=1}^n \frac{1}{Clst(m_i)} \quad (4)$$

In this equation, n is the number of the methods that the approach puts the synthesized code snippets in, m_i is a method that at least one synthesized code snippet is placed in, and the *Clst* function counts the number of code snippets that were placed in method m_i . Aiming for a high *CQS* helps IPSYNTH to synthesize high-quality tactic implementation (w.r.t. high-cohesion). The highest possible score is 1 where each code snippet is placed in a separate method. On the other side, the more functionality-irrelevant code snippets placed in a single method, the closer this score can get to 0.

For example, let's assume that there are two methods m_1 and m_2 where there are two code snippets synthesized and placed in m_1 and one code snippet synthesized and placed in m_2 . Based on the introduced *CQS*, the quality code score for such a synthesis task will be $\frac{1}{2}(\frac{1}{2} + \frac{1}{1}) = 0.75$.

Please note that this score only takes care of the code quality. For example, if there is a method in the program that there should be a synthesized code snippet placed in it but it is not, then, the other metrics (Criteria 1 and 2) will take care of this issue.

The introduced quality metric (Equation (4)) focuses on high-cohesion, however, it would be possible to extend it with Equation (5) that incorporates the coupling status of the code and computes the code *instability* [26].

$$CCS = \frac{1}{n} \sum_{i=1}^n \frac{Ce_{m_i}}{Ce_{m_i} + Ca_{m_i}} \quad (5)$$

In this equation, n is the number of the methods that the approach puts the synthesized code snippets in, m_i is a method that at least one synthesized code snippet is placed in, Ce_{m_i} is the *efferent* coupling, i.e., the number of other methods that method m_i depends upon, and Ca_{m_i} denotes the *afferent* coupling, i.e., the number of other methods that depend on method m_i . A CCS score close to 0 indicates a stable method with fewer dependencies, while a value close to 1 shows a high dependency on the other methods.

3.2.5 Putting All Together

We incorporate all four constraints above to spot the proper locations in the code for creating and adding to-be-synthesized codes to the program. More specifically, we use the first two scores (MNS and VAS) to generate a cluster-location score (CLS) to find and rank appropriate locations per cluster. Equation (6) demonstrates how we generate this score for each mapping.

$$CLS = \frac{(c_{MNS} \times MNS) + (c_{VAS} \times VAS)}{c_{MNS} + c_{VAS}} \quad (6)$$

In this equation, MNS and VAS are methods naming similarity and variable availability scores computed before. In addition, c_{MNS} , and c_{VAS} are coefficients for MNS and VAS . These coefficients denote the effect of each quantified criterion in the final score. For the sake of simplicity, we consider all the coefficients as 1 in our computations. However, in future work, it would be possible to leverage a training procedure using a training dataset to statistically learn each coefficient. The CLS score quantifies the worthiness of a location for adding a synthesized cluster to that location in the program. However, a tactic synthesis process is composed of a couple of cluster synthesis tasks. For a successful and correct tactic synthesis, we need to make sure that (i) the final code has high quality and (ii) there are correct control- and data dependencies between synthesized cluster-related code snippets in the final code. Considering all the selected mappings, Equation (7) generates a score for the entire mappings in the synthesis task.

$$CAS = \begin{cases} 0 & CDS = 0 \\ \frac{(cc_{CQS} \times CQS + cc_{CLS} \times (\frac{1}{n} \sum_{i=1}^n CLS_i))}{cc_{CQS} + cc_{CLS}} & CDS = 1 \end{cases} \quad (7)$$

In this equation, CLS_i is the computed CLS score for the i^{th} mapping in the synthesis task, n denotes the total number of mappings in the synthesis task, CQS is the code quality score for all the clusters incorporated in a synthesis

task, and CDS is the control- and data-dependency score computed for the mappings in the synthesis task. Moreover, cc_{CQS} and cc_{CLS} are coefficients for applying the effect of each of the CQS and CLS in the final score. Based on our observation, it would be very helpful if the cc_{CQS} is very low, just enough to make a difference when the CLS scores of two mappings are equal. Throughout our experiments, we found 0.0001 to be an efficient value for cc_{CQS} while cc_{CLS} is set to 1. Furthermore, in case the control- and data-dependency criteria is not satisfied (i.e. $CDS = 0$), the formula completely disqualifies the list of selected mappings and returns 0. In such a case, the synthesizer goes over the ranked list of the locations for clusters and tries to find substitute mappings such that the combination of CLS and CQS is the maximum possible value while the CDS is not 0.

Please note that considering only correct locations for synthesizing clusters will result in a **correct by construction** approach. In other words, by following this algorithm, we make sure that the final synthesized tactic is implemented correctly.

3.3 Step #3 - Sketch Generation

Through the previous step, we identified candidate locations in the program per cluster such that the to-be-synthesized code snippet for that cluster will be placed in that location. Each cluster consists of a set of APIs and their dependencies (i.e., control and data dependencies) represented as a graph in which the nodes are API calls and the edges are their dependencies. In this step, we translate this graph into a code snippet that is only composed of those API calls and generate the structure of the to-be-synthesized code snippet. This code snippet is in the format of a Static Single Assignment (SSA). For instance, if there is a data dependency between API A_1 and A_2 in a cluster such that A_2 uses the data generated by A_1 (i.e., $A_1 \xrightarrow{\text{data}} A_2$), the output of A_1 would be stored in a variable v_1 such that v_1 is an input argument to or a target object for A_2 in the corresponding generated code snippet. However, there are still two dependencies remained to be added to this code snippet, (i) dependencies between the code snippet and variables available (visible and initialized) at the location in which the code snippet will be added to, and (ii) dependencies between this code snippet and the other generated code snippets which reflects the dependencies between annotated clusters. To address these dependencies, in this step, we put *holes* in the generated code snippet and we will fill in these holes later in the next step (Section 3.4). Each hole is a placeholder for a variable in the code snippet. In other words, while translating a graph to a code snippet, if an argument or a target object is not filled with an API from the same graph, we put a hole in that part of the code snippet. The output of this step would be a *sketch* of the final version of the to-be-synthesized code snippet for each cluster. For instance, `#Initialization`, `#Logging_In`, and `#Subject_Inspection` annotations will be translated into the sketches shown in Figure 5. In this figure, question marks (i.e., ?) are the holes needed to be filled out by variables available at the location where the sketch will be added to. Note that from the

```

#Initialization
CallbackHandler callbackHandler = new MyCallbackHandler();
LoginContext loginContext = new LoginContext( ?[java.lang.String], callbackHandler );

#Logging_In
?jaxx.security.auth.login.LoginContext].login();

#Subject_Inspection
Subject subject = ?jaxx.security.auth.login.LoginContext].getSubject();
for( Object principal: subject.getPrincipals() )
    System.out.println( principal );

```

Fig. 5. Sketches generated for #Initialization, #Logging_In, and #Subject_Inspection annotations.

FSpec, we know the signature of all APIs inside a cluster. Therefore, wherever we put a hole in the constructed sketch, we are also aware of the type of variable that should be used to fill in that hole. For instance, the FSpec provides the signature of the constructor of `LoginContext`, including the types of its input arguments. Hence, we can infer that the type of hole in the constructor of `LoginContext` in Figure 5 is `java.lang.String`. Also, holes available in sketches created for #Logging_In and #Subject_Inspection are of type `javax.security.auth.login.LoginContext`.

3.4 Step #4 - Sketch Resolution

So far, we have created sketches of to-be-synthesized code snippets. These sketches contain holes that need to be filled in by available variables. From one side, we know the type of each hole, and from another side, there are a limited number of variables available with the same type as the hole's type. This is a constraint-based problem in which we need to find a solution that satisfies all the constraints. In this step, based on the mentioned constraints, we translate the problem of finding proper variables for filling in the holes in sketches to a Satisfiability Modulo Theory (SMT) problem. Then, we use an off-the-shelf SMT solver, Z3 [27], to find a proper solution that satisfies all the constraints. Finally, we translate back the found SMT solution to a concrete solution for the original problem of finding proper variables for holes.

To do this, we first collect information about all the available and initialized variables that were identified through Step 2 to create a ranked list of possible variables for each hole. The closer the variable is to the location of a hole in the code, the higher its position would be in the ranked list. Let us assume that for each hole h_i , there would be a list of n possible variable names $[v_{i,1}, v_{i,2}, \dots, v_{i,n}]$ to choose from. The final goal is to find a minimum number of variables such that all the holes could be filled with . We first translate the sketch problem to the following SMT problem:

$$\begin{aligned} Constraint_1 = & (v_{1,1} \vee v_{1,2} \vee v_{1,3} \vee \dots \vee v_{1,m_1}) \wedge \\ & (v_{2,1} \vee v_{2,2} \vee v_{2,3} \vee \dots \vee v_{2,m_2}) \wedge \quad (8) \\ & \dots \wedge (v_{n,1} \vee v_{n,2} \vee v_{n,3} \vee \dots \vee v_{n,m_n}) \end{aligned}$$

where $v_{i,j}$ is a Boolean variable that, when true, indicates that the j -th element in the list of available variables for hole h_i has been selected. Please note that holes' variable lists are not mutually exclusive, meaning that it is possible that a variable is in the ranked list of more than one hole. Let us say that the i -th variable of the ranked list of hole h_1 and

the j -th variable of the ranked list of hole h_2 are the same. Moreover, the k -th variable of the ranked list of hole h_1 and the s -th variable of the ranked list of hole h_3 are the same. In such a case, we have another constraint as shown in Equation (9).

$$Constraint_2 = (v_{1,i} \iff v_{2,j}) \wedge (v_{1,k} \iff v_{3,s}) \quad (9)$$

where the iff operator (i.e., \iff) implies that the value of its operands should be the same. Since we want to satisfy all the constraints at the same time, the final SMT problem would be the conjunction of $Constraint_1$ and $Constraint_2$, i.e., $Constraint_1 \wedge Constraint_2$. Please recall that the goal of this step is to use the minimum number of available variables for filling out all the holes in the sketch. Therefore, we ask the SMT solver to find the minimum number of boolean variables such that $Constraint_1 \wedge Constraint_2$ can be *True*.

4 EVALUATION

This section details our experimental study of IPSYNTH for synthesizing tactic codes and adding them to a program. Moreover, it compares the synthesis results of IPSYNTH against a powerful code generator tool, ChatGPT [19].

For the first time, we have created a dataset of architectural tactic synthesis tasks to be used to evaluate program synthesis techniques. This dataset was used as part of our experimental study in this paper. To evaluate the introduced approach, we conduct two types of evaluations: (i) a fine-grained evaluation of the components (i.e. steps) of our approach (Section 4.2), and (ii) an evaluation of the performance of the entire approach (Section 4.3). While in the former we investigate the performance and correctness of each component, in the latter, we inspect the output of the entire approach. We also compare the generated code by IPSYNTH against one of the most powerful related works that is an expert in code generation, ChatGPT [19]. In the rest of this section, we first introduce the dataset, then, we discuss the experiments and their results, and finally, we compare the results of our approach against the related work.

4.1 Architectural Tactic Synthesis Dataset

We created a dataset of architectural tactic synthesis tasks which includes test programs and their related labels. Each task represents a unique way of implementing a *security* tactic. The synthesizer is allowed to use JAAS API for synthesizing the tactic. The current version of the dataset consists of 20 programs. Each curated test program is syntactically correct, and there are places in the program where the synthesizer is expected to generate tactic pieces and add them to the mentioned locations. The correctness of test programs was confirmed by the Java compiler (syntax correctness) as well as the review team where the members were expert programmers and completely familiar with the JAAS framework. Figure 6 shows such a program where tactic codes are expected to be implemented in three separate methods that inter-procedurally collaborate to execute the tactic. Since we know where tactic pieces should be added,

```

1. public class JaasImplementor {
2.     String varStr;
3.     public LoginContext initializeLC(String name) throws LoginException(){
4.         //TODO: To be implemented
5.         return null;
6.     }
7.
8.     public void login(LoginContext loginContext) throws LoginException(){
9.         //TODO: To be implemented
10.    }
11.
12.    public void inspectSubject( LoginContext loginContext ){
13.        //TODO: To be implemented
14.    }
15.
16.    public static void main(String[] args) throws LoginException{
17.        String moduleName = args[0];
18.        JaasImplementor jaasImplementor = new JaasImplementor();
19.        LoginContext loginContext = jaasImplementor.initializeLC(moduleName );
20.        jaasImplementor.login(loginContext );
21.        jaasImplementor.inspectSubject(loginContext );
22.    }
23.
24. }
```

Fig. 6. A sample test program for the tactic synthesis task.

```

Init LoginContext(String; CallbackHandler;) : JaasImplementor [3-6]
Init CallbackHandler(String; String;) : JaasImplementor [3-6]
LoginContext.login() : JaasImplementor [8-10]
LoginContext.getSubject() : JaasImplementor [12-14]
Subject.getPrincipals() : JaasImplementor [12-14]
```

Fig. 7. A test label for the test case shown in Figure 6.

we were able to create a data structure that represents the expected locations of tactic pieces; we used that information as the label for each of the created test cases. Figure 7 presents the created label for the test program provided in Figure 6. Each label is in the format of a dictionary where the *key* is the tactic piece and the value is the possible lines in a file that the synthesized code is expected to be added such that the implementation is correct. In our experiments, we use this information to assess the correctness of the synthesized code.

All the test cases are inter-procedural tasks, meaning that there are more than one method (in average 4.5 methods) in a test case that collaborate to accomplish the tactic implementation task. We made this dataset publicly available at <https://anonymous.4open.science/r/Anonymous-82DE>. Table 1 provides a summary of this dataset. In this table, programs are numbered from 1 to 20. For each program, we provide its corresponding information including the number of methods in the program, number of global variables that might affect the synthesis process, number of classes, number of java files, and finally, a description of the complexity of each program compared to its previous program. Please note that the complexity of programs increases by their numbers, i.e., implementing a tactic in *P*10 is more complicated compared to *P*5.

In addition to the evaluation of the performance of IPSYNTH on the introduced dataset, we also studied the performance of IPSYNTH in comparison with one of the most powerful tools in code synthesis, ChatGPT. This study gives us a better understanding of the advantages of our semi-formal approach against LLM-based approaches such as ChatGPT. Moreover, we learned valuable lessons about the possibility of mixing the strength of IPSYNTH and LLM-based approaches for improving the results. We discuss

these findings in this section (Section 4.3.1.2), as well as in the future directions of the work (Section 7).

4.2 Evaluating Approach Components

We first investigate each component of the approach to find a better insight into the performance of that component. This would help us pinpoint possible bottlenecks as well as the strength points of the approach.

4.2.1 Step 1: FSpec Annotation

There are two main criteria for the evaluation of FSpec annotation (introduced in Section 3.1), (i) clustering API elements in the FSpec, and (ii) selecting a proper label for each cluster. For this purpose, the review team reviewed all clusters and their associated annotations. The team members are familiar with security tactics, especially with the JAAS framework.

- *API Clustering*: The built framework API specification model (FSpec) has 88 nodes, i.e., API elements, and 130 edges, i.e., dependencies. Following a similar approach discussed in Section 3.1, we created clusters of APIs. Each cluster contains a set of API calls that contribute to a functionality.

- *Label Generation*: We also followed a similar approach to what was discussed in Section 3.1 and generated labels (i.e., annotations) for clusters. The review team then verified the quality of generated annotations based on two metrics: (i) the name similarity metric between API names in a cluster and its generated annotation, as well as (ii) a manual review process that the review team performed to assess the relevancy of the generated annotations with the name of the methods in which APIs within the cluster can be placed in.

4.2.2 Step 2: Code Annotation

As discussed before in Section 4.1, we inspected all the test cases and created a list of correct locations for each of the tactic pieces to be synthesized and added to. In the following experimental studies, we measure the accuracy of the returned recommendations using two metrics that are widely used in the recommendation tasks, *Hit Ratio@K* (*HR@K*) and *Mean Reciprocal Rank* (*MRR*). Please note that due to the different number of possible correct answers per test case, measuring *Recall@K* and *Precision@K* would not be an option in our experimental studies. Therefore, we do not include these metrics in our experimental studies. To calculate the *HR@K*, we use the following equation:

$$HR@K = \frac{100}{n} \sum_{i=1}^n (hasCorrect(M_i[1:k]) ? 1 : 0) \quad (10)$$

where *k* denotes the cutoff point of the ranked list of suggested mappings returned by our approach, *n* represents the number of test cases in the evaluation process, *M_i[1..k]* is the sub-list of the ranked recommendations for test case *i* from element 1 to the *k*th element, and finally, *hasCorrect(...)* checks whether any of the mappings in the given list are amongst the correct mappings. For instance, if *k* = 1 (i.e., top-1), we only consider the first mapping in the ranked suggestion list returned by our code annotator and investigate whether that mapping is correct. Likewise, if *k* = 5, it means that we let the code annotator return a ranked list of

Prog. No.	# of methods	# of global vars	# of classes	# of files	Added complexity
P1	4	1	1	1	
P2	4	1	1	1	Nested blocks
P3	4	4	2	1	An inner class
P4	5	4	2	1	More methods
P5	5	2	1	1	Two non-initialized global vars in the main class
P6	5	2	1	1	A global and a static non-initialized string var
P7	6	2	1	1	More methods
P8	6	2	1	1	Different types of argument passing
P9	7	1	1	1	A global initialized double var
P10	8	1	1	1	Nested method calls
P11	1	1	1	1	Non-initialized global field
P12	2	1	1	1	A not-called method
P13	6	1	2	1	A non-initialized boolean field + an inner class
P14	7	4	2	1	Nested method calls + an inner class with a private method.
P15	8	7	3	1	Nested method calls + 2 inner classes, both with private methods
P16	9	7	4	2	Two java files in the same package, 2 inner classes with private methods, nested method calls
P17	11	7	5	3	Three java files in the same package, 2 inner classes with private methods, nested method calls
P18	14	8	6	4	Four java files in 2 different packages, 2 inner classes with private methods, nested method calls
P19	17	10	8	5	Five java files in 3 different packages, 3 inner classes with private methods, nested method calls
P20	21	15	10	6	Six java files in 4 different packages, 4 inner classes with private methods, nested method calls

TABLE 1
Description of the tactic synthesis dataset

5 suggestions and we search for a correct mapping among the returned list. Moreover, we use Equation (11) to calculate the MRR metric for the list of recommendations.

$$MRR = \frac{1}{n} \sum_{i=1}^n \left(\frac{1}{rank_i} \right) \quad (11)$$

In this equation, n denotes the number of test cases, and $rank_i$ refers to the rank position of the first correct mapping in the returned mapping list. Please note that, unlike HR@K, in which we find accuracy based on different cutoffs (i.e., k), we consider the entire recommended list while computing the MRR of the recommendations. In our experiments, we consider the maximum length of the ranked list as 100. Therefore, if a correct answer was not found in the ranked list, the $\frac{1}{rank_i}$ part in Equation (11) would be considered as 0.

- *Method Name Similarity*: We first only consider the MNS score (Section 3.2.1) to find the correct spots in the program to synthesize each FSpec annotation. We implemented the Levenshtein distance similarity through dynamic programming, which significantly reduces the run-time of the MNS computation process. The first row of Table 2 shows the result of this experiment. When we only incorporate the MNS score to identify locations, on average, only 80% of the top-1 locations suggested by the approach point to the correct spots in the program for the synthesis task. When extending

the accepted suggestions to top-100, 90% of the returned suggestions are correct. Moreover, the MRR of the returned suggestions is 0.75. Based on this experiment, relying only on the method naming similarity score (MNS) would not result in a precise location spotting in our approach.

- *Available Variable*: Next, we only consider the AVS score (discussed in Section 3.2.2) for finding the correct locations in the code for the tactic synthesis task. The second row of Table 2 presents the result of this experiment, which shows a worse result compared to the situation in which we only use the MNS score. For example, the hit ratio of top-1 dropped from 80% to 15%. The main reason is that in each program there are a couple of methods that might provide all the required available variables for a code piece. Therefore, relying only on AVS can lead to confusion for the synthesizer and it might choose an undesired method at the end. Also, the MMR for this experiment is 0.21, meaning that if available, the correct suggestion can be found between the fifth and sixth mappings in the recommended list.

- *Code Quality*: In the next experiment, we consider the effect of code quality alone, without other metrics. In this case, we assume that the MNS score, as well as the AVS score for all the mappings, are both 1, meaning that we do not bring their effect into the code annotator. As shown in the third row of the Table 2, the *CQS* cannot provide insightful

Criteria	HR@K								MRR
	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 10$	$k = 50$	$k = 100$	
MNS	80%	80%	80%	80%	80%	80%	80%	90%	0.75
AVS	15%	20%	20%	20%	20%	45%	50%	85%	0.21
CQS	5%	5%	5%	5%	5%	5%	10%	0.05	
CAS	85%	90%	90%	90%	90%	90%	90%	100%	0.88

TABLE 2
Code annotation Hit Ratio for Top-K suggestions (HR@K) per scoring method.

guidance for the synthesizer, however, as we discuss later, it can contribute to better annotation when used with the other metrics.

- *Control- and Data-dependency*: We do not provide a separate evaluation for the Control- and Data-dependency Score (CDS). The reason is that this score basically quantifies the worthiness of a mapping that has been created based on MNS, CQS, and AVS scores. Therefore, we do not generate a list of ranked suggestions only based on this score. However, the effect of CDS would be observable when all the scores are incorporated (all together) as discussed in the next item.

- *All Together*: Finally, we incorporate all the code annotation score (CAS) metrics introduced before for finding the correct locations in the code for the synthesis task. The fourth row of Table 2 presents the result of this experiment. The result shows that the hit ratio for the top-1 and top-2 suggestions are 85% and 90% respectively when considering the CAS score (All scores together). This is a significant improvement compared to using each of the metrics individually. In addition, the MMR for this metric is 0.88 which means that the correct suggestion if available, would most likely be the first element in the ranked list. The results suggest that the code annotation metric we introduced in this paper (i.e., CAS) enables the synthesizer to identify the code locations for the synthesis task with high accuracy.

4.2.3 Step 3: Sketch Generation

Each sketch is a template of a to-be-synthesized code snippet which will be a part of the to-be-implemented architectural tactic. Hence, we investigate the accuracy of sketches generated by our approach based on three criteria: (i) incorporation of a correct set of API calls in the sketch, (ii) the correctness of types associated to each hole in the sketch, and (iii) establishment of proper data- and control-dependencies between APIs inside the sketch. For this purpose, the review team reviewed all the generated sketches. Table 3 provides a summary of this investigation. More details are provided as follows:

- *Correct set of APIs*: There are specific API elements in each FSpec cluster that must be appeared in the corresponding sketch. These API calls are basic building blocks of the to-be-synthesized pieces. We inspected all the generated sketches to identify if they contain all the API elements from their corresponding FSpec clusters. There were 50 API calls identified in clusters. We were able to confirm that all API calls (100%) from FSpec clusters appeared in their corresponding sketches.

- *Correct types for holes*: There were 60 holes generated for the test cases in the experimental study. These holes were of

Criteria	# of cases	# of correct answers	Accuracy
Set of APIs	50	50	100%
Hole types.	60	60	100%
Control dep.	50	50	100%
Data dep.	60	60	100%

TABLE 3
Sketch generation results

Criteria	# of cases	# of correct answers	Accuracy
Resolving holes	60	60	100%
Compilable	20	20	100%

TABLE 4
Sketch resolving results

different types (e.g. `javax.security.auth.login.LoginContext`, `java.lang.String`, etc.). We manually inspected the associated types of each hole. All holes were associated with the correct type. Hence, the accuracy of the type assignment for the holes was 100%.

- *Correct control- and data-dependencies*: We also investigated the correctness of established inter-dependencies between APIs of sketched code snippets. Overall, there were 50 control-, and 60 data-dependencies. All the established dependencies were identified as correct dependencies. Therefore, the accuracy in establishing correct control- and data-dependencies inside each sketched code snippet was 100%.

4.2.4 Step 4: Sketch Resolving

The expected output of the sketch resolving step (Section 3.4) is a program without holes. Moreover, the program is supposed to be syntactically error-free, i.e., the program should be compilable. Table 4 shows the result of this experiment. We provide more details as follows:

- *Resolving all the holes*: Our synthesizer was able to resolve all 60 generated holes, which means that exactly one variable was assigned for each hole. In addition, the type of all the assigned variables was matched with the type of their corresponding holes.

- *No syntax error*: Moreover, at the end of the synthesis process, we inspected the synthesized programs (20 programs) to assess their syntax correctness. There were two types of syntax errors in the synthesized programs, none

of which was directly related to the synthesis task. First, since new types (e.g., `javax.security.auth.login.LoginContext`, `javax.security.auth.callback.CallbackHandler`, etc.) were added to the code, the corresponding *imports* had to be added to the program. This is similar to when a programmer copy-pastes a code snippet from a Q&A forum (e.g., Stack Overflow) and adds it to the code. Since the synthesizer knows the type of API calls that are used in the synthesis process, we will be able to add this feature to our synthesizer in the future. Second, there were some abstract classes or interfaces that needed customized implementations. For instance, `CallbackHandler` is an interface that has a single method, `handle(...)`, to be implemented by the programmer. Automatically implementing such methods requires a high level of communication with the programmer. Since our goal was to automatically capture the high-level specification of the tactic, we did not cover such customization in our synthesizer. However, it would be possible to augment the approach so that it takes advantage of active learning techniques for creating a low-level specification of those methods with little communication with the programmer.

4.3 Evaluating the Entire Approach

So far, we investigated the performance of each component of our approach, IPSYNTH, separately. In this part, we run the entire approach over the test dataset and measure its performance. There are a variety of metrics (e.g., BLEU [28] and CodeBLEU [29]) developed by researchers for assessing the accuracy of code-generation approaches, however, these metrics either only focus on the syntax similarity (not necessarily the syntax correctness!) between the generated and the expected code or fail to capture some of the important semantic similarities. Therefore, in this paper, in addition to the syntax correctness of the tactic implementation, we also investigate the semantic correctness of the synthesized tactic. We compare our approach on these criteria against one of the most powerful code generators that is vastly used by programmers, ChatGPT. Additionally, we provide the run-time complexity metrics of our approach including the time and storage.

4.3.1 Semantic correctness

IPSYNTH synthesizes tactic implementations that are composed of API calls from a framework of interest. Hence, the *semantic correctness* in the context of this task refers to the correctness of incorporating APIs of the framework in the program such that the tactic is implemented correctly. We assess such correctness through two different processes, (i) an evaluation that only focuses on IPSYNTH and its performance, and (ii) a comparative analysis where we compare the performance of IPSYNTH in tactic implementation tasks against ChatGPT.

4.3.1.1 IPSYNTH-centric evaluation: In this experiment, we study the top-ranked tactic implementation returned by IPSYNTH (i.e., top-1 returned implementation) for each of the test programs introduced in Table 1. Therefore, there are 20 synthesized programs in total that were reviewed to investigate their semantic correctness w.r.t. tactic implementation. Table 5 provides the results of this effort. In this table, the *set of APIs* represents the number

Criteria	# of cases	# of correct answers	Accuracy
Set of APIs	50	50	100%
I-Control dep.	50	50	100%
I-Data dep.	60	60	100%
E-Control dep.	40	34	85%
E-Data dep.	40	34	85%
Overall Prog.	20	17	85%

TABLE 5

The semantic correctness inspection. I- and E- dependencies represent dependencies between API calls in a synthesized piece and dependencies between API calls and their surrounding environment, respectively.

of APIs that were incorporated into tactic implementation tasks. Moreover, *I-Control dep.* and *I-Data dep.* represent control- and data-dependencies between API calls inside a synthesized piece. Finally, *E-Control dep.* and *E-Data dep.* represent control- and data-dependencies between API calls of the synthesized piece and their surrounding environment. Using a correct set of API calls alongside the correct establishment of I- and E-dependencies guarantees the semantic correctness of the tactic implementation. The last row (i.e., *Overall*) shows the overall status of the semantic correctness of the synthesized programs, considering the results of all the other criteria in this table. As the results suggest, in 100% of the cases the correct set of API calls was used for the synthesis task. Moreover, the control- and data-dependencies between API calls in the synthesized pieces (I- dependencies) were correctly established in all of the cases. However, the accuracy of establishing correct data- and control-dependencies between API calls of synthesized pieces and their environment (E- dependencies) was 85%. In other words, there were three programs where the synthesizer used the correct set of API, correctly created each code piece, but put them in undesired methods in the program. Our further investigation showed that due to some similarities in method naming, the synthesizer was confused and put the code pieces in unexpected methods. Although the accuracy of the approach was not 100%, as we will discuss in our comparative analysis, we consider this as a high-accuracy synthesis compared to the related work.

4.3.1.2 Comparative Analysis: We also compared the performance of IPSYNTH against ChatGPT, a tool that programmers rely on for code generation tasks. In order to ask ChatGPT to generate the tactic code, we provided a prompt that asks ChatGPT for incorporating the JAAS framework API for adding *authentication* to the given source code. Then, we shared the content of the program with ChatGPT, i.e., copy-pasted the Java file contents after the abovementioned prompt. In our comparative analysis, we considered the first response returned by ChatGPT for each prompt. All the provided prompts as well as the ChatGPT responses are available from the dataset repository¹. Table 6 provides the results of this study. In this table, we compare the syntax correctness as well as the semantic correctness of the tactic

1. <https://anonymous.4open.science/r/Anonymous-82DE>

Prog. No.	Syntax Correctness	Semantic Correctness		
	IPSYNTH	ChatGPT	IPSYNTH	ChatGPT
P1	✓	✓	✓	✓
P2	✓	✓	✗	✗
P3	✓	✓	✓	✗
P4	✓	✓	✗	✗
P5	✓	✓	✓	✗
P6	✓	✓	✓	✗
P7	✓	✓	✓	✗
P8	✓	✓	✓	✗
P9	✓	✓	✓	✗
P0	✓	✓	✓	✗
P11	✓	✓	✓	✗
P12	✓	✓	✓	✗
P13	✓	✗	✓	✗
P14	✓	✓	✓	✗
P15	✓	✓	✓	✗
P16	✓	✓	✓	✗
P17	✓	✓	✓	✗
P18	✓	✓	✓	✗
P19	✓	✓	✓	✗
P20	✓	✓	✗	✗

TABLE 6
Comparative study results

implementation by IPSYNTH and ChatGPT. Please note that to stay fair, we only considered the first response (i.e., top-1) implementation returned by IPSYNTH. As the results show, except for one program (i.e., CharGPT failed on P13), both IPSYNTH and ChatGPT synthesized syntactically correct programs. The problem with the code implementation by ChatGPT for P13 was that it generated a code that included an invalid API (`subject.hasRole(...)`), i.e., a method call that does not exist in the JAAS framework. Thus, it caused a syntax error in the code generated by ChatGPT. However, when it comes to semantic correctness, ChatGPT was only able to correctly synthesize the first program (i.e., P1) and failed to have semantically correct implementations for the rest of the programs. For example, it generates code for part of the tactic and leaves the implementation for the rest of the tactic to the programmer. On the other side, except for three programs (i.e., P2, P4, and P20), IPSYNTH was able to correctly implement all the programs. The reason why IPSYNTH was not successful in its synthesis task for the two mentioned programs was that due to the existence of different methods in P2, P4, and P20 with similar code mapping score, IPSYNTH was confused and chose an undesired method (i.e., a method that did not align with the test label). Therefore, the synthesis result for the mentioned two programs was incorrect.

4.3.1.3 Tool-based correctness demonstration: In addition to the studies on the synthesized tactics that we presented so far, we used a tool that is specialized in detecting deviations from the correct implementation of tactics in a program to demonstrate the semantic correctness of the synthesized tactics by IPSYNTH. To that extent, we used the ARCODE PLUGIN tool [20], which is able to run an inter-procedural context-sensitive analysis over a program under

```

1. public class Jaasimplementor {
2.     String varStr;
3.     public LoginContext initializeLC(String name) throws LoginException(){
4.         CallbackHandler callbackHandler = new MyCallbackHandler();
5.         LoginContext loginContext = new LoginContext( name, callbackHandler );
6.         return loginContext;
7.     }
8.
9.     public void login(LoginContext loginContext) throws LoginException(){
10.        loginContext.login();
11.    }
12.
13.    public void inspectSubject( LoginContext loginContext ){
14.        Subject subject = loginContext.getSubject();
15.        List principals = subject.getPrincipals();
16.    }
17.
18.    public static void main(String[] args) throws LoginException{
19.        String moduleName = args[0];
20.        JaasImplementor jaasImplementor = new JaasImplementor();
21.        LoginContext loginContext = jaasImplementor.initializeLC( moduleName );
22.        jaasImplementor.login(loginContext );
23.        jaasImplementor.inspectSubject( loginContext );
24.    }
25.
26. }
```

Fig. 8. The expected implementation of the security tactic in the test program shown in Figure 6 by incorporating the JAAS framework API.

query, create a graph-based representation of how the tactic is implemented in that program, and finally, identify possible deviations from the correct tactic implementation. Based on the performed analysis, this tool reports a score in the range of [0, 1]. The closer the score is to 0, the more deviated the program is from the correct tactic implementation. We ran this tool over the synthesized programs. The average ARCODE score for the tactics synthesized by our synthesizer was 0.95, meaning that the tool identifies the synthesized tactics as highly correctly implemented. Recall from the *comparative analysis* as well as the *IPSYNTH-centric evaluation* that there were 2 programs (among the total 20 programs) where IPSYNTH was not able to completely implement the tactic in the expected way. However, even in those two programs, apart from a few external dependencies, the other parts of the implementation was correct. The 0.95 score returned by the tool reflects on this phenomenon. Figure 9 demonstrates the result of running the ARCODE tool on the synthesized program shown in Figure 8. While the graph on the left side of this figure represents how API calls were used in the synthesized program, the graph on the right side visualizes the correct way of the tactic implementation, which is similar to how the synthesized program has implemented the tactic. The score in this figure measures the similarity of the implemented tactic in the synthesized program to the correct implementation. In the case of the synthesized program, the score is 1, i.e., it is identified as a correct implementation.

4.3.2 Run-time complexities

Finally, we measured the performance of our synthesizer in terms of the complexities of time and memory during the synthesis task. We instrumented the synthesizer so that we could capture needed statistics from each component, as well as from the entire approach. Table 7 presents the measured timing and memory usages per component, as well as the entire approach. We did not include FSpec annotation as it is just a one-time process in which the approach

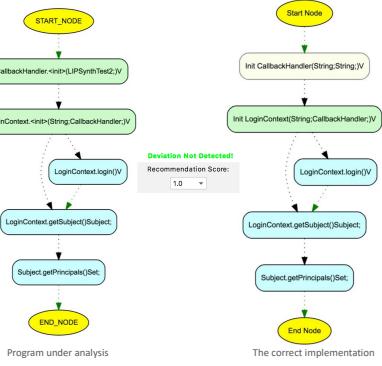


Fig. 9. A snapshot of evaluating semantic correctness of the synthesized program shown in Figure 8 by ARCODE tool.

Component	Avg. Time (ms)	Avg. Memory (MB)
Code Annotation	155.3	105
Sketch Generation	70.5	93
Sketch Resolving	33.2	25
Overall	285.2	901

TABLE 7
Run-time complexities of the synthesizer

creates the required annotations. Then, unless the FSpec is changed, there would be no more FSpec annotations while synthesizing tactics. Please note that since there are some overheads when running the entire approach, the timing and memory usage presented in the last row is higher compared to the summation of corresponding columns.

5 LIMITATIONS AND FUTURE WORK

Although the results show the effectiveness of the introduced approach in this paper, there are some limitations that need to be considered for the sake of generalizability. First, in this paper, we synthesized loop-free code snippets. We leave the task of synthesizing more complex implementation of architectural tactics to future work. Second, our experimental study has been conducted on a popular Java security framework, JAAS. Exploring more architectural tactic enabler frameworks would be beneficial for evaluating the performance of this approach in other scenarios. Next, in this paper, we created a repository of 20 test projects of architectural tactic synthesis tasks with different levels of complexity. Although this data set is introduced to the community for the first time, it would be interesting to investigate the performance of the approach through a wider range of tactic synthesis test cases. Finally, as part of the approach, we rely on method name similarities for identifying correct locations in the program for synthesizing and adding code snippets. However, in the case of obfuscated codes, this would become problematic as the method names are changed to meaningless names. To mitigate this problem, it would be possible to collect more semantic-related information around APIs while creating the FSpec. This could range from code structures to surrounding data- and control

dependencies around each API call. This information could be used while computing the score of different locations in the program (Section 3.2) with a higher coefficient.

6 RELATED WORK

Synthesizing a program based on a given list of components has been studied for over a decade [30], [31]. These components can range from simple instructions supported by the underlying programming language (e.g., mathematical and logical instructions) [32], [33], [34], to more complex components that implement crucial functionalities and use-cases (i.e., APIs) [35], [36], [37], [38], [39], [40].

6.1 API-based Program Synthesis

In recent years, a number of API-based program synthesis approaches have been developed. Morpheus [41] takes I/O examples of the desired to-be-synthesized R program, in addition to an optional set of components for the data manipulation task. It also incorporates a specification expressed by the user in the format of first-order logic. There have been some efforts [42] to leverage proof-based approaches to find a program that is constructed from a set of components and yet satisfies a series of constraints without exhaustively searching the program space. In another work [43], researchers apply constraints on a given list of components from functional and non-functional requirements points of view to synthesize a desired block of code. DAPIP [44] learns from a repository of examples of how to concatenate APIs in order to perform specific data manipulation tasks. SyPet [12] synthesizes a block of code based on a given list of APIs, the signature (i.e., type of input and output) of the expected code snippet, as well as some test cases. EdSynth [13] and FrAngle [14] are able to construct a short code snippet with more control structures compared to SyPet. TYGAR [15] aims to improve the performance and scalability of previous API-based synthesizers. ALPS [45] leverages a syntax-guided approach to learn from examples of Datalog programs on how to generate a sequence of Datalog rules for a given task. Hoogle+ [46] creates a code snippet from Haskell popular libraries based on the specified signature of the function and a set of input/output examples. In another work, SyRust [47] relies on the Rust language ownership type system to create a program from a sequence of Rust language API calls. RbSyn [48] aims to consider the side effects of API calls as guidance while synthesizing Ruby programs. It relies on the specified type of output of the program, as well as a set of test cases. ReSyn [49] leverages a typing system alongside a resource bound provided by the user to construct a program that is resource-efficient.

6.2 Leveraging API Usage Knowledge

Although the mentioned approaches use APIs as building blocks, they do not use some precious information about APIs, i.e., API usage patterns, in the synthesis process. In some of the follow-up works, researchers incorporated API usage knowledge in the synthesis task which resulted in improvement of the performance and accuracy of the synthesizer. In that regard, researchers observed a significant improvement in the run-time of state-of-the-art API-based

program synthesizers (e.g., SyPet) when they benefit from pre-learned API usage patterns [50]. ProSy [16] learns and uses a probabilistic API usage model in the synthesis task to reduce the program search space. This results in a reduction of the synthesis process by up to 80% compared to SyPet. In another similar work [17], a general API usage search engine has been developed to guide the synthesis process which resulted in an 86% reduction of synthesis time compared to SyPet. ITAS [51] also learns API usage models from a search engine and performs a bi-directional search strategy for improving the synthesis time.

Despite the capabilities of all these approaches, they suffer from two shortcomings that make them incapable of being used in architectural tactic synthesis tasks. First, the need for providing specifications of the to-be-synthesized code, and second, the lack of ability to synthesize discrete (and yet related) code snippets in different locations of the program. In this paper, we introduce an approach that addresses these concerns.

7 CONCLUSION

In this paper, we introduced IPSYNTH, a novel program synthesis approach for implementing architectural tactics in an existing code base. This approach is able to construct the to-be-synthesized code as smaller pieces of sketches, locate proper spots in the program for each piece, generate sketches corresponding to each piece, resolve all the sketches, and implement the desired architectural tactic in the program. This is an inter-procedural program synthesis approach in which the specifications are automatically collected from a pre-learned and annotated framework API specification model, as well as the structure of the program. The experimental study shows the high accuracy of the approach for tactic synthesis tasks. Moreover, the comparative analysis against related work demonstrates the effectiveness of the approach in inter-procedural code synthesis compared to the state-of-the-art.

This approach can enable many software development tools (e.g., IDEs) to support programmers with automation of architectural tactic recommendation and implementation. Moreover, the realization of this approach as an educational tool can serve educational purposes such as teaching software architecture to program developers.

REFERENCES

- [1] M. Mirakhori and J. Cleland-Huang, "Modifications, tweaks, and bug fixes in architectural tactics," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. IEEE Press, 2015, p. 377–380.
- [2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.
- [3] M. Mirakhori, Y. Shin, J. Cleland-Huang, and M. Çinar, "A tactic-centric approach for automating traceability of quality concerns," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 639–649. [Online]. Available: <https://doi.org/10.1109/ICSE.2012.6227153>
- [4] J. Van Gurp, S. Brinkkemper, and J. Bosch, "Design preservation over subsequent releases of a software product: a case study of baan erp," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 4, pp. 277–306, 2005.
- [5] R. Gopalakrishnan, P. Sharma, M. Mirakhori, and M. Galster, "Can latent topics in source code predict missing architectural tactics?" in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Üçhüel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 15–26. [Online]. Available: <https://doi.org/10.1109/ICSE.2017.10>
- [6] I. Rehman, M. Mirakhori, 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] J. Garcia, M. Mirakhori, L. Xiao, Y. Zhao, I. Mujhid, K. Pham, A. Okutan, S. Malek, R. Kazman, Y. Cai, and N. Medvidovic, "Constructing a shared infrastructure for software architecture analysis and maintenance," in *18th IEEE International Conference on Software Architecture, ICSA 2021, Stuttgart, Germany, March 22-26, 2021*. IEEE, 2021, pp. 150–161. [Online]. Available: <https://doi.org/10.1109/ICSA51549.2021.00022>
- [8] A. Shokri, J. C. S. Santos, and M. Mirakhori, "Arcode: Facilitating the use of application frameworks to implement tactics and patterns," in *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, 2021, pp. 138–149.
- [9] H. Cervantes, P. Velasco-Elizondo, and R. Kazman, "A principled way to use frameworks in architecture design," *IEEE software*, vol. 30, no. 2, pp. 46–53, 2012.
- [10] "Java authentication and authorization services (jaas)," <https://docs.oracle.com/en/java/javase/16/security/jaas-authentication.html>, accessed: 2021-08-21.
- [11] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.
- [12] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex apis," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 599–612.
- [13] Z. Yang, J. Hua, K. Wang, and S. Khurshid, "Edsynth: Synthesizing api sequences with conditionals and loops," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 161–171.
- [14] K. Shi, J. Steinhardt, and P. Liang, "Frangel: component-based synthesis with control structures," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.
- [15] Z. Guo, M. James, D. Justo, J. Zhou, Z. Wang, R. Jhala, and N. Polikarpova, "Program synthesis by type-guided abstraction refinement," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–28, 2019.
- [16] B.-B. Liu, W. Dong, J.-X. Liu, Y.-T. Zhang, and D.-Y. Wang, "Prosy: Api-based synthesis with probabilistic model," *Journal of Computer Science and Technology*, vol. 35, no. 6, pp. 1234–1257, 2020.
- [17] J. Liu, B. Liu, W. Dong, Y. Zhang, and D. Wang, "How much support can api recommendation methods provide for component-based synthesis?" in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2020, pp. 872–881.
- [18] A. Shokri, "A program synthesis approach for adding architectural tactics to an existing code base," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1388–1390.
- [19] OpenAI, "Gpt-4 technical report," *ArXiv*, vol. abs/2303.08774, 2023.
- [20] A. Shokri and M. Mirakhori, "Arcode: A tool for supporting comprehension and implementation of architectural concerns," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 485–489.
- [21] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.
- [22] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.
- [23] D. v. Bruggen, F. Tomassetti, R. Howell *et al.*, "avaparser/-javaparser: Release javaparser-parent-3.16.1," May 2020. [Online]. Available: <https://doi.org/10.5281/zenodo.3842713>
- [24] "T.j. watson libraries for analysis (wala)," <http://wala.sourceforge.net>, accessed: 2021-08-19.
- [25] E. Yourdon and L. L. Constantine, "Structured design. fundamentals of a discipline of computer program and systems design," *Englewood Cliffs: Yourdon Press*, 1979.

- [26] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.
- [27] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
- [28] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.
- [29] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.
- [30] Y. Lustig and M. Y. Vardi, "Synthesis from component libraries," in *International Conference on Foundations of Software Science and Computational Structures*. Springer, 2009, pp. 395–409.
- [31] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 215–224.
- [32] A. Taly, S. Gulwani, and A. Tiwari, "Synthesizing switching logic using constraint solving," *International journal on software tools for technology transfer*, vol. 13, no. 6, pp. 519–535, 2011.
- [33] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with smt," in *Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis*, 2014, pp. 30–39.
- [34] Y. Zhang, W. Dong, D. Wang, J. Liu, and B. Liu, "Probabilistic synthesis for program with non-api operations," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2020, pp. 451–457.
- [35] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis," in *International conference on computer aided verification*. Springer, 2013, pp. 934–950.
- [36] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica, "Autopandas: neural-backed generators for program synthesis," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.
- [37] A. Iannopollo, S. Tripakis, and A. Sangiovanni-Vincentelli, "Constrained synthesis from component libraries," *Science of Computer Programming*, vol. 171, pp. 21–41, 2019.
- [38] K. M. Ellis, M. Nye, Y. Pu, F. Sosa, J. Tennenbaum, and A. Solar-Lezama, "Write, execute, assess: Program synthesis with a repl," 2019.
- [39] Z. Liang and K. Tsushima, "Component-based program synthesis in ocaml," 2017.
- [40] B. Liu, W. Dong, Y. Zhang, D. Wang, and J. Liu, "Boosting component-based synthesis with control structure recommendation," in *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages*, 2020, pp. 19–28.
- [41] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 422–436, 2017.
- [42] A. Gascón, A. Tiwari, B. Carmer, and U. Mathur, "Look for the proof to find the program: Decorated-component-based program synthesis," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 86–103.
- [43] A. Tiwari, A. Gascón, and B. Dutertre, "Program synthesis using dual interpretation," in *International Conference on Automated Deduction*. Springer, 2015, pp. 482–497.
- [44] S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli, "Deep api programmer: Learning to program with apis," *arXiv preprint arXiv:1704.04327*, 2017.
- [45] X. Si, W. Lee, R. Zhang, A. Albarghouthi, P. Koutris, and M. Naik, "Syntax-guided synthesis of datalog programs," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 515–527.
- [46] M. B. James, Z. Guo, Z. Wang, S. Doshi, H. Peleg, R. Jhala, and N. Polikarpova, "Diggig for fold: synthesis-aided api discovery for haskell," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.
- [47] Y. Takashima, R. Martins, L. Jia, and C. S. Păsăreanu, "Syrust: automatic testing of rust libraries with semantic-aware program synthesis," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 899–913.
- [48] S. N. Guria, J. S. Foster, and D. Van Horn, "Rbsyn: type-and effect-guided program synthesis," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 344–358.
- [49] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann, "Resource-guided program synthesis," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 253–268.
- [50] B. Liu, W. Dong, and Y. Zhang, "Accelerating api-based program synthesis via api usage pattern mining," *IEEE Access*, vol. 7, pp. 159162–159176, 2019.
- [51] J. Liu, W. Dong, and B. Liu, "Boosting component-based synthesis with api usage knowledge," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 2020, pp. 91–97.