# Specializing Neural Networks for Cryptographic Code Completion Applications

Ya Xiao, Wenjia Song, Jingyuan Qi, Bimal Viswanath, Patrick McDaniel, *Fellow, IEEE*, and Danfeng (Daphne) Yao, *Fellow, IEEE*

**Abstract**—Similarities between natural languages and programming languages have prompted researchers to apply neural network models to software problems, such as code generation and repair. However, program-specific characteristics pose unique prediction challenges that require the design of new and specialized neural network solutions. In this work, we identify new prediction challenges in application programming interface (API) completion tasks and find that existing solutions are unable to capture complex program dependencies in program semantics and structures. We design a new neural network model Multi-HyLSTM to overcome the newly identified challenges and comprehend complex dependencies between API calls. Our neural network is empowered with a specialized dataflow analysis to extract multiple global API dependence paths for neural network predictions. We evaluate Multi-HyLSTM on 64,478 Android Apps and predict 774,460 Java cryptographic API calls that are usually challenging for developers to use correctly. Our Multi-HyLSTM achieves an excellent top-1 API completion accuracy at 98.99%. Moreover, we show the effectiveness of our design choices through an ablation study and have released our dataset.

**Index Terms**—API completion, neural networks, program dependencies

## 1 Introduction

Code completion is an important building block for many software engineering tasks, including code generation and program repair. Inspired by the success of natural language modeling [1], [2], [3], [4], [5], neural network based code completion has received much attention [6], [7], [8]. Early efforts [9], [10], [11] treated programs as a sequence of source code tokens and built statistical language models on the sequential context for the next token generation. While promising, these approaches are unable to guarantee syntax and semantic correctness and hence fail to generate high-quality code [12].

With the increasing data abundance and training resources, more advancements are achieved by increasing the language model size and training data. For example, the recently published code generation engines, AlphaCode [13] and Github Copilot [14], are powered with extremely large language models pretrained on available GitHub code. To increase the quality of the generated code, they often generate a large number of (e.g., 100) candidates and rely on task-specific filtering or searching techniques (e.g., unit tests) to find the correct one. However, these approaches that require well-designed post-processing might not be applicable for tasks without clear filtering conditions. To improve the top-1 accuracy of the neural networks, program-specific challenges need to be carefully addressed. Towards this direction, many studies [7], [15], [16], [17], [18] proposed solutions that incorporate program syntax or semantic properties. Some studies focus on representing programs as structural representations, such as syntax trees [7], [15], [16], [19], [20] or graphs that show the program's control flow or data flow [21], [22], [23]. Moreover, some studies applied formal language grammar (e.g., context-free grammar, attribute grammar) working with neural networks to guide the code generation process [17], [24].

Despite these recent advances, the prediction accuracy in some code completion tasks is low, in particular, the API completion problem. For example, our experiments validate that a state-of-the-art commercial code completion tool Codota [25] only achieves 64.9% accuracy in recommending the next cryptographic API method. Figures 1 (a) and (b) have slightly different code contexts and require two distinct cryptographic API methods. However, Codota gives the same recommendation for both of them, resulting in an incorrect suggestion for (b). This indicates that it cannot identify the slight change in the program context and make correct suggestions accordingly. Another state-of-the-art neural network-based API completion solution SLANG [26] also only obtains an accuracy of around 77.4% in our study.

We choose to focus on cryptographic API completion because they are well known to be complex, low-level, and error-prone [27], [28], [29], even for experienced developers [30]. Thus, a high-accuracy cryptographic API completion solution would be necessary. Such an effort is complementary to developing vulnerability screening tools [31], [32] and benchmarks [33], [34], [35].

In the context of Java cryptographic API completion, our analysis finds that the root cause of the low prediction accuracy is the lack of ability to learn program dependencies. Thus, we design a neural network solution that understands program dependencies, with the help of specialized static analysis. We identify two previously unreported program-

- *Ya Xiao, Wenjia Song, Jingyuan Qi, Bimal Viswanath, and Danfeng (Daphne) Yao are with the Department of Computer Science, Virginia Tech, Blacksburg, VA, 24060.*
  *E-mail: {yax99, wenjia7, jingyq1, vbimal, danfeng}@vt.edu*
- *Patrick McDaniel is with the School of Computer, Data and Information Sciences at the University of Wisconsin-Madison.*
  *E-mail: mcdaniel@cs.wisc.edu*

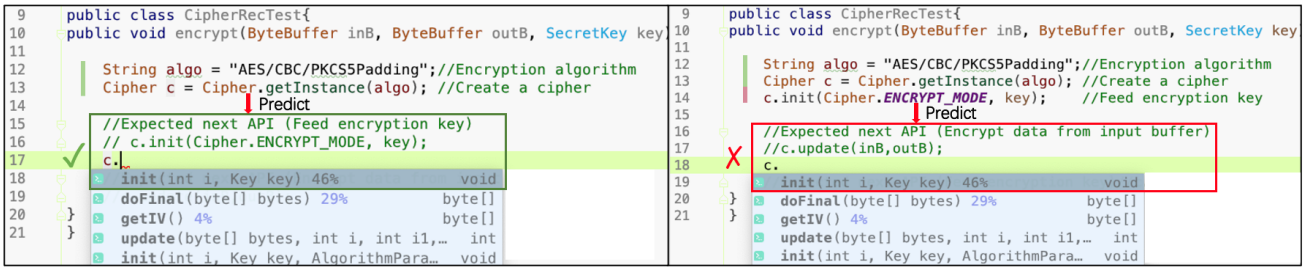



Fig. 1. API completion and accuracy. (a) Codota's recommendation for Line 17 is correct. (b) Codota's recommendation for Line 18 is wrong. The code context for (a) and (b) differs at Line 14, but Codota gives identical predictions for them.

ming language-specific challenges. The first challenge is how to recognize global dependencies. A code location can have dependencies that locate far away from the current location, even out of the current method or class, which is referred to as global dependencies in this paper. We found that the impact of global dependencies is often neglected by existing solutions, especially when they are less frequent. In our study, BERT and LSTM often fail to recognize the long but low-frequency dependence sequences when their shorter subpart is much more frequent but suggests a wrong prediction. To address it, we design a global dependence-enhancing mechanism that involves program analysis and a new neural network HyLSTM. Program analysis extracts the global dependencies. Our new neural network is sensitive to these global dependencies, even for low frequency API sequences.

We also identify a second prediction challenge on the multi-path nature of program dependencies. There are many functionally similar APIs that share most of their predecessors. We found that the existing approaches of representing program context sequentially have difficulty distinguishing these APIs and are problematic. We design a new multi-path neural network architecture to aggregate the impacts of multiple dependence paths to make predictions. Our ultimate model is named Multi-HYLSTM.

We extensively evaluate our Multi-HyLSTM design with Java cryptographic code extracted from Android apps. Our evaluation as well as case studies demonstrate that our approach Multi-HyLSTM is effective in making more accurate API suggestions, substantially advancing the state-of-the-art solutions.

Our major contributions are summarized as follows.

- We identified two previously unreported challenges for neural networks to predict code. We experimentally validated the limitation of common models in learning program dependencies, including BERT (Bidirectional Encoder Representations from Transformers) and LSTM (Long Short-Term Memory) models. We performed an in-depth manual analysis of the failed test cases to identify the weaknesses and gave case studies to document these new challenges.
- We designed a new neural network, referred to as Multi-HyLSTM, to overcome the challenges of learning the global dependencies and multi-path dependencies. Multi-HyLSTM includes two major features, a multi-path architecture and a global dependence-enhancing learning module named HyLSTM. This neural network works together with our program context representation, API dependence graph construction, and multi-path extraction algorithm, to accurately capture the program dependencies for an API call.
- We conducted a comprehensive experimental evaluation. We collected 64,478 Android Apps and identified 774,460 cryptographic API callsites from them. We conducted an extensive ablation study to validate the effectiveness of our design choices. Our solution, Multi-HyLSTM, outperforms its intermediate counterparts with a high top-1 accuracy at 98.99%. We also experimentally compared Multi-HyLSTM with two general-purpose API completion tools SLANG [26] and Codota [25].

We have also published a large-scale Java cryptographic code dataset [1] that can be used as a benchmark to evaluate API completion model accuracy.

## 2 Program Dependence Specific Challenges

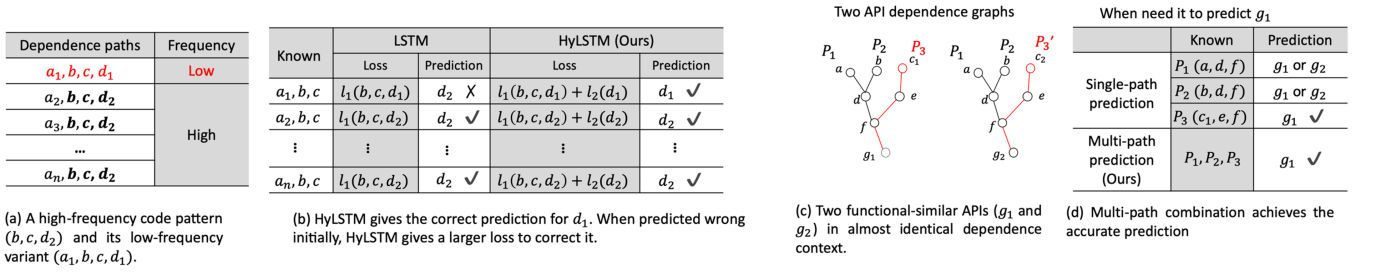
In this section, we use examples to illustrate the code prediction challenges associated with semantic dependencies in programming languages.

**Definition 2.1** (Global dependence)**.** We use global dependence of a program point $\mathrm{p}$ to refer to the code instructions that $\mathrm{p}$ depends on but locate outside of the wrapping method of $\mathrm{p}$.

*Global Dependence Challenge.* In programs, global dependencies widely exist. Locating far away, they need to be carefully extracted and covered by the neural network input. Moreover, we observed that some API patterns (i.e., API call subsequences) appear much less frequently than their shorter variants, as demonstrated in Figure 2 (a). The subsequence $(a_1, b, c, d_1)$ is a rare case compared with its variant subsequence $(b, c, d_2)$ that does not include the global dependence $a_1$. However, under the existence of $a_1$, the last token $d_1$ is the correct choice, instead of $d_2$. The high-frequency short pattern (e.g., $b, c, d_2$) makes it difficult for neural networks to recognize the low-frequency longer impacts from global dependencies. Our experiments show that both LSTM and BERT [5], [36] cannot deal with it (Section 4.2).

To address this issue, we present a new sequential model HyLSTM by modifying the LSTM loss function (Section 3.3.1).

[1]https://github.com/Anya92929/DL-crypto-api-auto-recommendation




(a) A high-frequency code pattern $(b, c, d_2)$ and its low-frequency variant $(a_1, b, c, d_1)$.

(b) HyLSTM gives the correct prediction for $d_1$. When predicted wrong initially, HyLSTM gives a larger loss to correct it.

(c) Two functional-similar APIs ($g_1$ and $g_2$) in almost identical dependence context.

(d) Multi-path combination achieves the accurate prediction

Fig. 2. Examples illustrating dependence challenges in code completion and how we address them.

Our idea is to give a strong signal to the model about the different patterns. We strategically amplify the importance of the last token (e.g., node $d_1$ and $d_2$) in the entire sequence to calculate the sequence loss. Our experiments verify that this new amplified loss function in HyLSTM makes the model better at identifying low-frequency variants (Section 4.2.1). We refer to this new loss function as *far-near loss*.

*Multi-path Dependence Challenge.* Some functionally similar API methods are difficult to be distinguished, because their predecessors overlap substantially. For example, `new String(byte[])` and `Base64.Encoder.encodeToString (byte[])` both work for encoding `byte` arrays into a `String` after the identical `Cipher` decryption operations. The shared dependence paths make them indistinguishable. As demonstrated in Figure 2 (c), the API methods $g_1$ and $g_2$ work in similar program contexts. They have two identical dependence paths $P_1$ and $P_2$. To distinguish them, a critical path $P_3$ must be captured and highlighted. We present a new multi-path architecture that can recognize their differences. Compared with a sequential model that takes a single path or a mixed context sequence as inputs, our architecture models every single path separately and aggregates their impacts.

## 3 Our Approach

We give the API completion task definition and describe our approach and the details.

**API Completion Task.** Given a program context $C = (x_1, x_2, \ldots, x_{n-1})$ where $C$ is composed of a sequence of API elements (i.e., API methods and constants) $(x_1, x_2, \ldots, x_{n-1})$, the task is to suggest the next API method call $x_n$ based on the given context $C$.

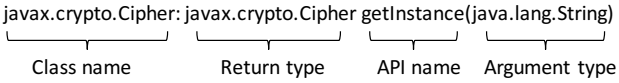



Fig. 3. An example of an embedded API method

Fig. 3 illustrates an example of an API method. We identify the API signature in Android byte code and regard every API signature as a unique token, i.e., each line is a token; each token is delimited by the new line symbol. Our dataset involves 4,543 tokens, including 3,739 API methods and 804 constants, covering standard Java cryptography libraries and associated Android API. Because of this API signature-based tokenization, our work does not have any out-of-vocabulary (OOV) situation.

### 3.1 Overview of Our Approach

Our approach takes advantage of both the program analysis and a neural network to make accurate API completion predictions. The program analysis part works to gather accurate dependence relationships between the API calls of the given program context. With the accurate dependence information, our neural network part is trained on a large code corpus to learn the conditional probability $p(x_n|C)$.

Instead of making the sequential language modeling, our approach decomposes the program context $C$ as a couple of paths reaching to the targeted location, as shown in Equation 1.

$$R(C) = P_1, P_2, \ldots, P_k \tag{1}$$

where $R(.)$ represents our program analysis preprocessing in Section 3.2. $P_i$ denotes an extracted dependence path.

Next, we calculate the conditional probability as:

$$\begin{aligned} p(x_n|C) &\simeq p(x_n|(P_1, P_2, \ldots, P_k)) \\ &= M(p(x_n|P_1), p(x_n|P_2), \ldots, p(x_n|P_k)) \end{aligned} \tag{2}$$

where $M(.)$ is a function that accepts the probability given each path and generates the overall conditional probability.

As Equation 2, our neural network, Multi-HyLSTM, includes two major components. One calculates the conditional probability given a single path $p(x_n|P_i)$. The second component acts as the function $M(.)$ to aggregate the impacts of every single path.

### 3.2 Dataflow based Context Extraction

We apply a specialized dataflow analysis to process the given program context and generate neural network features. The neural network features are a couple of data dependence paths between API calls. To make an accurate analysis, we use the dataflow analysis algorithm of the static analyzer CryptoGuard [31], which achieves high precision in cryptographic code screening. By performing the dataflow analysis, we have multiple steps to get the API dependence paths from the Android byte code, including INTERPROCEDURAL BACKWARD SLICING, API DEPENDENCE GRAPH CONSTRUCTION, and MULTI-PATH SELECTION.

INTERPROCEDURAL BACKWARD SLICING. We perform an interprocedural backward slicing to gather program slices from the Android byte code. A program slice is a subset of program statements that have influences on a specified value. Our specialized dataflow analysis is conducted for the purpose of interprocedural backward slicing. The

analysis starts at the program point where a targeted API method call happens. The analysis traces the data flow backwardly and collects all the program statements that have dependence relationship with the starting point. To guarantee the global dependences are captured, our analysis is interprocedural, which goes beyond the method boundary and collects dependence paths across the entire program. When encountering a code statement invoking a self-defined method created by the developer, the analysis jumps into the implementation body of the invoked method and replaces it with its implementation code. In this way, the extracted context is composed of the API method calls from standard libraries and eliminates the self-defined methods that only exist in the current program. Finally, the obtained program slice preserves all the code statements contributing to the targeted API call. All the irrelevant code statements are removed in the outcome.

API DEPENDENCE GRAPH CONSTRUCTION. Next, we leverage the data flow information we gathered during the dataflow analysis to further construct an API dependence graph. First of all, we give definitions for the concepts used in our API dependence graph construction.

**Definition 3.1** (API call node)**.** An API call node refers to an API method call as well as the analyzed information associated with it. Each API call node records 1) the container method the API call locates in, 2) the code statement that includes the invoked API method and the associated variables (i.e., object reference, arguments, and return values), 3) the control dependence of this invocation statement. An example is demonstrated as Node 8 in Figure 4.

**Definition 3.2** (API dependence edge)**.** An API dependence edge connects two API call nodes when there is a data flow from one API call node to the other with no intervening API call node. The data flow is composed of a variable chain $r_1, r_2, \ldots r_n$ where the variable $r_i$ is data dependent on its predecessor $r_{i-1}$. The edge records the start and end variables of this chain, that is, the variable flowing out of the start API call node and the variable flowing in the destination API call node. An example of the edge information is demonstrated in Figure 4.

**Definition 3.3** (API dependence graph)**.** An API dependence graph $G = (V, E)$ is a graph composed of a set of API call nodes $V$ and API dependence edges $E$.

The API dependence graph is constructed on the slice obtained from the interprocedural backward slicing. First, we treat every code statement in a slice as a regular code statement node. Then, we add data dependence edges between the regular nodes according to the data flow information. The data dependence edge exists between two nodes when a node uses a variable whose value is defined or changed by the other node. Next, we remove all the nodes that do not include an API method call or constants from the graph and only keep the API call nodes. If there exists a path of removed regular nodes between two API call nodes, we connect the two API call nodes directly with an API dependence edge. As a result, we obtain an API dependence graph that explicitly draws the dependence between API elements.

**Definition 3.4** (API dependence path)**.** An API dependence path is a sequence of API call nodes that are connected by the API dependence edges in an API dependence graph.

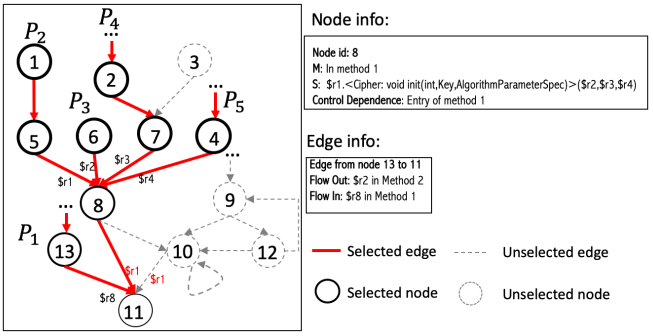



Fig. 4. Multiple paths selection for API completion. We use the information associated with the nodes and edges to select paths. The goal is to maximize the coverage for different nearby branches with minimal paths.

MULTI-PATH SELECTION. We extract multiple API dependence paths from the API dependence graph. We analyzed 957,151 graphs involved in the experiment; the number of paths varies from 1 to 798,598. The median number of paths is 3 and the majority (66%) of the graphs result in 5 or fewer paths. To avoid path explosion, we limit the number of selected paths to 5 in our experiments. We also found five is sufficient, as more paths bring negligible improvement. We use a greedy strategy to collect paths in an API dependence graph backwardly. The detailed algorithm is shown in the pseudo code (See Algorithm 1 ) in the appendix. Intuitively, our goal is to **maximize** the coverage of different nearby data and control flow branches with a **minimal** number of paths. As illustrated in Figure 4, we start from node 11. There are three edges to node 11 from nodes 8, 10, and 13. After examining the associated flow-in variables, we find nodes 8 and 10 deliver identical variable $\$r1$. Thus, we can select either one of them. In this example, nodes 8 and 13 are selected. They deliver different variables. We continue this breadth-first backward traversal until the path budget is used up. After that, we complete each selected branch to form a path via the depth-first search to an arbitrary beginning node. This greedy breadth-first approach outputs the locally optimal choice at every branch from the nearest to the farthest. The selected paths are used as neural network inputs.

Note that our prototype cannot distinguish virtual calls. The capability of distinguishing virtual calls depends on how precise the call graph is. We use CryptoGuard built on Soot to perform the dataflow analysis. When CryptoGuard constructs the call graph, all the potential virtual calls are added. Our analysis is context-, flow-, and field-sensitive, however, due to overhead it does not support any point-to analysis, which is required to handle virtual calls.

### 3.3 Our Neural Network Design

We design our neural network Multi-HyLSTM based on a multi-path architecture. In this architecture, each path is processed by a sequence model, HyLSTM, and the paths are then aggregated to the prediction outcome.

### 3.3.1 Global dependence enhancing learning

The purpose of designing HyLSTM is to improve the single-path modeling. As shown in Figure 2(a), high-frequency suffix makes the neural network ignore the beginning global dependencies. Intuitively, we force the model to assign larger weights to the beginning tokens (e.g., $a_1$ in Figure 2) when needed, making beginning tokens more influential for predicting the API (e.g., $d_1$). This is achieved by strengthening the different supervision signals (the last token) in the entire sequence.

HyLSTM differs from regular LSTM based language modeling in its architecture and loss function. We illustrate the HyLSTM architecture in Figure 5. It includes two parallel projection layers $FCL_1$ and $FCL_2$ after the LSTM cells. In contrast, regular LSTM based sequence learning only has the $FCL_2$ layer. We use the output of $FCL_1$ to generate our target (i.e., the recommended API method), given a dependence path.

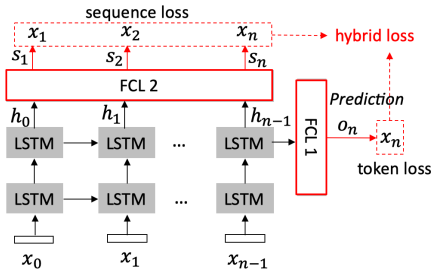



Fig. 5. Target amplification in HyLSTM with our new far-near loss function. Through the extra $FCL_1$ and $o_n$, similar input sequences followed by different $x_n$ are supervised by stronger signals to highlight their differences.

We use $h_i$ to represent the LSTM hidden state at the $i$-th timestep. In the forward propagation, $h_i$ is processed by a fully connected layer $FCL_2$ to generate the next token at every timestep.

$$\begin{bmatrix} s_1 \\ s_2 \\ \dots \\ s_n \end{bmatrix} = softmax( \begin{bmatrix} h_0 \\ h_1 \\ \dots \\ h_{n-1} \end{bmatrix} W_2 + \begin{bmatrix} B_2 \\ B_2 \\ \dots \\ B_2 \end{bmatrix} ) \quad (3)$$

where $W_2$ and $B_2$ are the weights and bias for $FCL_2$, respectively. $s_i$ represents the output of $FCL_2$ at the $i$-th timestep.

In HyLSTM, we add a fully connected layer $FCL_1$ that accepts $h_{n-1}$ that is the LSTM hidden state at the last timestep as Equation 4.

$$o_n = softmax(h_{n-1}W_1 + B_1) \quad (4)$$

where $W_1$ is a weight matrix, $B_1$ is the bias vector for $FCL_1$, and $o_n$ is the output of the projection layer $FCL_1$.

During the backward propagation, our far-near loss integrates the losses from the two projection layers as in Equation 5. Thus, the neural network is supervised simultaneously by the outputs of both $FCL_1$ and $FCL_2$. When a low-frequency path differs from a high-frequency path at the $n$-th step after a shared subsequence, the extra $FCL_1$ would enhance their differences at the hidden state $h_{n-1}$, while maintaining their similarity at other intermediate $h_i$. To the best of our knowledge, none of the existing related solutions uses a combination of token-level loss and sequence-level loss together.

The new far-near loss $l_h$ is defined as:

$$l_h = \alpha L(o_n, x_n) + (1-\alpha)\sum_{i=1}^{n}\frac{L(s_i, x_i)}{n} \quad (5)$$

where $L(\cdot)$ is the cross entropy loss between the output and label. We set the weight $\alpha$ to be 0.5 in our experiments.

When low-frequency sequences were initially predicted wrong due to its misleading high-frequency suffix, HyLSTM produces a larger loss than regular LSTM to correct it and treat the beginning tokens more seriously in our evaluation. Besides evaluating HyLSTM against regular LSTM models, we also compare it with BERT in Section 4.

### 3.3.2 Our multi-path architecture

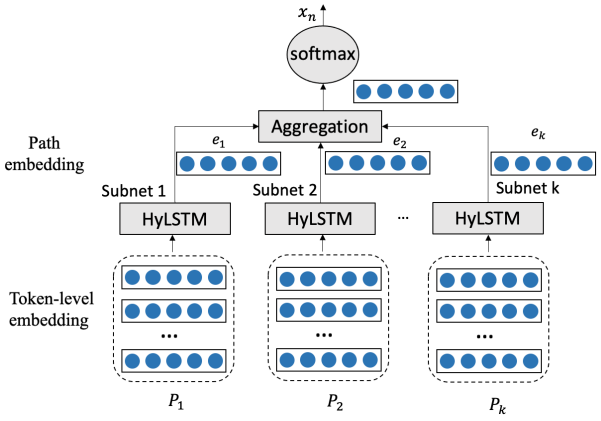



Fig. 6. Multi-path code suggestion architecture based on aggregating path embeddings

We further design a new multi-path architecture to incorporate parallel sequence modeling, aiming to address the multi-path dependence challenge. As shown in Figure 6, this architecture includes several identical subnets, each of which is a model, HyLSTM. These parallel subnets are followed by an aggregation layer to output the API suggestion. Compared with the graph-based neural networks, our multi-path architecture is not limited by the graph size, thus can achieve better efficiency and scalability.

The training of our Multi-HyLSTM includes two phases: single-path pretraining, and multi-path finetuning. To better calculate the conditional probability $p(x_n|P_i)$, we pretrain the subnet (i.e., HyLSTM) on all single paths. Every token of a path is represented as the word2vec-like embedding trained on the dependence path corpus. The output layer of HyLSTM before the softmax activation $e_i$ represents the embedding of the entire path.

$$e_i = h_{n-1}W_1 + B_1 \quad (6)$$

where $h_{n-1}$, $W_1$, and $B_1$ are the last timestep's hidden state of HyLSTM, the weights and bias of $FCL_1$, respectively.

The pretrained HyLSTM is used as the initial state of Multi-HyLSTM. We add an average pooling layer to aggregate these path embeddings $e_i$ into one vector. The softmax classifier is applied to generate the conditional probability:

$$p(x_n|C) = softmax(\sum_{i=1}^{k} e_i/k) \quad (7)$$

where $C$ is the program context, $x_n$ is the predicted API call, and $k$ is the number of input paths.

Under the multi-path architecture, HyLSTM is jointly updated with the following layers towards the task-specific distribution. The multi-path architecture successfully highlights the minor difference in the process of code suggestion. Candidate tokens having similar probabilities in some paths can now be accurately differentiated. We evaluated this multi-path architecture on the sequence model BERT and HyLSTM in Section 4.2.2.

## 4 Experimental Evaluation

We conduct API completion experiments on Java cryptographic code to compare the top-1 recommendation accuracy of our Multi-HyLSTM and alternative approaches.

*Dataset and baselines.* We collect 64,478 Android apps covering 21 categories from Google Play Store to form our dataset. We filter these Android apps with a state-of-the-art code screening tool CryptoGuard [31] and identify 774,460 Java cryptographic API method callsites that are used properly in codebase. These Android apps are processed with our program analysis at each cryptographic API callsite location to extract their dependence contexts, – the API dependence paths.

We compare our Multi-HyLSTM with two types of baselines on this dataset. First, we compare Multi-HyLSTM with two state-of-the-art API completion tools SLANG [26] and Codota [25]. SLANG is an academic API completion solution that combines static analysis and statistical language models to generate API method recommendations. We reproduce the static analysis preprocessing and neural network training of SLANG. Due to the long preprocessing time of SLANG, we conduct experiments on a subset of our entire dataset, 16,048 apps from 3 app categories (Business, Finance, and Communication). Overall, there are 36,029 cryptographic API callsites identified (see Figure 7). We train SLANG and our Multi-HyLSTM on the same data for comparison. Moreover, we compare the trained Multi-HyLSTM with a published commercial API completion tool Codota [25]. Codota can be used as a plugin in most of the mainstream IDEs. We manually evaluate Codota plugin in the IntelliJ IDE on 245 cryptographic API callsites (see Table 1) collected from 9 randomly selected Android apps. Our Multi-HyLSTM is also evaluated on these 245 test cases. Second, we conducted an ablation study (see Table 5) on the 774,460 cryptographic API method calls extracted from 107,282 Android apps. The ablation study evaluates our neural network design. The Multi-HyLSTM is compared with the intermediate solutions that remove either of our design choices.

We also tried to compare with the code completion solution BAYOU [16], NSG [24] and the large-scale pretrained programming language model CODEGPT [8]. However, the code of BAYOU and NSG cannot be successfully replicated. The CODEGPT that is pretrained on the subtoken level is difficult to be finetuned for our task, because it is hard to know which subtokens in its output sequence correspond to our target API method.

For all the training experiments, we randomly select 1/5 of the data as the test cases and train the baselines and our model with the other 4/5 cases. We train these models for 10 epochs with batch size 1,024. We record the highest accuracy the model achieves within 10 epochs. Although our baselines (e.g. SLANG, Codota) are not designed for cryptographic APIs, we think their data-driven approach should make them generalize to our domain specific data.

### 4.1 Comparison with Existing Tools

We compare our Multi-HyLSTM with two state-of-the-art API completion tools, SLANG [26] and Codota [25]. We show the average top-1 accuracy across all testcases as defined in Equation 8, where $n$ is the number of testcases and the binary value $accuracy_i$ is the top-1 accuracy of case$_i$ (1 for correct and 0 for incorrect).

$$\text{Avg\_accuracy} = \frac{\sum_{i=1}^{n} accuracy_i}{n} \tag{8}$$

**Comparison with SLANG.** SLANG uses a different program analysis preprocessing to extract the context (named object histories) for prediction. It combines the $n$-gram and RNN models to generate the probability of the next API method call. Figure 7 shows the top-1 accuracy of SLANG and our approach. We choose the hidden layer size of SLANG and our approach from 128, 256, and 512. With each hidden layer size setting, we also adjust SLANG with 3-, 4-, 5-gram model [2]. Our approach shows significant advantages over SLANG in all settings. The highest top-1 accuracy of SLANG is 77.44%, achieved with RNN-256. The $n$-gram model shows no impact on the top-1 accuracy. Our models achieve the best accuracy at 91.41% under Multi-HyLSTM with hidden layer size 512, achieving an improvement of 18% compared with SLANG.

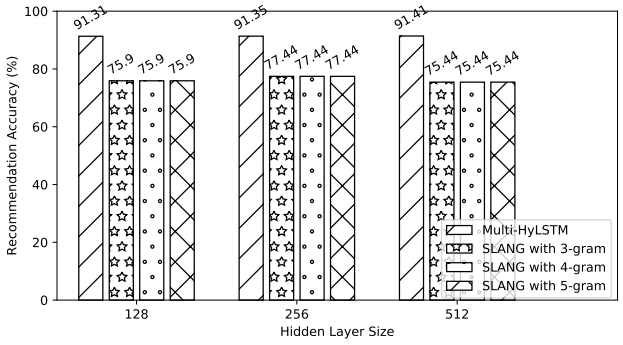



Fig. 7. The top-1 accuracy of SLANG and our approach on API completion

**Comparison with Codota.** Codota is a commercial AI code completion plugin, which is adopted by mainstream IDEs including IntelliJ, Eclipse, Android Studio, VS Code, etc. Given an incomplete code statement with an object with a `dot` in an IDE, Codota displays a ranked list of the recommended API methods associated with the object. We randomly selected 245 cryptographic API method invocations from 9 Android applications as the test cases. We decompiled the 9 apps into source code and load them into IntelliJ IDE with Codota. Then, we manually triggered Codota recommendation by

[2]Raychev et al choose 3-gram and hidden layer size 40 for RNN in [26]

TABLE 1
The top-1 accuracy of Codota and our Multi-HyLSTM on 245 randomly selected Java Cryptographic API invocation test cases. Codota gives recommendations based on the previous code and the return value. We show our accuracy under both conditions (i.e., w/o and with the return value).

| App Category | # of Test Cases | # of apps | Codota Accuracy | Multi-HyLSTM (Our approach) Accuracy | |
|---|---|---|---|---|---|
| | | | | w/o return value | with return value |
| Business | 66 | 3 | 66.67% | 89.39% | 98.49% |
| Finance | 99 | 3 | 65.66% | 90.91% | 97.98% |
| Communication | 80 | 3 | 62.5% | 86.25% | 97.59% |
| **Total** | **245** | **9** | **64.9%** | **88.98%** | **97.96%** |

removing the method name after the `dot`. Table 1 shows our approach has a significant accuracy improvement compared with Codota. The top-1 accuracy of the 245 cases is improved from 64.90% to 88.98%. Note that codota takes not only the previous context but also the return value type to decide the recommendation, whereas our approach only relies on the previous code. Thus, we further measure our accuracy if the return type is specified. We manually checked the return type to filter out the incompatible candidates. Our top-1 accuracy (last column in Table 1) rises to 97.96% if the return type is given, resulting in 51% improvement.

### 4.2 Ablation Study

We conduct an ablation study to evaluate the effectiveness of the two design components of our neural network, HyLSTM, and the multi-path architecture. All of these neural networks work with identical program analysis preprocessing that extracts the API dependence paths as the neural network inputs. We noticed that many dependence paths may have multiple correct choices for the next API method calls. This could happen when there are branches in the API dependence graphs. All of them are regarded as correct answers when counting the accuracy. Therefore, we introduce a new metric, referred to as *in-set accuracy* for this ablation study. We define in-set accuracy as the accuracy of top-1 recommendations that fall in a reasonable next API method set. The reasonable next API method set is collected based on all the situations that ever happen in our collected API dependence graphs.

#### 4.2.1 HyLSTM vs. LSTM

To evaluate the effectiveness of our HyLSTM, we compare it with two regular LSTM models, the LSTM sequence model trained with the token-level loss and the LSTM sequence model trained with the sequence-level loss. The token-level loss only considers the output at the last timestep, while the sequence-level loss is the loss calculated on the entire sequence, including the recurrent outputs at every timestep during training. As shown in Figure 5, HyLSTM has two parallel projection layers. One produces the token-level loss and the other produces the sequence-level loss. HyLSTM uses our new far-near loss combining both of them. The three models use identical LSTM cells with a hidden layer size of 256.

Table 2 shows the average *in-set accuracy* of our HyLSTM and the two regular LSTM models. Overall, HyLSTM achieves the best in-set accuracy at 93% compared with two LSTM models. We further analyze the capabilities of the three models by breaking down the test cases into two groups,

TABLE 2
The average in-set accuracy of HyLSTM for the next API recommendation. Acc.(U) is the in-set accuracy for the test cases with unknown features. Acc.(K) is the in-set accuracy for the test cases with known features. Acc.(A) refers to the in-set accuracy for all the test cases.

| | HyLSTM | LSTM (token-level loss) | LSTM (sequence-level loss) |
|---|---|---|---|
| Acc.(U) | 56.94% | 43.21% | **57.13%** |
| Acc.(K) | **99.86%** | 99.81% | 96.98% |
| Acc.(A) | **93.00%** | 90.77% | 90.62% |

the test cases with unknown features and the test cases with known features. The features refer to the API dependence paths we extracted from the Android apps. The test cases with known features mean their extracted input features (i.e., API dependence paths) are identical to the extracted features of certain cases in the training phase. The test cases with unknown features suggest that there are new dependence paths that never appear in the training code corpus.

It is more challenging for a neural network to handle unknown dependence paths. From Table 2, we observe HyLSTM outperforms the LSTM with token-level loss for unknown test cases. HyLSTM substantially improves the accuracy from 43.21% to 56.94% – a 31.78% improvement.

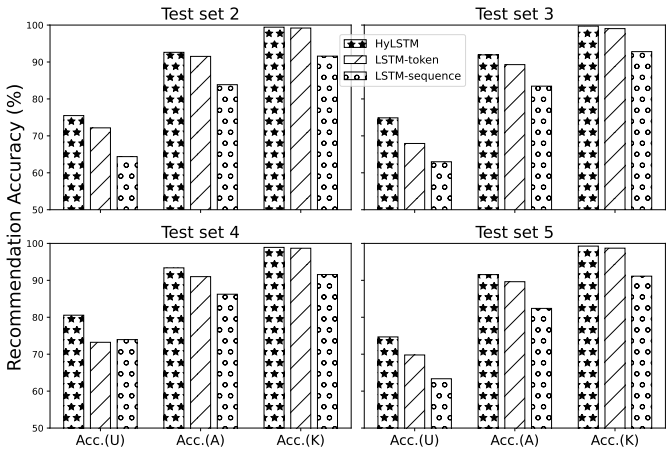



Fig. 8. The average in-set accuracy of HyLSTM and two regular LSTM models tested with four new app sets. LSTM-sequence represents the LSTM model with sequence-level loss, while LSTM-token represents the LSTM model with token-level loss.

**Test on new apps.** The excellent performance on the test cases with known features does not suggest overfitting, because our model can successfully handle test cases from

TABLE 3
Statistics of the test cases with known and unknown features in different test sets. Test set 1 is the original test set used in Table 2. 20% of the API dependence paths extracted from the 16,048 apps are used for testing while the other 80% are used for training. Test sets 2, 3, 4, and 5 are new apps that are never used during training.

| Test set ID | Number of apps | Number of test cases | | New App? |
|---|---|---|---|---|
| | | Known | Unknown | |
| 1 | 16,048 | 92,135 | 17,512 | No |
| 2 | 107 | 3,494 | 1,387 | Yes |
| 3 | 263 | 13,176 | 6,020 | Yes |
| 4 | 454 | 13,111 | 5,669 | Yes |
| 5 | 2,993 | 109,836 | 50,324 | Yes |

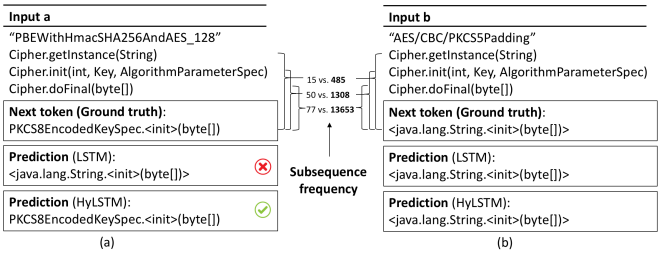



Fig. 9. Case Study 1, (a) A test case that is predicted incorrectly by regular LSTM and correctly by HyLSTM. (b) A test case that follows the frequent short pattern thus got correct prediction by both regular LSTM and our HyLSTM.

new apps. To validate it, we test the three models with new apps that are never used in the training phase. We gather four extra Android app sets from different Android app categories, weather, social, personalization, and 12 other categories mixed together. We use the four app sets to reduce the bias from the app categories. Table 3 shows their statistics as well as the original test set. Compared with the original test set, the percentage of the test cases with unknown features grows in the new test sets. In the original test set (Test set 1), the unknown group accounts for 15.97% of the test cases. In the total of the four new test sets, the test cases with unknown features account for 31.23% of the test cases.

Figure 8 shows the average *in-set accuracy* on the four new test sets. The average results of them are displayed in Table 4. Results show that our HyLSTM outperforms the two regular LSTM models on all four new test sets. On test cases with unknown features (denoted by U), HyLSTM gives an obvious advantage over the LSTM models. This experiment shows that the accuracy for the test cases with unknown features substantially increases when testing with the new apps. Overall, HyLSTM achieves an accuracy of 92.37%, which is substantially higher than the LSTM model with the sequence-level loss and slightly higher than the LSTM with the token-level loss.

TABLE 4
The average in-set accuracy of HyLSTM and the two regular LSTM models tested on four new test sets composed of new Android apps.

| | HyLSTM | LSTM (token-level loss) | LSTM (sequence-level loss) |
|---|---|---|---|
| Acc. (U) | **76.40%** | 70.78% | 66.17% |
| Acc. (K) | **99.34%** | 98.91% | 91.77% |
| Acc. (A) | **92.37%** | 90.36% | 83.99% |

*Case Study 1.* This case verifies that our HyLSTM is better at identifying the global dependence that is low-frequency. Figure 9 (a) shows a test case predicted incorrectly by regular LSTM, but correctly predicted by our HyLSTM. The wrong prediction of LSTM is due to the more frequent shorter patterns in Figure 9 (b). In contrast, HyLSTM successfully differentiates similar inputs (a) and (b).

#### 4.2.2 Multi-path vs. sequential architecture

We compare our multi-path architecture with their single-path counterparts in API completion. In Table 5, we test two versions of the Multi-HyLSTM design. Multi-HyLSTM (avg.) uses the average pooling to aggregate multiple paths embeddings, as described in Sec. 3.3.2, while Multi-HyLSTM (att.) uses an attention-based aggregation. Multi-HyLSTM models are compared with its single-path version HyLSTM, and two other alternative approaches DepBERT and Multi-BERT. DepBERT is the neural network of BERT [36] pre-trained on our dependence paths corpus with the masked language modeling task. Multi-BERT is where DepBERT replaces HyLSTM. To be fair in our comparison, we also pretrained our HyLSTM on the same dataset. All of these pretrained models are finetuned by our API completion task.

*Multi-path vs. Single-path*. Both Multi-HyLSTM and Multi-BERT are more accurate compared with their single-path counterparts. The in-set accuracy is improved from 95.79% of HyLSTM to 98.99% of Multi-HyLSTM, and from 92.49% of DepBERT to 95.78% of Multi-BERT. More importantly, multi-path aggregation gives significant accuracy improvement on unknown cases– by 11.53% for HyLSTM and 36.50% for DepBERT.

*Average Aggregation vs. Attention Aggregation*. We also compare the average aggregation approach with the more complicated attention-based aggregation in Multi-HyLSTM. In attention-based aggregation, we replace the averaged path embeddings $\sum_{i=1}^{k} e_i/k$ in equation (7) with $\sum_{i=1}^{k} w_i e_i$, where weight $w_i$ is the attention learned through training. Results in Table 5 show no significant benefit of the attention mechanism. For the unknown cases, the average aggregation achieves a slightly higher accuracy of 83.02% compared with 80.71% of the attention aggregation. Their overall accuracy is similar.

*Improvement from path embedding*. The single-path pretraining can benefit accuracy, especially for unknown cases. Compared with the basic HyLSTM in Table 2, HyLSTM with the extra path embedding improves the in-set accuracy by 30.74% for unknown cases.

*HyLSTM vs. DepBERT*. HyLSTM is better at API completion compared with DepBERT. HyLSTM increases the in-set accuracy by 33.57% for unknown cases from DepBERT. This can be attributed to our specialized global dependence enhancing learning. With the far-near loss, HyLSTM is forced to pay more attention to the long but low-frequency API sequence even if the more frequent shorter dependence exists.

*Case Study 2.* Figure 10 demonstrates how the multi-

TABLE 5
Comparison between multi-dependence suggestion and sequential suggestion. U, K, and A stand for the average in-set accuracy for unknown cases, known cases, and all cases, respectively.

| | Multi-HyLSTM (avg.) | Multi-HyLSTM (att.) | HyLSTM (path embedding) | DepBERT | Multi-BERT |
|---|---|---|---|---|---|
| Acc.(U) | **83.02%** | 80.71% | 74.44% | 55.73% | 76.07% |
| Acc.(K) | 99.59% | 99.74% | **99.84%** | 99.48% | 96.52% |
| Acc.(A) | 98.99% | **99.06%** | 95.79% | 92.49% | 95.78% |

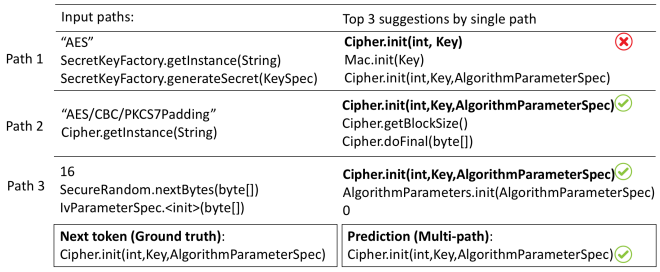



Fig. 10. Case Study 2. A test case that needs multiple paths to predict correctly. The wrong prediction suggested by Path 1 can be fixed after aggregating the influences from two extra paths.

path model improves over the single path model. The label `Cipher.init(int,Key,AlgorithmParameterSpec)` and `Cipher.init(int,Key)` are indistinguishable, given dependence path 1. Fortunately, paths 2 and 3 provide complementary information to correct it. To determine the usage of API method `Cipher.init(int,Key,AlgorithmParameterSpec)` instead of `Cipher.init(int,Key)`, the decisive dependence is the API call `IvParameterSpec.<init>(byte[])` (in path 3) where `IvParameterSpec` is a subtype of `AlgorithmParameterSpec`. Path 2 also provides an additional indicator `"AES/CBC/PKCS7Padding"` considering the cryptographic knowledge that AES-CBC mode requires initial vectors for security. The single-path model (e.g., depBERT, HyLSTM) cannot make the correct prediction when the informative dependencies are not covered.

### 4.3 Cross-app Evaluation

We further conduct cross-app experiments to confirm Multi-HyLSTMs prediction capability on new apps, i.e., never appearing in the training phase. We use the four datasets extracted from 12 categories, namely social, weather, personalization, and other categories mixed together. The test sets contain up to 50% unknown cases (Table 6). Among the four datasets from different categories, Multi-HyLSTM achieved up to 99.61% in-set accuracy with an average of 98.69% (Table 7). Multi-HyLSTM also showed good performance when only considering the unknown cases, attaining 95.28% to 99.21% in-set accuracy across the four datasets. One possible reason for the excellent performance could be that, even if a combination of multiple paths has never appeared in the training set, the model learned some parts of these paths during training and is able to make correct predictions using this knowledge. These results prove that Multi-HyLSTM is not overfitted to known apps from the training set and is able to handle unknown cases from new apps, suggesting good performance in realistic settings.

TABLE 6
Number of test cases in the four test sets used in the cross-app experiments.

| App category | Number of test cases | Number of known cases | Number of unknown cases |
|---|---|---|---|
| Social | 7,588 | 3,941 | 3,647 |
| Weather | 1,989 | 993 | 996 |
| Personalization | 8,260 | 5,356 | 2,914 |
| Other categories | 64,732 | 41,339 | 23,393 |

TABLE 7
The average in-set accuracy of the Multi-HyLSTM model in cross-app settings, where training and testing cases are selected from different apps. U, K, and A stand for unknown cases, known cases, and all cases, respectively.

| | Social | Weather | Personalization | Other categories |
|---|---|---|---|---|
| Acc. (U) | 97.23% | 95.28% | 99.21% | 97.32% |
| Acc. (K) | 99.95% | 100% | 99.83% | 99.73% |
| Acc. (A) | 98.64% | 97.64% | 99.61% | 98.86% |

### 4.4 Examples of Sequence Frequencies

We briefly show two example sequences to illustrate the importance of our HyLSTM design and the new far-near loss function for countering high frequency influences. In Figure 11, LSTM predicts this case wrong. The wrong prediction comes from a subsequence that is much more frequent (5,891 times) than the correct token (4 times). In Figure 12, LSTMs prediction String.getBytes(String) is wrong. We found that the 3-gram subsequence (StringBuilder.append(String), StringBuilder.toString(), String.getBytes(String)) is extremely frequent (70,244 times) in the training corpus. Our HyLSTM model predicts both cases correctly.

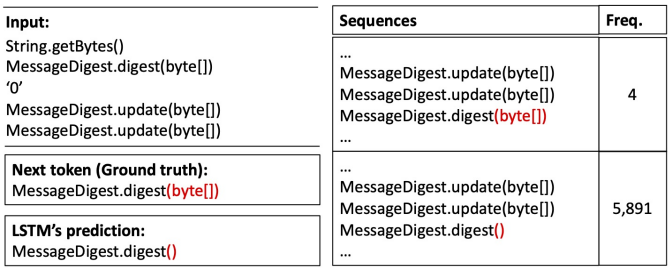



Fig. 11. An example illustrating that LSTM's wrong prediction is influenced by the high-frequency sequence.

**We summarize our experimental findings as follows:**

- Our Multi-HyLSTM substantially outperforms the state-of-the-art academic API completion solution SLANG

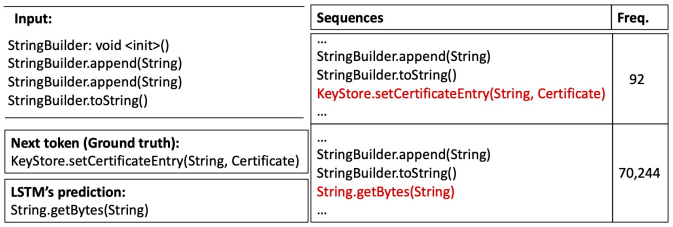



Fig. 12. Another example illustrating the strong disparity in frequency.

and commercial solution Codota. Multi-HyLSTM achieves an excellent top-1 accuracy of 91.41%, a 18.04% improvement over SLANG with the best accuracy of 77.44%. In a manual analysis of 245 test cases compared with Codota, Multi-HyLSTM achieves the top-1 accuracy at 97.96%, a 50.94% improvement over Codota with an accuracy of 64.9%.

- Our multi-path architecture excels at recognizing unseen cases. Multi-HyLSTM and Multi-BERT achieve the in-set accuracy for unknown cases of 83.02% and 76.07%, improving their single-path counterparts HyLSTM (74.44%) and DepBERT (55.73%) by 11.53% and 36.50%, respectively.
- Our HyLSTM outperforms two regular LSTM models. It improves the inference capability of the LSTM with token-level loss by 31.78%.

*Performance and runtime.* With the distributed training of 8 workers, our training time is significantly improved. Most of our experiments are completed within 5 hours.

**Limitations.** First, many static analyses overestimate execution paths. Thus, some extracted dependence paths might not necessarily occur, which may lead to a wrong prediction. However, since our approach relies on multiple paths, we expect the deep learning model to automatically learn which path to use by training. Second, the extracted dependence paths may be incomplete, as we omit recursions in the graph. We also terminate the path when the depth of call stacks is beyond 10. However, a previous study experimentally showed the impact of limited depth exploration to be negligible in practice [31]. Another limitation is that there might be difficult to apply static analysis on incomplete source code that the code developers are writing in IDEs. The real-world application scenario requires enabling the partial program analysis that can work with the incomplete source code.

## 5 Related Work

We summarize the related work based on the program representation strategies.

*Treating programs as text.* Many studies [7], [37], [38], [39], [40] treat programs as code sequences. Programs are tokenized into source code token sequences and modeled like textual sentences. The giga-token models are built by applying $n$-gram model [37] or more powerful network models (e.g., LSTM, Transformers, GPT-2) [39] on them. However, the out-of-vocabulary (OOV) issue in program token sequences is much more severe than in natural languages and requires advanced tokenization techniques to address [7], [40].

*Extracting syntactic information as context.* When treating programs as text, syntactical errors are common. Therefore, abstract syntax trees (ASTs) and probabilistic context free grammar (PCFG) are widely adopted to enforce the syntax correctness [41], [42]. However, PCFG is found insufficient due to the limited context coverage of ASTs. Bielik *et al.* [43] extend PCFG to probabilistic higher order grammar (PHOG) by enriching its context. Another direction is to use more powerful neural networks that can automatically identify significant dependencies from longer contexts [44], [45].

*Extracting semantic information as context.* To generate code following the program semantics, a couple of studies [12], [17], [21], [46] represent programs as graphs. For example, Allamanis [12] build a graph that uses AST as the backbone and adds different types of edges according to their dataflows. However, compared with ours, the graph based approaches are highly limited by the graph size. Besides graphs, grammar-based production rules (e.g., attribute grammar (AG) [47]) are incorporated to guide the generation process on graphs or program sketches [17], [24].

Then, according to different completion targets, we summarize the code completion work as follows.

*Completing API methods.* Some studies focus on the completion of invoked API methods to improve the productivity of developers and solve API related problems [26], [48], [49], [50]. Program analysis techniques are often applied to extract API sequences from source code to build language models. Nguyen *et al.* presented a graph representation of the object usage model (GORUM) to represent interactions between different objects and associated methods [51]. They built Hidden Markov models for the state of objects and predict methods [48], [50]. However, these methods may require building endless Markov models for different object types. Raychev *et al.* [26] built RNN and n-gram models on top of the object histories defined by themselves for API method recommendation. The object histories consist of the method call events in the temporal order. Although its top-16 accuracy (96.43%) is pretty good, it only achieves a top-1 accuracy of 69.05%.

*Completing variable names.* There are a couple of solutions that target to complete the correct names for variables in codebase [12], [52], [53], [54]. Allamanis *et al.* defined two tasks VarNaming [54] and VarMisuse [12] that focus on completing a code snippet with a "hole" at the location of a variable. In their approaches, other variables in the local context are extracted as candidates. These candidates are ranked with statistical language modeling combined with program analysis focusing on the variable definition and usage.

*Completing general tokens.* Some studies treated different functional tokens (e.g., variables, API calls, etc.) identically and aim to generate an entire code block or function by continuously generating the next tokens [6], [7], [14], [55]. These approaches often rely on large language models [3] pretrained with huge amounts of online code. In the code generating process, optimized search strategies, such as beam search, are often used to dynamically rank the growing sequence candidates. However, the generated sequences are usually evaluated with the BLEU score [56] that is designed to measure the similarity of two natural language sequences.

This might be problematic since the correctness of the code sequence is not guaranteed [57].

## 6 Conclusions

Data-driven code suggestion approaches need to be deeply integrated with program-specific techniques, as code and natural languages have fundamental differences. We proposed new neural network based API completion techniques to capture program dependencies. We compared our approach with the state-of-the-art API completion tools and conducted extensive studies to evaluate the effectiveness of our two design choices, the multi-path architecture and global dependence enhancing learning. Our results confirmed that our approach is effective at capturing the program dependencies for API completion tasks. Our future work will focus on enabling real-world code completion applications to help developers in real-time. Towards this direction, the static analysis needs to be available on incomplete code and the neural network inferences to be efficient to meet latency or throughput requirements.

## Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. CNS-1929701 and Virginia Commonwealth Cyber Initiative (CCI).

## References

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[2] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[4] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "XLNET: Generalized autoregressive pretraining for language understanding," *Advances in neural information processing systems*, vol. 32, 2019.

[5] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-XL: Attentive language models beyond a fixed-length context," *arXiv preprint arXiv:1901.02860*, 2019.

[6] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[7] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "IntelliCode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[8] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[9] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[10] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[11] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 428–439.

[12] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations (ICLR)*, 2018.

[13] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago *et al.*, "Competition-level code generation with alphacode," *arXiv preprint arXiv:2203.07814*, 2022.

[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[15] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *arXiv preprint arXiv:1802.03691*, 2018.

[16] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural sketch learning for conditional program generation," *International Conference on Learning Representations (ICLR)*, 2018.

[17] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," *International Conference on Learning Representations (ICLR)*, 2019.

[18] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow," Mar. 2021, If you use this software, please cite it using these metadata. [Online]. Available: https://doi.org/10.5281/zenodo.5297715

[19] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," *arXiv preprint arXiv:1711.09573*, 2017.

[20] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, "A self-attentional neural architecture for code completion with multi-task learning," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 37–47.

[21] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "GraphCodeBERT: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[22] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 10 197–10 207.

[23] M. Allamanis, P. Chanthirasegaran, P. Kohli, and C. Sutton, "Learning continuous semantic representations of symbolic expressions," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 80–88.

[24] R. Mukherjee, Y. Wen, D. Chaudhari, T. Reps, S. Chaudhuri, and C. Jermaine, "Neural program generation modulo static analysis," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[25] "Codota AI Autocomplete for Java and JavaScript," https://plugins.jetbrains.com/plugin/7638-codota-ai-autocomplete-for-java\-and-javascript, 2021.

[26] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *ACM Sigplan Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[27] S. Nadi, S. Kruger, M. Mezini, and E. Bodden, "Jumping through hoops: Why do Java developers struggle with cryptography APIs?" in *In Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2016.

[28] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You get where you're looking for: The impact of information sources on code security," in *In 2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 289 – 305.

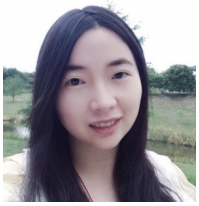
[29] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. Arango-Argoty, "Secure coding practices in Java: Challenges and vulnerabilities," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 372–383.

[30] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic APIs," in *In 2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 154 – 171.

[31] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao, "CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 2455–2472.

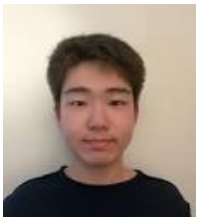
[32] Y. Xiao, Y. Zhao, N. Allen, N. Keynes, D. D. Yao, and C. Cifuentes, "Industrial experience of finding cryptographic vulnerabilities in large-scale codebases," *ACM Digital Threats: Research and Practice (DTRAP)*, 2022.

[33] M. Schlichtig, A.-K. Wickert, S. Krüger, E. Bodden, and M. Mezini, "CamBench - cryptographic API misuse detection tool benchmark suite," 2022, available at https://arxiv.org/pdf/2204.06447.pdf.
[34] S. Afrose, S. Rahaman, and D. Yao, "CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses," in *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 2019, pp. 49–61.
[35] S. Afrose, Y. Xiao, S. Rahaman, B. P. Miller, and D. D. Yao, "Evaluation of static vulnerability detection tools with Java cryptographic API benchmarks," *IEEE Transactions on Software Engineering (TSE)*, 2022.
[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.
[37] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 207–216.
[38] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: Improving error reporting with language models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 252–261.
[39] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," *arXiv preprint arXiv:1608.02715*, 2016.
[40] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1073–1085.
[41] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *International Conference on Machine Learning*. PMLR, 2014, pp. 649–657.
[42] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," *ACM Sigplan Notices*, vol. 51, no. 1, pp. 761–774, 2016.
[43] P. Bielik, V. Raychev, and M. Vechev, "PHOG: Probabilistic model for code," in *International Conference on Machine Learning*. PMLR, 2016, pp. 2933–2942.
[44] C. Liu, X. Wang, R. Shin, J. E. Gonzalez, and D. Song, "Neural code completion," 2016.
[45] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *arXiv preprint arXiv:1704.01696*, 2017.
[46] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 858–868.
[47] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968.
[48] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 532–542.
[49] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "API code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 511–522.
[50] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning API usages from bytecode: A statistical approach," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 416–427.
[51] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, 2009, pp. 383–392.
[52] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from " big code"," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 111–124, 2015.
[53] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 38–49.
[54] ——, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 281–293.
[55] C. B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "PyMT5: Multi-mode translation of natural language and Python code with transformers," *arXiv preprint arXiv:2010.03150*, 2020.
[56] 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.
[57] 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.



**Ya Xiao** Dr. Ya Xiao received her Ph.D. degree in Computer Science from Virginia Tech. Her research lies in deep learning applications in programming language, software security, and cryptography, including code embedding, learning-based automatic vulnerabilities detection and repair, neural cryptanalysis. She earned her bachelor and master degrees from Beijing University of Posts and Telecommunications in China.

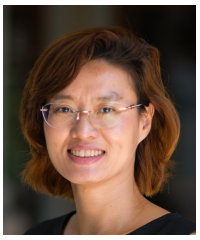
**Wenjia Song** Wenjia Song is a Ph.D. student in Computer Science at Virginia Tech. Her research falls on applications of machine learning in medical predictions and cybersecurity. More specifically, in medical field, her focus is on bias detection and correction of machine learning models on clinical datasets. In cybersecurity, her focus is on the detection and analysis of advanced attack behaviors.



**Jingyuan Qi** Jingyuan Qi is currently a Ph.D. student at Virginia Tech. His research interests include natural language processing (NLP) and multimodal learning. He received his B.S. from Virginia Tech, Blacksburg, USA, in 2020.

**Bimal Viswanath** Dr. Bimal Viswanath is an Assistant Professor of Computer Science at Virginia Tech. His research interests are in security. His ongoing work investigates machine learning systems through the lens of security. He uses data-driven methods to understand new threats raised by advances in machine learning, and also investigates how machine learning can improve security of online services. He obtained his PhD from the Max Planck Institute for Software Systems, and MS from IIT Madras. He also worked as a Researcher at Nokia Bell Labs before starting an academic position.

**Patrick McDaniel** Dr. Patrick McDaniel is the Tsun-Ming Shih Professor of Computer Sciences in the School of Computer, Data and Information Sciences at the University of Wisconsin-Madison. Professor McDaniel is a Fellow of IEEE, ACM and AAAS, a recipient of the SIGOPS Hall of Fame Award and SIGSAC Outstanding Innovation Award, and the director of the NSF Frontier Center for Trustworthy Machine Learning. He also served as the program manager and lead scientist for the Army Research Laboratory's Cyber-Security Collaborative Research Alliance from 2013 to 2018. Patrick's research focuses on a wide range of topics in computer and network security and technical public policy. Prior to joining Wisconsin in 2022, he was the William L. Weiss Professor of Information and Communications Technology and Director of the Institute for Networking and Security Research at Pennsylvania State University.



**Danfeng (Daphne) Yao** Dr. Danfeng (Daphne) Yao is a Professor of Computer Science at Virginia Tech. She is an Elizabeth and James E. Turner Jr. '56 Faculty Fellow and CACI Faculty Fellow. Her research interests include building cyber defenses, as well as machine learning for digital health, with a shared focus on accuracy and deployment. She creates new models, algorithms, techniques, and deployment-quality tools for securing large-scale software and systems. Her tool CryptoGuard helps large software companies and Apache projects harden their cryptographic code. She systematized program anomaly detection in the book Anomaly Detection as a Service. Her patents on anomaly detection are extremely influential in the industry, cited by patents from major cybersecurity firms and technology companies, including FireEye, Symantec, Qualcomm, Cisco, IBM, SAP, Boeing, and Palo Alto Networks. Dr. Yao is an IEEE Fellow for her contributions to enterprise data security and high-precision vulnerability screening. She received her Ph.D. degree from Brown University (Computer Science), M.S. degrees from Princeton University (Chemistry) and Indiana University (Computer Science), Bloomington, B.S. degree from Peking University in China (Chemistry).

## APPENDIX

We provide the pseudo code for our multi-path selection algorithm.

**Algorithm 1** MultiPathSelection($k$, $G$, $s$): Select $i$ ($i \leq k$) paths originating from the $s$, with the constraint of being as non-overlapping as possible

Input: ($k$, $G$, $s$), where $k$ is the path budget, $G$ is an API dependence graph, and $s$ is the starting node in $G$. $f_{ab}$ denotes the data fact flowing from node $a$ to node $b$}
Output: $C$, where $C$ includes $i$ data-flow paths ($i \leq k$).}
let $Q$ be a queue
$Q$.enqueue($s$)
mark $s$ as visited
**while** $Q$ is not empty and Q.length+C.length $<$ n **do**
    $n$ = $Q$.dequeue()
    **if** $n$ has no predecessor **then**
        Collect the path from $s$ to $n$ into $C$
    **end if**
    **for** all predecessor $p$ of $n$ in Graph $G$ **do**
        **if** $p$ is not visited and $f_{pn}$ is not recorded **then**
            $Q$.enqueue($p$)
            mark $p$ is visited, $f_{pn}$ is recorded
            **if** $Q$.length +$C$.length == n **then**
                break
            **end if**
        **end if**
    **end for**
**end while**
**for** all node $n$ in $Q$ **do**
    **while** $n$ has predecessors **do**
        Select a predecessor $p$ of $n$ randomly
        $n = p$
    **end while**
    collect a path from $s$ to $n$ into $C$
**end for**
**return**