Poster: Comparing Neural Network Solutions in Cryptographic API Completion

Ya Xiao*, Salman Ahmed*, Wenjia Song*, Bimal Viswanath*, Na Meng*, Patrick McDaniel†, Xinyang Ge‡, and Danfeng (Daphne) Yao*
Department of Computer Science, Virginia Tech, Blacksburg, VA*
Department of Computer Science and Engineering, Penn State University, State College, PA†
Microsoft Research, Redmond, WA‡
{yax99,ahmedms,wenjia7,vbimal,nm8247,danfeng}@vt.edu, mcdaniel@cse.psu.edu, aegiryy@gmail.com

***Abstract*—With the strong interest in neural network based software engineering approaches and a plethora of proposed solutions, we point out the need for measurement studies in this space. Focusing on a specific application scenario, Java cryptographic API code completion, we outline several potential measurement problems, ranging from embedding design and evaluation, to methodology development of models, and to metrics and benchmarks.**

I. Introduction

The attractive vision of automatic code engineering, e.g., repair [5], [8] and generation [11], [1], has motivated a line of neural networks based machine learning solutions [4], [6]. Given the tremendous success in natural language processing, it is conceivable that deep learning has the potential to revolutionize how code is generalized, transformed, and patched.

In this project, we focus on a specific application scenario, Cryptographic API completion. Cryptographic APIs are reported to be error-prone and result in many security vulnerabilities that seriously threaten software security [10]. We systematically measured the accuracy impacts of the state-of-the-art neural network solutions for cryptographic API completion [12]. The neural network solutions include two key steps, representing programs as numeric vectors, and training a neural network on these vectors. Therefore, our experiments compare different choices for the vectorization, aka code embedding, and the neural networks. We further performed in-depth manual analysis to uncover the unreported challenges from programming language-specific properties.

Comparisons of program analysis guided embeddings. For neural network based approaches, programs need to be first represented as vectors to feed into neural networks. Code embedding refers to the process of automatically learning the low-dimensional vector representations of program elements [2], [7]. Intuitively, it is about how to meaningfully express code in vectors. This transformation is important, as subsequent tasks are performed on the embeddings of code.

Despite recent progress [2], [7], [4], there has not been any systematic investigation of various code embedding designs or comprehensive evaluation in terms of their security and accuracy capabilities. Such side-by-side comparisons would help better design neural network based methodologies and harness their power for code embedding approaches.

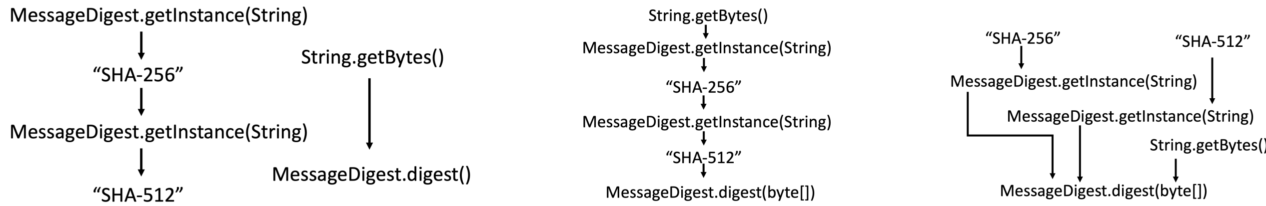
We conducted a comprehensive comparison to learn the impacts of program analysis guidance on the quality of code embedding. By applying program analysis preprocessing, the code sequence can be transformed into more structural representations. These structural representations can provide more *meaningful* context information for code embedding. As shown in Fig. 1, (a) shows the API sequences extracted from byte code while (b) and (c) display API sequences of program slices and API dependence graphs that are obtained by program analysis, respectively. We apply skip-gram embedding model [9] on the byte code, slices, and dependence paths extracted from the API dependence graphs, respectively, to produce three types of API embeddings, *byte2vec*, *slice2vec*, and *dep2vec*. These embeddings, as well as a basic one-hot vector baseline, are used as the inputs when training LSTM based models for cryptographic API completion tasks.

TABLE I: Accuracy of next API Recommendation.

LSTM Units	Byte Code		Slices		Dependence Paths	
	1-hot	byte2vec	1-hot	slice2vec	1-hot	dep2vec
64	49.78%	48.31%	66.39%	78.91%	86.00%	86.33%
128	53.01%	53.52%	68.51%	80.57%	84.81%	87.75%
256	54.91%	54.59%	70.35%	82.26%	84.57%	91.07%
512	**55.80%**	**55.96%**	**71.78%**	**83.35%**	**86.34 %**	**92.04%**

Table I shows the accuracy of the cryptographic API completion task trained with different embedding settings and LSTM models. We have three comparison groups. First, we compare three types of embeddings, *byte2vec*, *slice2vec*, and *dep2vec*, which are trained on different program analysis preprocessed corpora, byte code, slices, and dependence paths, respectively. Second, we also compare the embedding option with its one-hot baseline on each type of code corpus. Moreover, we compare different sizes of LSTM models in this task. An important observation is that program analysis brings significant benefits, improving the accuracy of cryptographic API completion from 55.96% (with *byte2vec*) to 92.04% (with *dep2vec*). We also found that embedding options, *slice2vec* and *dep2vec* significantly improve the accuracies by 12% and 6%, compared with their one-hot baselines.

Analysis on programming language-specific challenges. Besides program analysis guided embedding, the neural network design is another important aspect of the API completion solution. Although many neural language models (e.g. LSTM,



(a) Two byte code Sequences (b) Slice sequence (c) API dependence graph

Fig. 1: Sequences from slice and dependence graph

BERT [3]) achieved great success for natural languages, we observe that they are still insufficient in the cryptograhpic API completion experiments. Our manual analysis reveals that they have difficulties in dealing with program specific properties. Programming language-specific challenges need to be identified and well addressed when we design neural network models for API completion.

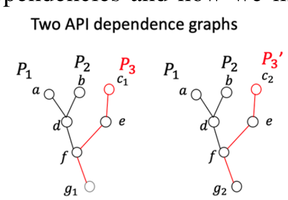
Dependence paths	Frequency
a_1, b, c, d_1	Low
$a_2, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d_2}$	High
$a_3, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d_2}$	
...	
$a_n, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d_2}$	

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

Known	LSTM		HyLSTM (Ours)	
	Loss	Prediction	Loss	Prediction
a_1, b, c	$l_1(b, c, d_1)$	d_2 ✗	$l_1(b, c, d_1) + l_2(d_1)$	d_1 ✓
a_2, b, c	$l_1(b, c, d_2)$	d_2 ✓	$l_1(b, c, d_2) + l_2(d_2)$	d_2 ✓
⋮	⋮	⋮	⋮	⋮
a_n, b, c	$l_1(b, c, d_1)$	d_2 ✓	$l_1(b, c, d_2) + l_2(d_2)$	d_2 ✓

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

Fig. 2: Examples illustrating the challenge of learning global dependencies and how we fix them.



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

When need it to predict g_1

	Known	Prediction
Single-path prediction	P_1 (a, d, f)	g_1 or g_2
	P_2 (b, d, f)	g_1 or g_2
	P_3 (c_1, e, f)	g_1 ✓
Multi-path prediction (Ours)	P_1, P_2, P_3	g_1 ✓

(b) A single path cannot lead to the correct prediction. Multi-path combination achieves the accurate prediction.

Fig. 3: Examples illustrating the challenge of learning multi-path dependencies and how we fix them.

Fig. 2 and Fig. 3 illustrate two unreported programming language-specific challenges from the global dependencies and the multi-path dependencies identified based on our case studies. As shown in Fig. 2, we noticed that an API completion can be decided by an early dependence far away from the current location, referred to as global dependencies. While global dependencies can be captured by program analysis and fed into the neural network, they are very likely to be neglected when appearing in a less-frequent API pattern. Neural networks tend to recognize the shorter but more frequent subsequences, instead of the longer but less frequent ones. Moreover, Fig. 3 (a) demonstrates that two functionally similar APIs that share some identical dependence paths. A sequential model that only relies on a single path often fails to distinguish them. To fix it, we design a model relying on multiple paths.

Comparisons of specialized neural network designs. We present a new neural network Multi-HyLSTM to overcome the programming language-specific challenges. It includes two important features, a global dependence enhancing learning module HyLSTM and a new multi-path architecture. We conducted an ablation study to compare Multi-HyLSTM with two intermediate solutions, HyLSTM, and Multi-BERT, which remove or replace one of our designs. We further compare our model with BERT and LSTM in the same task. The experiments show that our Multi-HyLSTM achieves the best accuracy of cryptographic API completion at 98.99%, showing a boost compared with BERT (92.49%) and LSTM (90.62%).

Acknowledgment

This work has been partly supported by the National Science Foundation under Grant No. CNS-1929701.

References

[1] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 245–256.

[2] 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, p. 40, 2019.

[3] 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.

[4] 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.

[5] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[6] J. Harer, O. Ozdemir, T. Lazovich, C. Reale, R. Russell, L. Kim *et al.*, "Learning to repair software vulnerabilities with generative adversarial networks," in *32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.*, 2018.

[7] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: understanding programs through embedded abstracted symbolic traces," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 163–174.

[8] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[9] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[10] 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.

[11] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 419–428.

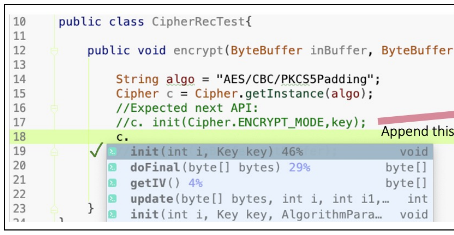
[12] Y. Xiao, S. Ahmed, W. Song, X. Ge, B. Viswanath, and D. Yao, "Embedding code contexts for cryptographic api suggestion: New methodologies and comparisons," *arXiv preprint arXiv:2103.08747*, 2021.

Comparing Neural Network solutions in Cryptographic API Completion[1]

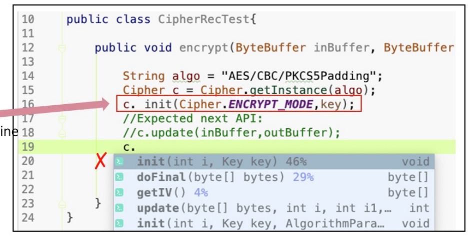
Ya Xiao[1], Salman Ahmed[1], Wenjia Song[1], Bimal Viswanath[1], Na Meng[1], Patrick McDaniel[2], Xinyang Ge[3], Danfeng (Daphne) Yao[1]
[1]Department of Computer Science, Virginia Tech
[2]Department of Computer Science and Engineering, Penn State University
[3]Microsoft Research
{yax99, ahmedms,wenjia7,vbimal, nm8247, danfeng}@vt.edu, mcdaniel@cse.psu.edu, aegiryy@gmail.com

1. Motivation

- *State-of-the-art API Completion is insufficient.*
- *Systematical Comparisons for neural network solutions are necessary.*
- *Programming language specific challenges need to be identified.*



(a) A correct example of Codota completion



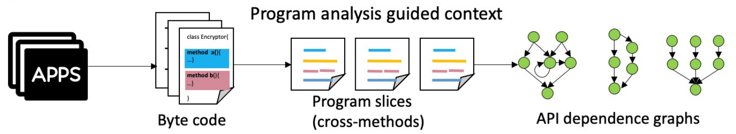
(b) A incorrect example of Codota completion

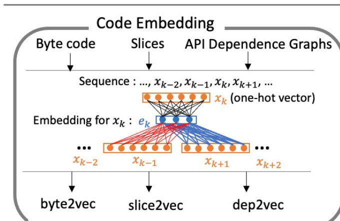
2. Research Questions

What are our research questions of the comparisons?

- *RQ1: How does program analysis guidance influence code embedding?*
- *RQ2: What are the programming language specific challenges?*
- *RQ3: How well do our neural network design choices in cryptographic API completion?*

3. Program Analysis Guided Embedding Approaches





RQ1: How does program analysis guidance influence code embedding?

- We obtain *byte code sequences*, *program slices*, and *API dependence graphs* by applying program analysis.
- We train 3 types of code embeddings, *byte2vec*, *slice2vec*, and *dep2vec* respectively.

4. Programming Language Specific Challenges and Our Designs

Global dependence challenge

Dependence paths	Frequency
a_1, b, c, d_1	Low
$a_2, \mathbf{b}, \mathbf{c}, \mathbf{d_2}$	High
$a_3, \mathbf{b}, \mathbf{c}, \mathbf{d_2}$	
…	
$a_n, \mathbf{b}, \mathbf{c}, \mathbf{d_2}$	

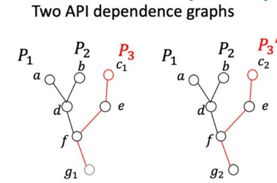
(a) A high-frequency code pattern (b, c, d_2) and its low-frequency variant (a_1, b, c, d_1).

Known	LSTM		HyLSTM (Ours)	
	Loss	Prediction	Loss	Prediction
a_1, b, c	$l_1(b, c, d_1)$	d_2 ✗	$l_1(b, c, d_1) + l_2(d_1)$	d_1 ✓
a_2, b, c	$l_1(b, c, d_2)$	d_2 ✓	$l_1(b, c, d_2) + l_2(d_2)$	d_2 ✓
⋮	⋮	⋮	⋮	⋮
a_n, b, c	$l_1(b, c, d_1)$	d_2 ✓	$l_1(b, c, d_2) + l_2(d_2)$	d_2 ✓

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

Multi-path dependence challenge

Two API dependence graphs



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

When need it to predict g_1

	Known	Prediction
Single-path prediction	$P_1\ (a, d, f)$	g_1 or g_2
	$P_2\ (b, d, f)$	g_1 or g_2
	$P_3\ (c_1, e, f)$	g_1 ✓
Multi-path prediction (Ours)	P_1, P_2, P_3	g_1 ✓

(d) Multi-path combination achieves the accurate prediction

5. Comprehensive Comparisons

Table 1: Accuracy comparison between different embedding settings in cryptographic API Completion.

LSTM Units	Byte Code		Slices		Dependence Paths	
	1-hot	byte2vec	1-hot	slice2vec	1-hot	dep2vec
64	49.78%	48.31%	66.39%	78.91%	86.00%	86.33%
128	53.01%	53.52%	68.51%	80.57%	84.81%	87.75%
256	54.91%	54.59%	70.35%	82.26%	84.57%	91.07%
512	**55.80%**	**55.96%**	**71.78%**	**83.35%**	**86.34 %**	**92.04%**

- With program analysis, *dep2vec* improves the accuracy by 36.10% on average, compared with *byte2vec*.
- With *slice2vec*, the accuracy is improved by 12.02% on average compared with its one-hot baseline.
- With *dep2vec*, the accuracy is improved by 3.97% on average compared with one-hot vectors.

Table 2: Accuracy comparison between our neural network (Multi-HySTLM) and its intermediate baselines in cryptographic API completion. A, K and U stand for accuracy for all cases, known cases, and unknown cases.

	Multi-HyLSTM	HyLSTM (path embedding)	BERT	Multi-BERT
Acc.(A)	**98.99%**	95.79%	92.49%	95.78%
Acc.(K)	99.59%	**99.84%**	99.48%	96.52%
Acc.(U)	**83.02%**	74.44%	55.73%	76.07%

- Our design, Multi-HyLSTM achieves the best accuracy at 98.99%.
- Outperform the state-of-the-art, BERT, by 6.5% accuracy improvement.

Ongoing Work: Publishing an API Completion Plugin and an Evaluation Benchmark

Our ongoing work is to publish an API completion plugin based on our program analysis guided embedding and neural network design. We are also preparing our cryptographic API dataset as an API completion evaluating benchmark.

[1] Ya Xiao, Salman Ahmed, Wenjia Song, Xinyang Ge, Bimal Viswanath, Danfeng (Daphne) Yao. Embedding Code Contexts for Cryptographic API Suggestion: New Methodologies and Comparisons. *arXiv:2103.08747.*

Yao Group on Cyber Security
http://yaogroup.cs.vt.edu

VIRGINIA TECH.