Neural Network-based Methodologies for Securing Cryptographic Code

Ya Xiao

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Danfeng (Daphne) Yao, Chair
Naren Ramakrishnan
Matthew Hicks
Patrick Drew McDaniel
Xinyang Ge

July 26th, 2022
Blacksburg, Virginia

Keywords: Code embedding, Code suggestion, Neural networks, Cryptographic APIs, etc.

Copyright 2022, Ya Xiao

Neural Network-based Methodologies for Securing Cryptographic Code

Ya Xiao

(ABSTRACT)

Many studies show that manual code generation is error-prone and results in vulnerabilities. Vulnerability fixing has been shown as the most time-consuming process among multiple steps of code repair. To help developers repair these security vulnerabilities, my dissertation aims to develop an automatic or semi-automatic secure code generation system with neural network based approaches. Trained with huge amounts of good-quality code, I expect the neural network to learn the secure usage and produce the correct code suggestions.

Despite the great success of neural networks, the vision of comprehending and generating programming languages through neural networks has not been fully realized. There are many fundamental questions that need to be answered. These questions include *1)* what are the accuracy impacts of the various choices in code embedding? *2)* How to address the accuracy challenges caused by the programming language specific properties in the task of secure code suggestion? My dissertation work answers the two questions with a systematical measurement study and specialized neural network designs. My experiments show that program analysis is a necessary preprocessing step to guide the code embedding – resulting in a 36.2% accuracy improvement. Furthermore, I identify two previously unreported deficiencies in the cryptographic API suggestion task. To close the gap, I invent a highly accurate API method suggestion solution, referred to as Multi-HyLSTM, with specialized neural network designs to recognize unique programming language characteristics. My work points out the

important differences between natural languages and programming languages, which pure data-driven learning approaches may not recognize.

Neural Network-based Methodologies for Securing Cryptographic Code

Ya Xiao

(GENERAL AUDIENCE ABSTRACT)

Neural network techniques that automatically learn rules from data show great potential to provide vulnerability-agnostic solutions for securing code. Recent research community has witnessed the rapid progress of neural network techniques in various application domains, such as computer vision, natural language processing, etc. However, how to harness the success of neural network based approaches for dealing with programs is still largely unknown. Many fundamental questions are required to be answered. This dissertation aims to provide neural network based solutions to help developers write secure code, as well as answer several important but unknown research questions about promoting neural network based approaches specialized for the programming language domain. Learning from Java cryptographic code, I explore the accuracy challenges for neural networks to understand the secure API usage rules and generate appropriate suggestions based on them. One of my research focuses is on how to express code in a way that neural networks can comprehend, aka code embedding. Code embedding is the process of transforming code into numeric vectors. It is important for accuracy as all the subsequent neural network calculation is performed on it. I conduct a systematic comparison to evaluate several key embedding design choices and reveal their impacts on accuracy improvements. To further improve the accuracy, I focus on the accuracy challenges in the specific task, generating API suggestions by neural networks. I identify the unreported program dependency specific challenges and present several specialized neural network designs to address them.

Dedication

To everyone I love and who loves me.

Acknowledgments

I will never regret my decision of pursuing a Ph.D. in the past valuable five years. It has been one of the most precious journeys in my life, which fuels me strength and courage for future challenges. This experience makes me become wiser, more proactive, and stronger.

First of all, I would like to express my deepest gratitude to my parents. From my hometown to Beijing, from China to the U.S., no matter how far away, my parents are always caring about and supporting me. They are so enlightened and wise parents. From my little age to now, they always respect every decision I made and encourage me to follow my heart. They gave me their best without expecting me to pay back. One of the luckiest things for me is being their daughter and I'll try my best to make them feel the same.

I would like to express my sincere appreciation to my advisor Dr. Danfeng (Daphne) Yao. During the past five years, we shared our enthusiasm and dedication to do the excited research together. With her supervision, I gained a lot not only for the career skills but also for the life attitude. She is such an intelligent and supportive advisor who always gives insightful guidance and encourages students to shine. She taught me how to be a good researcher and form an insightful vision. Besides, I admire her unparalleled courage and great contribution for improving the environment for female professionals. She is a role model who taught me to be the person shaping the world instead of being shaped by the world.

I would also like to thank my committee members, Dr. Naran Ramakrishnan, Dr. Mathew Hicks, Dr. Patrick McDaniel, and Dr. Xinyang Ge. They helped me a lots for completing my Ph.D. research and gave me lots of insightful suggestions. Furthermore, I would like to thank all our collaborators and Lab mates for contributing to my research work.

At last but not least, I want to thank all my friends I've met in Virginia Tech. I am so grateful for them to accompany me through many happy and difficult moments.

Contents

List of Figures xii

List of Tables xv

1 Introduction 1

1.1 Background 1

1.2 Research Scope 3

1.2.1 The Need for High-Precision and Scalable Cryptographic API Misuse Detector 3

1.2.2 The Need for Systematic Code Embedding Measurement 4

1.2.3 The Need for Specialized Neural Network Design 8

1.3 Contributions 10

1.4 Dissertation Organization 14

2 Review of Literature 15

2.1 Code Embedding and Pretrained Language Models 15

2.1.1 Text-based Code Embedding 17

2.1.2 Structure-based Code Embedding 19

2.2 Neural Network based Code Suggestion 22

2.2.1 Token-level Code Suggestion 22

2.2.2 Statement-level Code Suggestion 23

2.2.3 Block-level Code Suggestion 24

3 Industrial Experience of Cryptographic Vulnerability Detection in Large-scale Codebases 26

3.1 Introduction on Cryptographic Vulnerability Detection 27

3.2 Background 30

3.2.1 Java Cryptographic API misuses 31

3.2.2 CryptoGuard 32

3.2.3 Data-flow Analysis in CryptoGuard and Parfait. 33

3.3 Detection Methods and Implementation 35

3.3.1 Detection Methods 35

3.3.2 Cryptographic Vulnerability Detection Implementation 36

3.3.3 Pseudo-influences and Refined Analysis 39

3.4 Accuracy Analysis and Real-world Findings 41

3.4.1 Accuracy Analysis on CryptoAPI-Bench 41

3.4.2 Evaluation on Real World Projects 42

3.4.3 Discussion 53

3.5 Conclusion and Future Work 54

4 Neural Network based Code Embedding **55**

4.1 Introduction on Code Embedding 56

4.2 Code Embedding Background 61

4.2.1 Token-level Embedding 61

4.2.2 Sequence-level Embedding 62

4.2.3 Cryptographic API Completion 63

4.3 Embedding Measurement Setting 63

4.3.1 Program Analysis Preprocessing Strategies 64

4.3.2 Token-level Embedding Settings 65

4.3.3 Sequence-level Embedding Settings 67

4.3.4 Dataset Overview 68

4.4 Evaluation Results 70

4.4.1 Comparative Experiments for RQ1 70

4.4.2 Comparative Experiments for RQ2 73

4.4.3 Comparative Experiments for RQ3 75

4.4.4 Comparative Experiments for RQ4 77

4.4.5 Analogy Tests of Token-level Embedding 80

4.5 Case Studies and Discussion 83

4.6 Related Work 86

4.7 Conclusions and Future Work 88

5 Neural Network based Code Suggestion **89**

5.1 Introduction on API Recommendation 90

5.2 Program dependence specific challenges 94

5.3 Our Approach 95

5.3.1 Overview of Our Approach 95

5.3.2 Dataflow based Context Extraction 96

5.3.3 Our Neural Network Design 100

5.4 Experimental Evaluation 104

5.4.1 Comparison with Existing Tools 105

5.4.2 Ablation Study 107

5.5 Related Work 114

5.6 Conclusions 117

6 Summary and Future Work **119**

6.1 Conclusion 119

6.2 Future Work 121

Bibliography **124**

List of Figures

3.1 The comparison of the ordinary iterative analysis and IFDS analysis. (a) is a program demo. (b) shows the ordinary flow-set based analysis by collecting and updating a flow set. (c) shows the IFDS based analysis process which builds edges and then summarizes edges. 34

3.2 The inter-procedural analysis under Parfait's layered framework. This design is important to achieve the scalability of Parfait. 37

3.3 The false-positive reduction refinements represented in IFDS. It shows the dataflow propagating edges for three situations. The above one is the default propagating edges. The bottom one is the refined propagating edges. 39

3.4 Runtime performance of Parfait for screening the eleven real-world codebases. The size shows how many lines of code these codebases have. 43

3.5 The number of vulnerabilities reported by Parfait and CryptoGuard in the eleven real-world industrial applications. The upper area of the x axis shows the true positive alerts while the bottom area of the x axis shows the false positive alerts. Nine of them are internal codebases of Oracle. Two of them are open-source projects. 44

3.6 The number of vulnerabilities reported by Parfait and CryptoGuard in the eleven real-world industrial applications. We break them down into the weak PRNG vulnerabilities and the other vulnerability types. Parfait reported 0 weak PRNG vulnerability. CryptoGuard reported 48 weak PRNG vulnerabilities while only 1 of them is a true positive case. Parfait and CryptoGuard both achieves 100% precision in the other vulnerability types excluding the weak PRNG cases. 46

4.1 API and constant sequences from byte code, program slices and API dependency graphs 63

4.2 Case Study 1. (a) A slice sequence that is predicted incorrectly. (b) Model trained on API dependence graphs makes the correct prediction. 84

4.3 Case Study 2. (a) A test case that is correctly predicted with *slice2vec*, but incorrectly with one-hot vector. (b) A test case similar to (a), but both *slice2vec* and one-hot give the correct prediction. 85

5.1 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 give identical predictions for them. 92

5.2 Examples illustrating dependence challenges in code completion and how we address them. 93

5.3 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. 98

5.4 Target amplification in HyLSTM with our new hybrid loss function. Through the extra FCL_1 and o_n, similar input sequences followed by different x_n are supervised by stronger signal to highlight their differences. 100

5.5 Multi-path code suggestion architecture based on aggregating path embeddings 102

5.6 The top-1 accuracy of SLANG and our approach on API completion 106

5.7 The 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. . . 111

5.8 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. 111

5.9 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. 113

List of Tables

3.1 Error-prone Java Cryptographic APIs covered by Parfait's cryptographic API misuses detection and the eleven involved vulnerability types in CWE. S stands for severity level from CryptoGuard [147]. 32

3.2 Parfait's Evaluation Results on 158 Test Cases of CryptoAPI-Bench. There are basic cases (intra-procedural), different inter-procedural cases that require across methods, across classes, field sensitivity, path-sensitivity, and heuristics to handle. FPs represents false positives. FNs represents false negatives. . . 41

3.3 False positive reduction derived from applying the refinement insights (RIs). We compare Parfait cryptographic vulnerability detection with its intermediate version without the refinement insights. 42

4.1 The overview of our comparative settings. Each cell shows the embedding and machine learning model we use for the cryptographic API completion. We have comparisons between three program analysis preprocessed sequences, token-level embeddings vs. one-hot, and sequence-level embeddings vs. token-level embeddings. 59

4.2 The library sources of our embedding APIs 66

4.3 Overview of our datasets 68

4.4 Embedding corpora statistics of the basic dataset 69

4.5 Accuracy of the next API completion on the basic dataset. 71

4.6 Accuracy of the next API sequence completion on the basic dataset. We use the LSTM-based sequence model with hidden layer size 256 for this task. . . 71

4.7 Accuracy of the next API completion with or without sequence-level embedding (pretrain) on the basic dataset. 72

4.8 Accuracy of the next API completion with or without sequence-level embedding (pretrain) on dependence paths of the advanced dataset (cross-app learning). 76

4.9 Accuracy of the next API completion with or without sequence-level embedding (pretrain) on byte code sequences of the advanced dataset (cross-app learning). 76

4.10 Accuracy of the next API completion by finetuning the general purpose pretrained model GraphCodeBert and CodeBert. 78

4.11 Four categories of analogous pairs we define among API methods and constants. We give a representative example for each category, where two pairs (a and a' vs. b and b') have a high degree of relational similarity (i.e., analogous) in terms of some programming property. For each category, the number of analogies used in our top k evaluation (Table 4.12) is also shown. 81

4.12 Top k accuracy of 14 analogous pairs in different embedding vectors. Smaller k suggests more accurate embedding vectors that better maintain analogous relationships. 82

5.1 The top-1 accuracy of Codota and our Multi-HyLSTM on 245 randomly selected Java Cryptographic API invocation test cases. Codota gives recommendation 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). 106

5.2 The accuracy of HyLSTM for the next API recommendation. Acc.(A) refers to the in-set accuracy for all the test cases. Acc.(K) is the in-set accuracy for the test cases with known features. Acc.(U) is the in-set accuracy for the test cases with unknown features. 108

5.3 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 5.2. 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, 5 are new Apps that are never used during training. 109

5.4 The average accuracy of HyLSTM and the two regular LSTM models tested on four new test sets composed of new Android Apps. 110

5.5 Comparison between multi-dependence suggestion and sequential suggestion. A, K, and U stand for in-set accuracy for all cases, known cases, and unknown cases, respectively. 112

Chapter 1

Introduction

1.1 Background

Manual code generation is error-prone and causes vulnerabilities [11, 69, 75, 115, 147, 182]. Cryptographic API misuse issues, such as exposed secrets, predictable random numbers, and vulnerable certificate verification, are reported common in practices, which seriously threatens software security [66, 147, 175]. Due to the lack of security expertise, it is difficult for developers to use cryptographic APIs correctly. The complex documentation and misleading online forum posts even worsen this situation. These cryptographic issues are only the tip of the insecure coding iceberg. Many studies show that writing security code is challenging [12, 52, 66, 115], even for experts [175]. The lack of security training in developers is a serious issue in the software engineering industry [115]. In addition, the organizational culture and the upper administration may not encourage or incentivize developers security efforts. Thus, it is unrealistic to assume that developers will better themselves on their own without any external tool help.

To aid the insecure coding practices, current cybersecurity research has been largely focused on automatic detection, prevention, as well as proof-of-concept attack development [124, 126, 142, 159, 179]. A huge missing link in the cybersecurity ecosystem is the lack of systematic research on security repair suggestions. Current code repair practices heavily rely on manual effort. Because of the complexities of modern programs and systems, this process is

challenging and time consuming [58]. Existing automatic code suggestion or repair methods are still in the infancy [74, 136]. Genetic programming-based repair approaches [99, 100] generates a one-line mutation each time and validate the mutation by a test suite. However, it only applies to bugs that can be fixed with one-line change, which make it fail for complex vulnerability repair that requires changes in multiple lines. Moreover, the effective patches are too expensive to be found since the mutation space is too large while the effective ones are sparse. These patches passing the test suite are also verified as plausible rather than real patches [144]. Manual defined template is another way to automatically fix vulnerabilities [97, 112]. However, this approach does not generalize for the endless increasing vulnerability types.

The attractive vision of automatic code engineering, e.g., repair [74, 101, 109, 110] and generation [26, 150, 161, 184], has motivated a line of neural networks based machine learning solutions [79, 80, 82]. 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. Although some related studies exist [18, 22, 87, 149], how to harness the deep learning revolution for software and software security is largely unknown. Many in-depth problems remain unanswered. For example, what are the design requirements when generating embeddings? What are the program specific challenges for neural network to learn from codebase? What are the fundamental limitations of deep-learning based code generation? This fundamental research effort is crucially important for creating concrete solutions and will have long-lasting impacts on the field of software security. The lack of benchmarks and datasets in the area of deep learning for software security also hinders the advancement of methodology development.

My dissertation work aims to help developers write secure code with deep-learning based approaches. The ultimate goal is to develop an automatic or semi-automatic secure code

generation system. This work focuses on a specific type of secure coding problem, the cryptographic API misuses, as it is a challenging and important problem for software security. With neural networks learning from huge amount of secure cryptographic code, I expect to secure the Java cryptographic code by generating the code suggestions based on the previous code context. This work also aims to build fundamental roadblocks for the neural network based programming language learning solutions and promote the interpretation for some important design decisions. Under this vision, I introduce the specific research problems I focus on and the contributions I have achieved during the Ph.D. research exploration in the rest chapter.

1.2 Research Scope

My dissertation work makes research efforts to deepen the understanding and improve the accuracy and security of neural network based programming language learning. Specifically, I address three critical research problems towards this goal, 1) *the real-world cryptographic vulnerability detection*, 2) *the comprehensive measurement study on code embedding approaches* 3) *the specialized neural network design for cryptographic API completion.* I introduce the necessity and significance of tackling the three research problems.

1.2.1 The Need for High-Precision and Scalable Cryptographic API Misuse Detector

First of all, we focus on the detection of the cryptographic API misuses. To improve the secure coding practices, detection is the first step. With a precise detector, the reported vulnerabilities can be identified and get necessary attention to fix. There are many static

analysis tools, such as FixDroid [129], CogniCrypt [97], CryptoGuard [147], presented to screen the codebase and expose these vulnerabilities. However, the acceptance and prevalence of these tools in the industrial community are still low [192], which suggests a gap between the state-of-the-art tools and the real-world demands.

Moreover, to train the neural network to produce code suggestions, the data quality is very important. It is necessary to guarantee that the code the neural networks learn from is secure. Therefore, a precise and scalable detector is also important to work as a filter for training data.

1.2.2 The Need for Systematic Code Embedding Measurement

Code embedding is a fundamental concept of neural network based solutions for programming languages. It refers to the process of automatically learning the low-dimensional vector representations of program elements [25, 86, 197]. Intuitively, it is about how to meaningfully express code in vectors. This transformation is important, as subsequent inference or decision tasks are performed on the embeddings of code.

Accurate vector representations can achieve significantly higher accuracy of downstream tasks with much fewer data and less training time. There have been multiple interesting research solutions that embed programming languages or machine instructions [25, 64, 68, 86, 132, 197]. Using program analysis techniques to guide the code embedding process enables the embedding to better capture the semantic relationships between embedded tokens.

Despite these recent advances, 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 and embedding-based applica-

tions. We outline multiple important research problems in this direction, relating to the following:

The need for comparing pure data-driven approaches vs. hybrid approaches guided by program analysis. Data-driven solutions refer to training based machine learning approaches, i.e., learning from massive amounts of samples. For example, for code completion tasks, a pure data-driven model can be obtained by training a Long Short-term Memory (LSTM) model with correct code sequences. There is a line of research that develops pure data-driven solutions for code related tasks [46, 68, 93, 94, 163].

In contrast, classic static or dynamic program analysis approaches require no training. Even with powerful machine learning, conventional program analysis techniques should not be abandoned, as program analysis (static or dynamic) techniques provide a direct and valuable channel to extract code properties. Data-driven approaches can be adjusted to incorporate program analysis insights, resulting in hybrid approaches.

A number of recent code embedding solutions (e.g., [23, 42, 85, 195]) stressed the advantages of cooperating with program analysis to leverage the structural information of code. For example, the authors of [195] learned code embedding after constructing the graph representations (e.g., control flow graphs, data flow graphs) of code. Hellendoor *et al.* [85] advocated a hybrid embedding method that considers both the graph structure and the raw sequence-level information to overcome the size limit of graphs. However, there have not been systematic studies on how various hybrid approaches compare with a pure data-driven approach or with each other, in terms of downstream task performance.

The need for comparing token-level embeddings vs. pre-trained models with fine-tuning. Code embedding [19, 25, 86] automatically learns the vector representations of code tokens. General-purpose embedding (e.g., [83, 132]) is pre-trained without downstream

tasks and can be reused, which is advantageous, as training in neural network settings is usually time-consuming.

Token-level embedding aims to obtain a static vector for representing a code token, no matter what context it appears in. These vectors can be stored in a look up table and reused for different tasks that relate to the semantics of the code tokens. For example, The work in [132] applied the word2vec solution to obtain the token-level embedding of the encountered Javascript and C# APIs. These embedding vectors are effective map the Javascript APIs to functionally similar C# APIs. Authors in [31] trained the token-level embeddings of the LLVM IR instructions and demonstrated the advantages of using them in several performance prediction tasks (e.g., computer device mapping) over the specialized traditional approaches.

Pre-trained language models are neural network models that take as inputs language sequences and produce contextual embedding for them. Contextual embedding assigns a program token with different representing vectors based on its context in the given sequence. Contextual embedding is considered a good solution for the polysemy problem of words. Pre-trained language models can be fine-tuned to different downstream tasks with a small amount of extra task-specific data. For example, Karampatsis *et al.* [94] produced the code contextual embedding SCELMo by pre-training the ELMo [141] language model on a large JavaScript source code corpus. SCELMo is then fined-tuned by the bug detection task to show its effectiveness.

Side-by-side experimental evaluation of the various design choices of these embeddings and their application scopes [68, 78, 91, 94], would be useful. The evaluation can be conducted on accuracy and overhead, as well as the ease of deployment.

The need for metrics and benchmarks. Accuracy in downstream tasks is an ultimate

performance metric, however, intermediate metrics would also be useful for quickly assessing embedding quality. For example, in natural language settings, embedding vectors are usually validated using the analogous pairs (e.g., $men - women \approx king - queen$) method [117, 118, 119].

However, in programming languages, identifying such analogous pairs requires substantial efforts. Nevertheless, one can still similarly define analogous pairs as two pairs of APIs or constants, (a and a') with (b and b'), having a high degree of relational similarity (i.e., analogous), in terms of some programming property. For example, for Java cryptographic code, one can identify multiple categories of analogous pairs, which we briefly describe next. *Direct Dependency:* for two accompanying APIs where one accepts the other's output, they form a direct dependent pair. *Argument Symmetry:* There is an analogous relation between API and constant pairs. *Syntactic Variants:* functionally equivalent APIs have different arguments types or return values. Such analogous pairs based metrics are useful and easy-to-understand sanity checks for some code embeddings.

Besides metrics, task-specific benchmarks (with labels) are also needed. In order to compare with existing approaches, code transparency in published solutions needs to be improved. To improve it, we made a Java cryptographic code completion benchmark for measurement [1]. It includes the Java cryptographic code collected from 79,887 Android Apps. These Apps are processed by different program analysis strategies to produce API sequences from byte code, inter-procedural program slices, and data-dependence paths. These sequences can be used to evaluate the effectiveness of embedding design choices combined with different program analysis strategies.

Measurement work is often under-appreciated in the research community, as it does not present new neural network solutions. However, while inventing the next-generation neural

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

network model or application is important in this rapid-moving field of AI, our software engineering research community should also encourage and foster seemingly-less-attractive-but-equally-important measurement studies. Such investigation would reveal insights and principles that guide future designs towards the vision of automatic code generation and repair.

1.2.3 The Need for Specialized Neural Network Design

Programming language models require specialized neural network design. Although many existing solutions show promising results by borrowing similar model and training paradigm from natural language processing [51, 154, 163, 178], the programming languages are quite different from the natural languages on several aspects. On one hand, the APIs, algorithms, and coding patterns are much fewer and more rigid than the flexible expression of words. On the other hand, there are more complicated structural information and relationships (e.g. control flow branches, method invocations, etc.) in code corpus. The natural language and programming language, which one is easier to learn? There is no agreement yet.

Many studies directly applied the neural network based solutions designed for natural languages. Many studies build statistical language models on source code token sequences [33, 48, 102]. The program are treated as natural language sentences composed of code tokens. The standard models such as RNN, LSTM, and BERT are directly applied on the tokenized code sequences for code generation. Moreover, the generated code is often measured by matching based metrics designed for natural languages [154]. For example, bilingual evaluation understudy (BLEU) score [137] is one of the most common metric to measure the quality of a generated code sequence. However, BLEU score can only calculate the similarity between a sequence with its reference. There is no guarantee that the generated

code is syntactically correct or semantically functional. Without paying attention to the program-specific properties and requirements, there could be many issues in the existing code generation solutions.

We briefly summarize the major differences between programming languages and natural languages that requires special attention.

Faraway Context vs. Local Context. Modulization and encapsulation are important features of the modern programs. One functionality may be implemented into separated modules. Therefore, when only looking at the local context in one module, it is difficult to understand the full functionality of the code. Furthermore, the code body implementing a developer-defined method will be encapsulated. When invoking the developer-defined method, the actual executed code may locate far away from the current invocation place.

This programming language specific property makes special requirement for neural networks to identify and understand the faraway program context. The vicinity-based program context is not reliable and often misses important contextual information. However, most of the state-of-the-arts deep learning solutions for software are not able to address this issue. Many existing solutions only feed the local context within one method to the neural networks [49, 55, 102, 168, 169]. Even though the AST or dependence graphs are extracted, the neural networks cannot make correct decisions when necessary context is missing.

Structural Patterns vs. Sequential Patterns. Programs have abundant structural patterns, which is difficult to be handled by sequential neural network models. Different from natural languages, there are various control flows between code statements. For example, the `if` condition and `for` loop causes branches or circles that propagate information in a way beyond sequence. In addition, the data dependence between code statements are also complex. One code statements can often impact and be impacted by multiple code

statement simultaneously. A minor change in a code sequence may result in significant differences reflected by its structure. The sequential models, such as N-gram, RNN, LSTM, that are designed for sequential patterns may have difficulties to recognize the structural control and data flows that are important for code semantics.

1.3 Contributions

My dissertation work achieved contributions by making efforts to address the three research problems mentioned in Section 1.2. These contributions not only target to present specific neural network solutions for secure cryptographic code, but also benefit the future research by promoting the understanding and answering the fundamental questions of adapting neural network techniques for programming languages. We summarize our contributions to the domains of software security and deep learning on code, respectively.

Our Contributions to Software Security. We realize an industrial-strength detection tool and a high-precision suggestion tool for an impactful software security problem, the cryptographic API misuses. Our contributions include:

- We realize the complex Java cryptographic vulnerability detection in Oracle's Parfait platform. Specifically, we identify eleven CWE types caused by misusing Java cryptographic APIs and eighteen associated cryptographic API methods. The detection relies on a backward inter-procedural, flow-, context-, field-sensitivity data-flow analysis under Parfait/LLVM support. We design different alarm criteria to detect these cryptographic vulnerabilities.

- We specialize the backward IFDS taint analysis provided by Parfait to overcome the precision challenge caused by pseudo-influences that are security-irrelevant constants

used in constructing security-critical values. Inspired by the refinement insights in CryptoGuard [147], we define the refinement rules in the form of IFDS dataflow analysis with LLVM IR. The refined analysis significantly not only reduces false alarms but also improves scalability.

- We evaluated the precision and scalability of Parfait cryptographic vulnerability detection on the comprehensive CryptoAPI-Bench [13] and several large-scale industrial applications. The experience result demonstrates that our detection achieves a high precision (86.62%) and recall (98.40%) overall. The precision excluding the path-sensitivity test cases even reaches 100%. Parfait-based cryptographic vulnerability detection achieves 100% precision on the eleven large-scale applications. The runtime for codebases that include 2K to 1321K lines of code ranges from 2 seconds to 36 minutes and most of them can be finished within 10 minutes. We further show some noteworthy examples to help readers better understand the practices.

- To help developers write secure code easier, we present a high-precision cryptographic API suggestion tool to recommend the appropriate API method call given the precious code context. Our tool improves the accuracy by a well-designed program dependence analysis and the specialized neural network designs. Demonstrated by the comparisons with two state-of-the-art API suggestion tools SLANG [150] and Codota [6], our tool achieves a best accuracy at 98.99%, outperforming the two state-of-the-arts in the top-1 accuracy by large margins (18% and 51%).

Our Contributions to Deep Learning on Code. To understand the impact of various neural network design choices and deliver research implications, we conducted comprehensive experimental comparisons in code embedding settings and our specialized neural network designed for API suggestion.

1) Our major findings related to code embedding techniques include:

- Our findings show that program analysis preprocessing plays a significant role in Crypto API embedding and completion. For both token-level embedding and sequence-level embedding, the API dependence paths produce higher prediction accuracy, compared with slices and byte code. With program analysis, the token-level embedding *dep2vec* achieves an accuracy 36% higher than *byte2vec*. The sequence-level embedding *depBERT* achieves an accuracy 45.86% higher than *byteBERT* without program analysis preprocessing.

- Our findings show that applying embeddings with program analysis significantly improves the task accuracy compared with the one-hot baseline (no embedding). On dependence paths, the token-level embedding *dep2vec* and sequence-level embedding *depBERT* both outperform the one-hot encoding baseline by the accuracy boost of 6% and 7%, respectively. Although sequence-level embedding is slightly (0.55%) better than token-level embedding in our experiments. Considering the expensive cost of sequence-level embedding, token-level embedding is more recommended according to it.

- Our findings show that the improvements derived from program analysis and embedding is valid for cryptographic API completion on new apps. In cross-app learning scenario, the program analysis guided embedding *depBERT* and *dep2vec* still achieve good accuracy at 95.75% and. 93.58%, respectively. Another observation is the advantage of *depBERT* over *dep2vec* is slightly more obvious by the 2.17% accuracy boost compared with 0.55% in the basic setting. The sequence-level embedding *depBERT* is most recommended to be used in the data scarce situation, as the largest improvement (5.10%) of *depBERT* compared with *dep2vec* is observed on the smallest task dataset

with 26,357 dependence paths.

- The state-of-the-art general purpose source code embedding solutions GraphCodeBert and CodeBert are insufficient in our cryptographic API completion tasks with the low accuracy 59.94%. Experiments still shows the advantage of applying program analysis preprocessing in their embedding solutions. GraphCodeBert substantially outperform its non-program-analysis counterpart CodeBert by accuracy boost of 20.07% on average. The experiments also suggest the method-level context is more recommended than the class-level context for Cryptographic API completion.

2) Our major contributions to specialized neural networks are summarized as follows.

- We identified two previously unreported challenges for neural networks to predict code. We experimentally validated the limitation of the state-of-the-art models (e.g., LSTM, BERT) in learning program dependencies. We performed in-depth manual analysis on 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 for 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 HyLTM. This neural network works together with our program context representation part, – API dependence graph construction and multi-path extraction algorithm, to accurately capture the program dependencies for an API call.

- We conducted an extensive ablation study to validate the effectiveness of our design choices. We identified 774,460 Cryptographic API callsites from 64,478 Android Apps.

Experiments show that our multi-path architecture design excels at the inference capability. Multi-HyLSTM and Multi-BERT improve the accuracy for unknown cases by 11.53% and 36.50% compared to HyLSTM and DepBERT, respectively. Besides, our design, HyLSTM, outperforms two regular LSTM models. It improves the inference capability of the LSTM with token-level loss by 7.94% and the LSTM with sequence-level loss by 15.47%.

We have also published a large-scale Java cryptographic code dataset [2] that can be used as a benchmark to evaluate API completion model accuracy.

1.4 Dissertation Organization

The structure of this dissertation is as follows. Chapter 2 is the literature review about the related studies of neural network based software engineering solutions. Chapter 3 introduces our experience of developing a precise and scalable cryptographic vulnerability detector for industrial codebases. Chapter 4 focuses on our design and comprehensive comparison experiments on program-analysis guided embedding techniques for cryptographic API methods and constants. Chapter 5 presents our secure code generation approach based on recommending the next API method given previous lines of code. I summarize the dissertation work and discuss the future work in Chapter 6.

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

Chapter 2

Review of Literature

2.1 Code Embedding and Pretrained Language Models

With the rapid progress of neural network based approaches in many application domains, such as natural language processing [63, 108, 118] and computer vision [153, 176], researchers in software engineering field are paying increasing attention to leverage neural network techniques help with code related tasks. Many neural network models are applied to solve software engineering problems, such as clone detection [44, 64, 194], program repair [79, 80, 82], code summarization [14, 81], code completion [87, 162, 163] and synthesis [16, 121], etc. Despite of the various task-specific designs, a general research problem that all kinds of code-related applications need to address is how to parse a piece of code with neural networks. Neural networks accept a group of numeric vectors and transform them into different dimensional vectors in the latent space through complex calculation. The process of mapping program code into numeric vectors for neural network processing is referred to as code embedding.

Embedding is an important process for the following neural network modeling, which has been observed in many studies [39, 63, 117, 118, 119, 140, 141]. The most naive approach to vectorize the language tokens is the one-hot encoding. Under the one-hot encoding, the language tokens in a vocabulary are indexed with sequence numbers. The k-th token is represented by a vector consisting of a 1 at the k-th dimension and 0s at all other dimensions.

The drawback of this approach is that the relationship between tokens are not included. All the vectors are orthogonal and not depend on others, which neglects that language tokens often have syntactical or semantic connections (e.g., synonyms) between each other. Another deficiency is that the vectors are sparse and inefficient. In contrast, good embedding vectors reflect the semantics in arithmetic relation. For example, with word2vec embedding [117, 117], synonyms are represented by close vectors. Moreover, similar relationships are preserved in the vectors. An example shows that the difference between the vectors for king and queen is similar to the difference between the vectors for man and woman ($vec(king) - vec(queen) \approx vec(man) - vec(woman)$).

Despite that the great benefits of the word embedding approaches, such as word2vec [117, 117] and BERT [63], are well-recognized, similar success of code embedding solutions haven't been observed yet. Embedding code is more challenging than embedding words. The reason is multi-fold. First, programs have various representation formats, including the high-level source code, intermediate representations during the compilation, and the low-level machine code. With different compilation systems or computer architectures, a piece of source code may be translated into different intermediate representations or machine code. The complexity of these representations increases the difficulties of an unified embedding approach. Second, programs have delicate syntax rules. As a formal language, programs are usually parsed with a set of strict syntax rules, which is different from the flexible natural languages. A small difference in source code may result in different code syntax. The embedding of code needs to identify the syntactical differences even the code is text-based similar, which is very challenging. Furthermore, programs have rich structural information. While embedding for words can only focus on the sequential relationship between words, code embedding is required to capture and understand various code specific dependencies caused by the complicated structures.

To solve these challenges, many studies have been proposed for the code embedding approaches. We introduce the related literature from the two main categories, **text-based code embedding** and **structure-based code embedding**.

2.1.1 Text-based Code Embedding

Text-based code embedding approaches [19, 68, 93, 128, 131] follow the idea of the well-known word embedding solutions that treat programs as sequences composed of code tokens. Early research efforts [19, 20, 128, 131, 132] apply similar context-based embedding training as word2vec. The source code based context of each token in a large code corpus is collected. The skip-gram model [117] is used to take a code token as input and predict its neighboring tokens. The vector in the latent space is recorded as the embedding vector for the input token. Based on this approach, the embedding vector of a token is decided by the statistics of its neighboring tokens in a large code corpus. These embedding vectors show effective in many semantic related tasks. For example, the embedding vectors of the API elements are used to map the analogical APIs and help API migration tasks [45, 130, 131]. Nguyen *et al.* [131, 132] mapped the semantic similar JDK APIs with .NET APIs through their similar embedding vectors. Chen *et al.* [50] trained the API embedding to infer the likely analogical APIs between third party libraries. Gu *et al.* [77] trained the RNN-based Encoder-Decoder model with the API embedding to translate between natural language sentences and API sequences. The embedding vectors of variables are used to suggest semantic-aware names for identifiers [19, 20].

Most recently, given the abundant resources of the open-sourced data and training devices, pretrained large language models become a popular trend for code embedding. Language models are those sequence-based models (e.g., LSTM, Transformers) trained on large lan-

guage corpus to evaluate the probability of a language sequence. The pretrained language models can be used for generating embedding vectors or fine-tuned for various downstream tasks, such as code completion, and code summarization. Given a sequence of tokens as inputs, the pretrained language models can produce the embedding vectors for every token in this sequence. Compared with the word2vec-like embedding, the embedding vectors generated from language models are contextualized because they are produced not only for the code tokens itself but also taking their accompanying tokens in the given sequence into consideration. In another word, pretrained language models produce the sequence-level embedding vectors while the word2vec-like embedding produce the token-level embedding vectors.

With the huge amount of public code repositories, there are several large language models pretrained on source code [28, 35, 36, 51, 68, 93, 171, 177, 178, 185]. Most of these pretrained language models utilize the neural networks and pretraining tasks of BERT [63] or GPT-3 [43], which are successful pretrained language models for natural languages. Among these work, the pretrained programming language models, such as CodeBert [68], CuBert [93], follow the BERT embedding approach. The Transformers neural network [174] is used. The tokenized code sequences are input to the neural network and expected to predict for the masked language modeling. Masked language modeling is a well-designed pretraining task. It randomly masks a few tokens in the given sequences and aims to predict the original tokens at the masked positions. With this pretraining task, the language model is trained to reconstruct the given sequences based on the bidirectional context. Therefore, the produced embedding vectors incorporate the semantic meaning of the tokens with the bidirectional context. Another type of pretrained programming language models, such as CodeGPT [111], CodeParrot [171], GPT-NEO [35], GPT-J [177], Codex [51], PolyCoder [185], follow the auto-regressive language modeling manner. Auto-regressive language modeling is trained by

predicting the next token given the previous tokens in a sequence. The Transformers neural network is used to generate the embedding vector for every token in a sequence based on its left context.

Although these large pretrained language models for code have shown promissing results in various downstream applications, there are some common issues they suffer from. Most of these models are trained on top of the tokenized source code. Beside the programming key words, programs can have unlimited tokens that are customized identifier names and function names. The open vocabulary of programming languages causes a serious out-of-vocabulary (OOV) problem for embedding. That is, there are more likely to encounter new tokens that are unseen in the pretraining phase. These OOV issue can seriously impact the embedding quality [84, 95]. To solve this issue, advanced tokenization techniques [116, 158, 173] are often used. Moreover, since the pretrained language models are directly built on the source code tokens, syntax or semantic rules are not implied. When using them for the purpose of generation, there is no guarantee that the generated sequence would follow the program syntax or semantic rules. Therefore, optimized techniques such as beam search [181] are often required to guarantee the output sequence quality.

2.1.2 Structure-based Code Embedding

Different from the text-based code embedding, structure-based code embedding aims to incorporate the structure nature of programs. Abstract syntax trees (ASTs) are largely used to explicitly show the syntax information of the code [25, 26, 34, 91, 114, 152]. Alon *et al.* present a piece of code as a bag of paths from AST [24, 25, 26]. The AST paths are embedded as vectors and aggregated through an attention network. Besides, many studies presented tree-based neural networks, such as Tree-LSTM [180], Tree-Bert [91], Tree-CNN [120], to

learn the syntactical knowledge from the tree structure. These neural networks are either equipped with modified architecture [180] for trees or advanced tree traversal approaches [91].

However, AST based representation is highly limited by its size and code coverage. The faraway program context may not be included in an AST. To solve this problem, Zhang *et al.* [189] built large ASTs and split a large AST into a sequence of small statement ASTs to model. They introduced an advanced AST-based Neural Network (ASTNN) to deal with the decomposed large AST. Another approach to solve this issue is through augmented information [34, 107, 188]. In the ordinary ASTs, terminal nodes are mainly determined by its local context, which are the parent nodes, according to the probabilistic context free grammar (PCFG) [114]. Bielik *et al.* extend PCFG into the probabilistic higher order grammar (PHOG) that incorporates faraway context beyond the AST neighboring nodes to build the conditional models. Additionally, many studies use ASTs as the backbone and augment them with various dependence edges to include the semantic information [23, 195].

Besides, program analysis preprocessing are often used to explicitly embed programs with semantic information. Early attempts combine various program analysis strategies with the embedding training. For example, Henkel *et al.* [86] performed intra-procedural symbolic execution and trained embedding vectors of symbolic abstractions from symbolic traces. Zhao *et al.* [194] embedded program functions starting from a semantic feature matrix obtained based on the control flow analysis and data flow analysis. Li *et al.* [105] decomposed software into code gadgets with program slicing and then embedded the gadgets into vectors.

With the help of program analysis, graph-based representations are most common choices observed in recent efforts. Various graphs are constructed for code embedding. Xu *et al.* [186] embedded binary functions based on the constructed graph representation called attributed control flow graph (ACFG). Asm2Vec [64] transform code functions into control flow graph (CFG) of assembly instructions as nodes. The random walk strategy is used to process the

graph and extract the context of a node to perform word2vec-like embedding. Beside the control flow, data flow is more and more used in code embedding solutions. Ben *et al.* [31] learned embeddings of LLVM IR instructions in contextual flow graphs that are constructed based on both the control flow and data flow. Allamanis *et al.* [23] propose a type of graph by adding the data flow edges on top of the ASTs. Zhou *et al.* [195] constructed a type of graph that incorporates source code context edges, AST edges, control flow edges, and data flow edges.

There are different approaches to process these graphs and produce code embeddings. the context-based embedding approaches, such as word2vec are often combined with random walk or graph path traversal to handle graph inputs [31, 64, 76]. Besides, many studies borrow the graph embedding or network embedding approaches [23, 47, 59, 103, 106, 135, 160, 190, 193]. Flow2vec [160] and High Order Proximity preserved Embedding (HOPE) [135] use matrix factorization technique [62] to generate embedding vectors for every graph vertex based on their topology. Another approach is through graph neural networks (GNNs) [73, 156] designed for graph-structured data. Allamanis *et al.* [23] applied the Gated Graph Neural Networks (GGNN) [103] to embed their program graphs. Zhang *et al.* introduce a heterogeneous graph embedding approach to handle the heterogeneous program graph they constructed. However, the scalability is still a bottle-neck for those graph based neural network approaches. Sequence-based models with specialized designs are still good options to be scale for large code scope. With the graph-guided attention mechanism, the Transformer neural network is able to take the sequence inputs with their graph structure. The published large-scale pretrained language model, GraphCodeBert [78] is a representative of this category.

2.2 Neural Network based Code Suggestion

Many studies validated the naturalness of code sequences [22, 87, 88, 148, 149, 167], which inspires a line of work to build language models for code completion/suggestion tasks [32, 33, 48, 155, 163, 170]. These language models are trained on a large code corpus and calculate the statistical probability of each given program. An important application the language models enable is code suggestion. With the estimated conditional probability calculated by language models, the code candidate with the highest probability is suggested.

Code suggestion is a broad concept that generally refers to generate new code at the targeted place based on the given condition. There are many different specific scenarios or requirements for code suggestion. Hence, there are various designs for the neural network and program analysis to meet different requirements. We introduce the state-of-the-art code suggestion work based on there different goals.

2.2.1 Token-level Code Suggestion

A common objective of code suggestion is to generate a program token given the program context. Next token suggestion or predicting for program with a hole are two usual paradigms of the token-level code suggestion. Next token suggestion aims to predict the next token give all previous code tokens. The sequential models are often used to parse programs as a sequence of tokens $(t_1, t_2, \ldots, t_n)$ and model the probability of the n-the token as $P(t_n|t_{n-k}, \ldots, t_{n-2}, t_{n-1})$ where k is the window size for the context [29, 95]. The recurrent models are often applied for this task. For example, Campbell *et al.* [48] utilized the n-gram model trained on the code tokens to localize and give suggestions for Java syntax error correction. Bhatia *et al.* [32, 33] trained the recurrent neural networks (RNN) on the student programming assignments to solve syntax errors in C++ by replacing or inserting

suggested code tokens.

Instead of the general program tokens, a line of research focuses on the specific type of tokens, such as API calls [71, 127, 130, 143, 150]. These approaches have specialized designs for the required context. Raychev *et al.* [150] defined the concrete semantic history that is a sequence of concrete events to be the given context. Proksch *et al.* [143] applied static analysis to extract object usage sequence as the context. Nguyen *et al.* [127, 130, 130] built a graph to represent the API usage scenario for modeling the next API usage. The recent progress Pythia [162, 164] trained the statistical language models on the file-level ASTs of a large scale codebase and is able to generate the suggestion for API methods at the edit time.

Predicting for program with a hole aims to predict a missing token given the bidirectional program context. With a missing token at the n-th place, the neural network models its probability as $P(t_n|t_{n-k}, \ldots, t_{n-1}, t_{n+1}, \ldots, t_{n+k})$ [18, 19, 23]. The masked language modeling follows this paradigm to train language models [68, 78, 93]. Besides, Allamanis *et al.* [23] defined two identifier name related tasks, VarNaming and VarMisuse that leave the identifier as a hole and model it predict the name for it based on the surrounded context. The surrounded context is represented as the AST and dataflow graphs.

2.2.2 Statement-level Code Suggestion

Sometimes, only suggesting a token is insufficient. There is a line of research aiming to generating an entire code statement composed of a sequence of tokens []. A straight-forward approach is to extend the token-level suggestion model to produce a sequence of suggestions. The encoder-decoder based sequence-to-sequence models are often used to serve this purpose. A decoder neural network is added after the pretrained encoder layers [68, 78].

However, the generated sequence may contains syntactical errors because there is no syntax

control for the sequence generation. Svyatkovskiy *et al.* [163] show that this issue can be well handled by applying the beam search strategy in the decoding phase. With beam search, not only one sequence but a tree of tokens are generated. At every step of token generation, the top-n candidates are produced as the children of their previous node. In this way, a larger generation space is searched and the sequence with highest overall probability is suggested. Jiang *et al.* [90] introduced an optimized beam search by adding the code-aware filtering strategies. The generated identifier tokens are checked by static analysis and the length of the entire sequence is controlled. Different from the search based approaches, Brockschmidt *et al.* [41] guarantee the syntactically correct code by ASTs. Instead of producing a sequence of code tokens, the authors choose to produce an AST of a missing code expression and then transform the generated AST into source code.

2.2.3 Block-level Code Suggestion

A more challenging requirement is to generate an entire code block or function. There are different assumptions of the block-level code suggestion. Murali *et al.* [122] described a scenario that requires to generate a piece of source code given a sequence of APIs. mukherjee *et al.* [121] defined a task that generate an entire Java method implementation based on the reminder code in the same class. Another general assumption is to generate a piece of code by translating a sequence of natural language text or other programming languages [15, 51].

In these block-level code suggestion tasks, the search based generation control may be inefficient and too expensive. Several specialized approaches to enforce the program structure in the generation phase have been presented [53, 121, 122]. For example, structured abstraction, such as program sketches, are introduced to represent programs and restore syntactically correct code [122]. Moreover, the static analysis and attribute grammars (AG) [96] are

used during the code generation process to guarantee that the generated code satisfy the syntactical rules [121].

Chapter 3

Industrial Experience of Cryptographic Vulnerability Detection in Large-scale Codebases

Enterprise environment often screens large-scale (millions of lines of code) codebases with static analysis tools to find bugs and vulnerabilities. Parfait is a static code analysis tool used in Oracle to find security vulnerabilities in industrial codebases. Recently, many studies show that there are complicated cryptographic vulnerabilities caused by misusing cryptographic APIs in Java™[1]. In this paper, we describe how we realize a precise and scalable detection of these complicated cryptographic vulnerabilities based on Parfait framework. The key challenge in the detection of cryptographic vulnerabilities is the high false alarm rate caused by pseudo-influences. Pseudo-influences happen if security-irrelevant constants are used in constructing security-critical values. Static analysis is usually unable to distinguish them from hard-coded constants that expose sensitive information. We tackle this problem by specializing the backward dataflow analysis used in Parfait with refinement insights, an idea from the tool CryptoGuard [147]. We evaluate our analyzer on a comprehensive Java cryptographic vulnerability benchmark and eleven large real-world applications. The results show that the Parfait-based cryptographic vulnerability detector can find real-world crypto-

[1]Java is a registered trademark of Oracle and/or its affiliates.

graphic vulnerabilities in large-scale codebases with high true-positive rates and low runtime cost.

3.1 Introduction on Cryptographic Vulnerability Detection

To guarantee the security of large projects, companies usually deploy various bug checking tools in the development process. Parfait [56] is such a static code analysis tool designed for large-scale codebases to find security and quality defects in C/C++, Java, Python and PL/SQL languages. In particular, Parfait focuses on defects from the lists of CWE Top 25 [4] and OWASP Top 10 [5]. Cryptographic vulnerabilities caused by misusing Java Cryptographic APIs are getting more and more attention [9, 66, 72, 115, 196]. A survey shows that cryptographic API misuses dominate the cryptographic vulnerabilities, accounting for 83% in "cryptography issues" category of the Common Vulnerabilities and Exposures (CVE) database [98]. Java provides basic cryptographic objects (e.g., `Cipher`, `MessageDigest`) in Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE) libraries. Due to complex documentation and the lack of security expertise, developers may not know how to use these APIs correctly [10, 125]. Parfait covers simple cryptographic vulnerabilities like using broken Cipher or Hash algorithms. However, many studies show that cryptographic API misuses could be more complicated and involve more security rules [40, 66, 67, 115, 129, 138, 166].

Software developers and researchers have already set many cryptographic security rules. Violating them may cause various vulnerabilities including exposing sensitive information, bypassing necessary authentication, etc. Egele et al. [66] identified six types of cryptographic

API misuses that violate different security rules. Nguyen et al. [129] showed thirteen security pitfalls common in Android development and nine of them are Java cryptographic API misuses. Recently, Rahaman et al. [147] summarized sixteen common types of cryptographic API misuses in Java and developed the CryptoGuard tool to detect them. It relies on backward and forward program slicing and introduces several refinement insights to achieve high precision and scalability in large projects.

We extend Parfait to detect more Java cryptographic vulnerabilities. In particular, we need to develop a precise and scalable cryptographic API misuse detection based on Parfait's scalable data-flow analysis. In this work, we identify eleven cryptographic vulnerability types (see Table 3.1) that can be mapped to backward data-flow analysis problems. By monitoring their different vulnerable usages, we design corresponding alarm criteria for them. For example, the alarm criterion for the vulnerability "Use of a Broken or Risky Cryptographic Algorithm" is a constant matching certain weak algorithm names (e.g., "DES") while the alarm criterion for the vulnerability "Use of Password Hash With Insufficient Computational Effort" is an iteration count number less than 1000.

Cryptographic vulnerabilities are hard to be identified. Most of these vulnerabilities are caused by assigning inappropriate values (e.g., hard-coded values) to sensitive information (e.g., keys, passwords) that are required to be secret or unpredictable. To detect them, the backward data-flow analysis is used to trace all the sources influencing these security-critical variables in a program. Sources that are constants are treated as the hard-coded values and they may be reported as vulnerabilities. However, this detection process could cause many false alarms. There are many cases that involve constants in constructing a non-constant value [13]. For example, a constant string can represent a file location where the secret key is loaded. Those constants that do not impact security are called *pseudo-influences* in the work of CryptoGuard [147] which has identified five types of pseudo-influences (e.g., state

indicator) and refinement insights to reduce them. In our work, these refinement insights are further adjusted to improve the detection precision.

We build our cryptographic vulnerability detection on top of Parfait because the latter has many built-in program analysis techniques. In particular, we specialize the backward IFDS taint analysis provided by Parfait for cryptographic vulnerability detection. It allows program analysis designers to configure API methods as taint sources or sinks, and then check whether there is a data-flow from a source to a sink. In this work, we first identify the sensitive variables by setting eighteen error-prone Java cryptographic API methods (see Table 3.1) as sinks. Because ordinary taint analysis does not track constants, we further modify the taint analysis to be capable of tracking all constant sources. Moreover, we refine the taint analysis by eliminating tracing the pseudo-influences identified by the refinement rules of CryptoGuard. The refinement significantly reduces the false alarms and improves efficiency by eliminating unnecessary data-flows in an early stage. Finally, we manage to improve the scalability by leveraging Parfait's layered framework to break down the interprocedural analysis into method-level pieces and schedule them adaptively.

Our contributions are summarized as follows:

- We realize the complex Java cryptographic vulnerability detection in Oracle's Parfait platform. Specifically, we identify eleven CWE types caused by misusing Java cryptographic APIs and eighteen associated cryptographic API methods. The detection relies on a backward inter-procedural, flow-, context-, field-sensitivity data-flow analysis under Parfait/LLVM support. We design different alarm criteria to detect these cryptographic vulnerabilities.

- We specialize the backward IFDS taint analysis provided by Parfait to overcome the precision challenge caused by pseudo-influences that are security-irrelevant constants

used in constructing security-critical values. Inspired by the refinement insights in CryptoGuard [147], we define the refinement rules in the form of IFDS dataflow analysis with LLVM IR. The refined analysis significantly not only reduces false alarms but also improves scalability.

- We evaluated the precision and scalability of Parfait cryptographic vulnerability detection on the comprehensive CryptoAPI-Bench [13] and several large-scale industrial applications. The experience result demonstrates that our detection achieves a high precision (86.62%) and recall (98.40%) overall. The precision excluding the path-sensitivity test cases even reaches 100%. Parfait-based cryptographic vulnerability detection achieves 100% precision on the eleven large-scale applications. The runtime for codebases that include 2K to 1321K lines of code ranges from 2 seconds to 36 minutes and most of them can be finished within 10 minutes. We further show some noteworthy examples to help readers better understand the practices.

In summary, we have developed a precise and scalable analysis to detect cryptographic vulnerabilities. Our work incorporates the false positive reduction refinements of CryptoGuard, the scalable framework of Parfait and the IFDS analysis based on LLVM IR. The evaluation results show that our tool works well in an industrial setting.

3.2 Background

Our detection tool mainly aims at Java cryptographic API misuses. We introduce the targeted vulnerabilities as well as the background about Parfait in this section.

3.2.1 Java Cryptographic API misuses

We summarize the Java cryptographic API misuses covered by our detection from the developers' perspective, showing the involved error-prone APIs and their common vulnerable usages in Table 3.1. The involved Java classes include:

`SecureRandom` *Class.* Any nonce used in cryptography operations should be generated from `SecureRandom` instead of `Random`. Furthermore, setting a static or predictable seed via the constructors or `setSeed` methods[2] is potentially vulnerable.

`MessageDigest` *Class.* Passing a broken hash algorithm (e.g., MD5) to the `getInstance` API is vulnerable.

`Cipher` *Class.* The `getInstance` API is error-prone of using broken ciphers or insecure mode. The specific vulnerable usages include 1) passing a weak cipher algorithm (e.g., `"DES"`); 2) specifying `"ECB"` mode for a block cipher (e.g., `"AES/ECB/NoPadding"`); 3) a block cipher without explicitly specifying a mode (e.g., `"AES"`) because the vulnerable ECB mode is used by default.

`KeyStore` and *Key Specification Classes.* The APIs in `KeyStore` and the various key specification classes (e.g., `SecretKeySpec`, `PBEKeySpec`) accept secrets (e.g., passwords, key materials) through their arguments. Any method call accepting a hard-coded or predictable secret is vulnerable.

Algorithm Parameters Classes. `IvParameterSpec` and `PBEParameterSpec` classes manage the initial vector (IV), salt, and PBE iteration count. IVs and salts that are static or predictable cause vulnerabilities. Besides, the iteration count is required to be not fewer than 1000.

[2]This API has two different method signatures (setSeed(long seed) and setSeed(byte[] seed)), we skip them for simplicity.

Table 3.1: Error-prone Java Cryptographic APIs covered by Parfait's cryptographic API misuses detection and the eleven involved vulnerability types in CWE. S stands for severity level from CryptoGuard [147].

Class	Method Names	Vulnerable Usage	S	CWE
Random	constructor	used in cryptography operations	M	338: Use of Cryptographically Weak PRNG
SecureRandom	constructor setSeed	pass static or predictable seed	M	337: Predictable Seed in PRNG
MessageDigest	getInstance	pass weak algorithm	H	328: Reversible One-Way Hash
Cipher	getInstance	pass weak algorithm pass ECB mode for block ciphers	L	327: Use of a Broken or Risky Cryptographic Algorithm
KeyStore	load store setKeyEntry getKey	pass hard-coded password	H	259: Use of Hard-coded Password
SecretKeySpec	constructor	pass hard-coded key materials	H	321: Use of Hard-coded Cryptographic Key
PBEKeySpec	constructor	pass hard-coded password	H	259: Use of Hard-coded Password
		pass static or predictable salt	M	760: Use of a One-Way Hash with a Predictable Salt
		pass iteration <1000	L	916: Use of Password Hash With Insufficient Computational Effort
PBEParameterSpec	constructor	pass static or predictable salt	M	760: Use of a One-Way Hash with a Predictable Salt
		pass iteration <1000	M	916: Use of Password Hash With Insufficient Computational Effort
IvParameterSpec	constructor	pass static or predictable IV	M	329: Not Using a Random IV with CBC Mode
TrustManager	checkClientTrusted checkServerTrusted getAcceptedIssuers	override to skip validation override to skip validation override to return null	H	303: Incorrect Implementation of Authentication Algorithm
HostnameVerifier	verify	override to always return True	H	303: Incorrect Implementation of Authentication
SSLSocketFactory	createSocket	miss hostname verification	H	304: Missing Critical Step in Authentication

`javax.net.ssl` *Classes.* The methods of Java classes `TrustManager`, `HostnameVerifier` and `SSLSocketFactory` in `javax.net.ssl` package provide the SSL/TLS services. Issues usually happen when developers override the default methods or skip necessary steps to bypass proper verifications.

3.2.2 CryptoGuard

CryptoGuard [147] applies the backward and forward program slicing to discover constant sources and configurations causing Java cryptographic API misuses. It develops a set of

refined slicing algorithms to achieve high precision.

False Positive Reduction. CryptoGuard adopts five refinement insights to remove the language-specific irrelevant elements that cause false positives. During the analysis process, the state indicators (e.g., `getBytes("UTF-8")`), resource identifiers (e.g., keys of a map), bookkeeping indices (e.g., size parameters of an array), contextually incompatible constants, and constants in infeasible paths are removed by refinements conditioned on their Jimple representations.

Runtime Improvement. The most costly parts of the inter-procedural analysis are usually the iterative orthogonal explorations. CryptoGuard improves the runtime by limiting the orthogonal explorations to depth 1, whereas deeper orthogonal method calls are handled by the refinement insights.

3.2.3 Data-flow Analysis in CryptoGuard and Parfait.

Parfait includes many functionalities to support various program analyses. An important feature of Parfiat that has not appeared in CryptoGuard [147] is the IFDS analysis framework[3].

Data-flow Analysis in CryptoGuard. CryptoGuard achieves data-flow analysis based on Soot's `FlowAnalysis` library. `FlowAnalysis` includes the intra-procedural data-flow analysis that keeps a flow set and updates it along the data-flow traces as shown in Fig. 3.1(b). CryptoGuard iteratively runs its intra-procedural analysis for callee and caller methods on the call graph. However, this design might cause re-exploring callee methods multiple times. To reduce complexity, its implementation sets the default depth of the clipping callee method exploration to 1.

[3]The project Hero [37] implements the IFDS framework on top of Soot, however, CryptoGuard only uses the FlowAnalysis library in Soot, which does not provide IFDS.

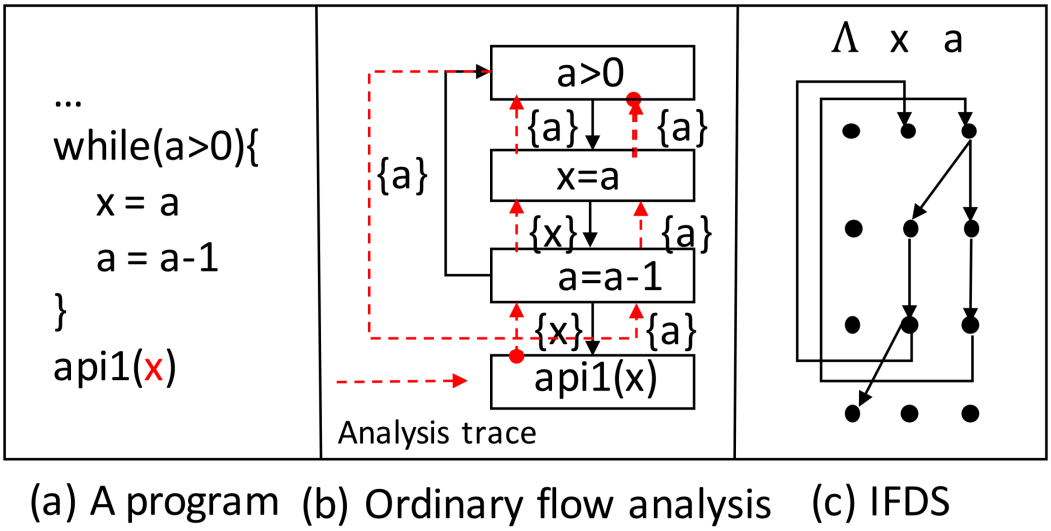


Figure 3.1: The comparison of the ordinary iterative analysis and IFDS analysis. (a) is a program demo. (b) shows the ordinary flow-set based analysis by collecting and updating a flow set. (c) shows the IFDS based analysis process which builds edges and then summarizes edges.

IFDS in Parfait. Parfait contains the data-flow analysis as well as the IFDS framework. As shown in Fig. 3.1(c), the IFDS framework handles the analysis by building edges among the data facts (i.e., variables) and summarizing the edges between two program points on the super control-flow graph. It can avoid unnecessary re-analysis as much as possible.

Parfait Framework. To improve scalability, Parfait offers a layered framework to optimize the ensemble of static program analyses. According to the time cost, the analyses are scheduled from the quickest to the slowest. In this way, more bugs can be found with a lower time overhead. Specifically, in cryptographic vulnerability detection, we dynamically schedule the analyses into different layers according to the depth of callers. More details are in Section 3.3.2.

3.3 Detection Methods and Implementation

Our detection covers all the misuses shown in Table 3.1. Two scalability enablers of it are the layered framework of Parfait and the summarization mechanism in IFDS to handle callee methods.

3.3.1 Detection Methods

The detecting logic is similar to CryptoGuard which maps the cryptographic API misuses to the data-flow analysis problems. In terms of the specific detection methods, there are three groups.

Group 1: Inter-procedural Backward Data-flow Analysis. This group includes the API misuses determined by constant sources. Specifically, these are APIs in Table 3.1 of Java Class `SecureRandom`, `MessageDigest`, `Cipher`, `KeyStore`, `SecretKeySpec`, `PBEKeySpec`, `PBEParameterSpec`, and `IvParameterSpec`. We require an inter-procedural backward data-flow analysis to capture the constant sources of the API arguments. We apply different verifying rules to the collected constant sources according to the vulnerability types. The verifying rules include whether it is a constant, whether it is a number less than 1000, or whether it matches some weak algorithms (e.g., "DES").

Group 2: Intra-procedural Pattern Matching. The vulnerabilities related to `TrustManager`, `HostnameVerifier`, and `SSLSocketFactory` in Table 3.1 belong to this group. These vulnerabilities often happen within one method that is responsible for authentication operations. We find them by the intra-procedural pattern matching. Specifically, for `HostnameVerifier`, we detect whether the return value of the method `verify` is always "True" regardless of the verification. For `TrustManager`, we detect three vulnerable patterns in the `checkClientTrusted`

and `checkServerTrusted` methods including 1) missing verification behavior; 2) catching the verification exception without throwing it; 3) missing verification under a certain path. For `SSLSocketFactory`, we perform the intra-procedural pattern matching to check whether the `HostNameVerifier.verify` method is called after the `SSLSocketFactory` instance creation.

Group 3: Sanitizer vs. Verifier. In cryptography operations, `Random` is not strong enough [1]. However, it is unreasonable to report every `Random` used in a program as a vulnerability. Therefore, we regard `Random` as a verifier and `SecureRandom` as a sanitizer for the traced arguments in group 1. Accordingly, we only report `Random` in these cryptographic usages.

3.3.2 Cryptographic Vulnerability Detection Implementation

Supported by Parfait, we implement the inter-procedural flow-, context-, and field-sensitive backward data-flow analysis for cryptographic vulnerabilities detection. Next, we introduce the specific designs of Parfait for scalability and good precision.

Layered Scheduler for Caller Methods. Parfait optimizes the analysis ensemble to improve scalability. Figure 3.2 demonstrates the backward analyses that are broken down and assigned to different layers. The analyses are scheduled layer by layer. At each layer, the backward analysis ends up at the entry point of the current method with three situations. First, a real bug is verified. Second, the potential bug is sanitized as no bug. Third, further analyses are required in its caller methods. Further analyses will be scheduled at the next layer. In this way, the analysis requiring less time can be performed first. It also avoids the duplicated parts of two potential vulnerabilities detection traces. This layered framework effectively improves the efficiency of finding bugs.

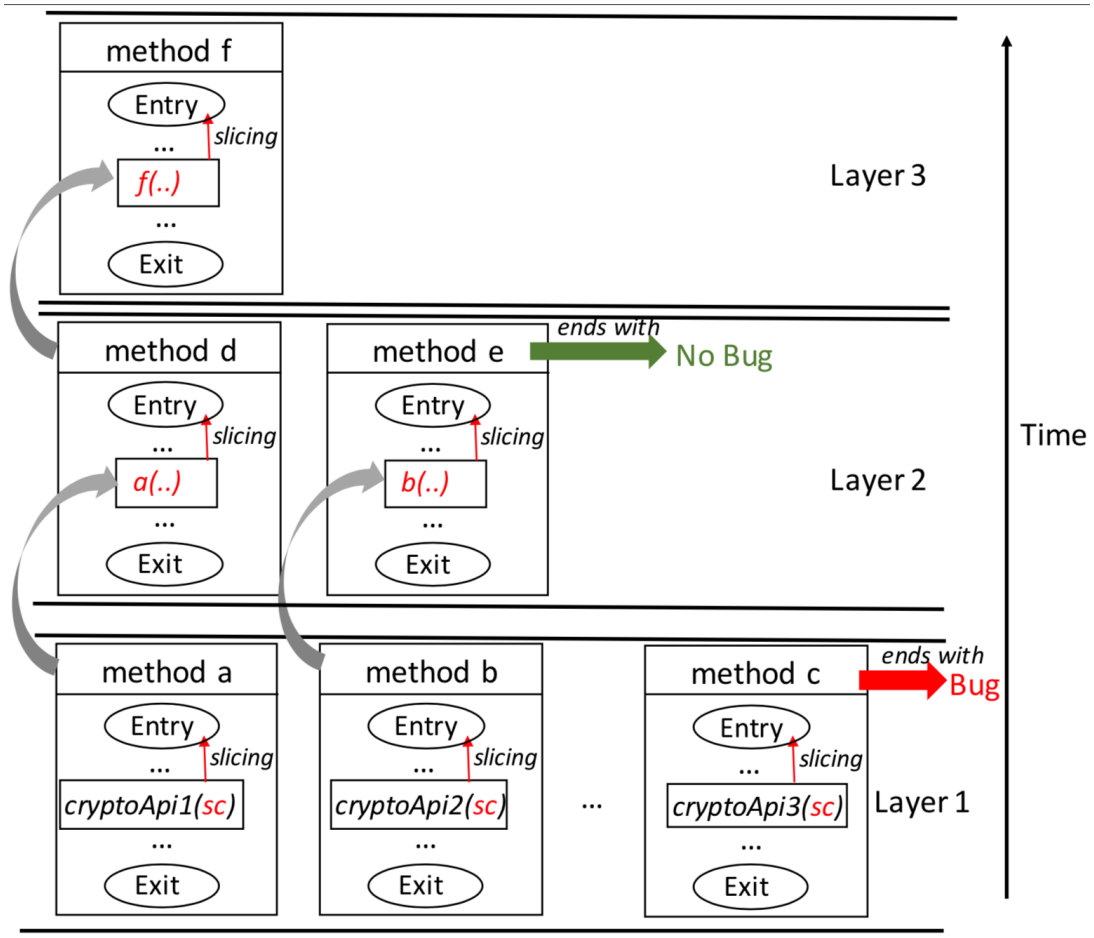


Figure 3.2: The inter-procedural analysis under Parfait's layered framework. This design is important to achieve the scalability of Parfait.

Flow Functions in IFDS. There are several flow functions used to define the analysis. In our cryptographic vulnerability detection, they are:

- `flow`: This function specifies the data-flow edges through ordinary non-call instructions. Specifically, it applies to the LLVM instructions `ReturnInst`, `LoadInst`, `StoreInst`, and `BitCastInst`.
- `phiFlow`: This function specifies the data-flow edges through the LLVM `phi` instruction.
- `returnVal`: The function specifies the data-flow edges between the `ReturnInst` of the callee method and its callsite. The summary edges of the callee method are queried at this point to handle the callee method.
- `passArgs`: The function specifies the data-flow edges between the arguments of the callee method and the parameters passed in its callsite.
- `callFlow`: The function handles the data-flow edges regardless of the callee method. Most of the refinements happen here to handle the callee method whose implementation is unavailable.

The major differences of these flow functions between the analysis for cryptographic vulnerabilities and taint analysis are the data-flow edges from constants. The cryptographic vulnerability detection covers the edges flowing out from constants and refines them according to five refinement insights, which does not happen in the taint analysis. Furthermore, cryptography vulnerability detection redefines the default data-flow edges in `callFlow`. More details are in Section 3.3.3.

Summarization for Callee Methods. Another design improving the scalability is the summarization mechanism for the callee methods. After a method is explored, the summary

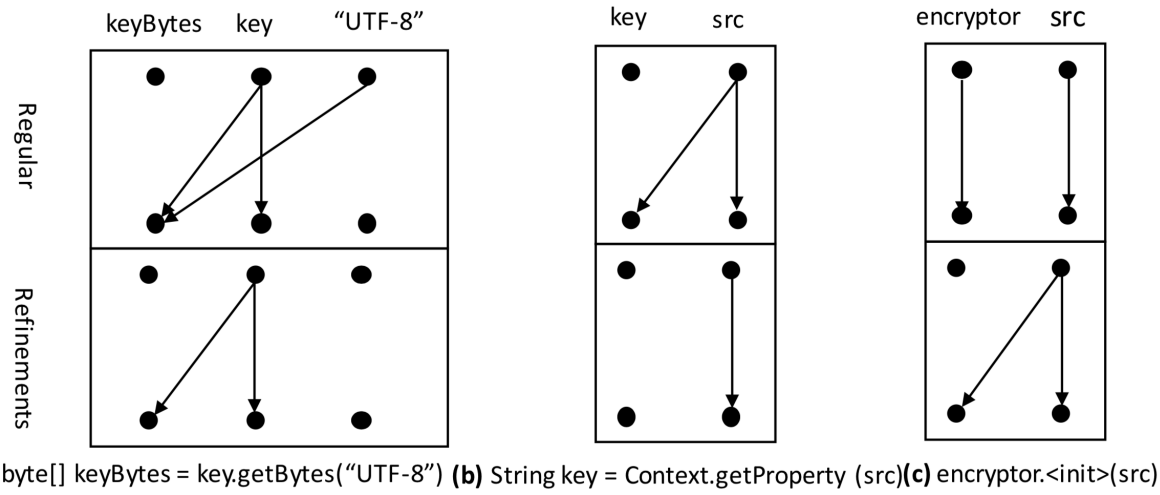


Figure 3.3: The false-positive reduction refinements represented in IFDS. It shows the dataflow propagating edges for three situations. The above one is the default propagating edges. The bottom one is the refined propagating edges.

edges for it are stored for future usage. Parfait exhaustively summarizes all methods in advance and queries the summary edges of the callee methods on demand. All the methods are summarized in a bottom-up manner according to the call graph, beginning from leaf methods to their callers. This design guarantees every method is only explored once. Hence, the re-exploration for callee methods is eliminated to avoid complexity explosion.

3.3.3 Pseudo-influences and Refined Analysis

Pseudo-influences. We use the backward dataflow analysis to capture the constants involved in constructing a security-critical value. When a constant is used to hard-code the security-critical value (e.g., secret key, password), it may cause vulnerability by exposing sensitive information. However, some constants do not have security impacts on the value, referred to as pseudo-influences. Static analysis is unable to identify them. Reporting all the captured constants as the dangerous sources leads to an extremely high false-positive rate. In the work CryptoGuard [147], the authors summarize five language-specific scenarios

that use constants without resulting in hard-coded values. These scenarios include using constants as a state indicator, resource identifier, and bookkeeping indices to retrieve the value. The contextually incompatible constants, and constants in infeasible paths are also regarded as pseudo-influences.

Refined Dataflow Analysis. We refine our dataflow analysis to exclude these pseudo-influences thus achieve good precision. According to the refinement insights from CryptoGuard, we define our pseudo-influence excluding rules in the context of IFDS algorithms and LLVM IR instructions. We select `callFlow` function in our IFDS dataflow analysis to apply the refinement rules. The reason is that most of the pseudo-influences appear as the arguments of a method call. For example, the pseudo-influence `"UTF-8"` is the argument of the method `<String: byte[] getBytes(String)>`.

In the form of IFDS, we describe the rules with the graph reachability between the data variables given an LLVM instruction. As shown in Fig. 3.3, the data flow edges are refined according to the method signature we obtained from the LLVM instruction. Specifically, there are three types of call instructions. We apply different data flow propagation rules to them. First, if the call instruction has a return value and invoking an instance method that belongs to an object, we change the default data flow propagation edges as described in Fig. 3.3 (a). The edge from the argument to the return value is eliminated because the argument is likely to be a pseudo-influences. Second, if the call instruction has a return value and invoking a static method without an associated object, we also terminate the edge from its argument to the return value to avoid pseudo-influences, as shown in Fig. 3.3 (b). Finally, if the call instruction does not have a return value and belongs to an object, we add a data flow edge from its argument to the object holder. Meanwhile, we remove the edge between the object holder itself before and after this call instruction. This allows us to stop tracing the object but tracing the argument that influences the object. The example is given

Table 3.2: Parfait's Evaluation Results on 158 Test Cases of CryptoAPI-Bench. There are basic cases (intra-procedural), different inter-procedural cases that require across methods, across classes, field sensitivity, path-sensitivity, and heuristics to handle. FPs represents false positives. FNs represents false negatives.

Type	Total	Insecure	Secure	Reported	FPs	FNs	Precision	Recall
Basic cases	27	24	3	24	0	0	100%	100%
Multiple methods	57	56	1	54	0	2	100%	96.43%
Multiple classes	23	18	5	18	0	0	100%	100%
Field sensitivity	19	18	1	18	0	0	100%	100%
Path sensitivity	19	0	19	19	19	0	0 %	0 %
Heuristics	13	9	4	9	0	0	100%	100%
Total	**158**	**125**	**33**	**142**	**19**	**2**	**86.62%**	**98.40%**

in Fig. 3.3 (c).

3.4 Accuracy Analysis and Real-world Findings

We have tested our cryptographic vulnerability detection on a comprehensive cryptographic vulnerability benchmark (CryptoAPI-Bench [13]) to evaluate the precision and recall. To learn its scalability, we further perform experiments by scanning eleven large real-world codebases to obtain the runtime performance.

3.4.1 Accuracy Analysis on CryptoAPI-Bench

We have tested Parfait on 158 test cases in CryptoAPI-Bench [13]. CryptoAPI-Bench includes various kinds of test units from basic ones to more advanced cases. The basic test cases only require intra-procedural analysis to handle. The advanced cases are inter-procedural ones that require the analyses across multiple methods, multiple classes, achieving field sensitivity, and path sensitivity.

Table 3.3: False positive reduction derived from applying the refinement insights (RIs). We compare Parfait cryptographic vulnerability detection with its intermediate version without the refinement insights.

Type	# of Vulnerabilities	FPs (w/o RIs)	FPs (with RIs)	Reduction
Basic cases	24	1	0	100%
Multiple methods	56	3	0	100%
Multiple classes	18	1	0	100%
Field sensitivity	18	2	0	100%
Path sensitivity	0	19	19	0
Heuristics	9	12	0	100%
Total	**125**	**38**	**19**	**50%**

The breakdown numbers are shown in Table 3.2. The overall precision and recall are 86.62% and 98.40%, respectively. All the false positive cases come from path sensitivity cases, which verifies that our tool has achieved high precision for the cases excluding path-sensitive ones. We analyze several examples to further reveal the details of Parfait cryptographic vulnerability detection and discuss possible improvements.

Impact of Refinement Insights. We demonstrate the impact of our refinement insights by comparing the Parfait cryptographic vulnerability detection with its intermediate version that does not have the refinement strategies. Table 3.3 shows the comparison. Without the refinements, there are 38 false positive cases. Based on our manual analysis, most of the false positives are caused by the pseudo-influences we introduced in Section 3.3.3. The refinement insights successfully reduce all the false positive cases except for the path-sensitive case.

3.4.2 Evaluation on Real World Projects

We evaluated our tool on eleven real world codebases. Nine of them are Oracle while two are open-source projects Spring-Security[4] and Orchid[5]. We select these projects because they

[4]https://github.com/spring-projects/spring-security
[5]https://github.com/OrchidTechnologies/orchid

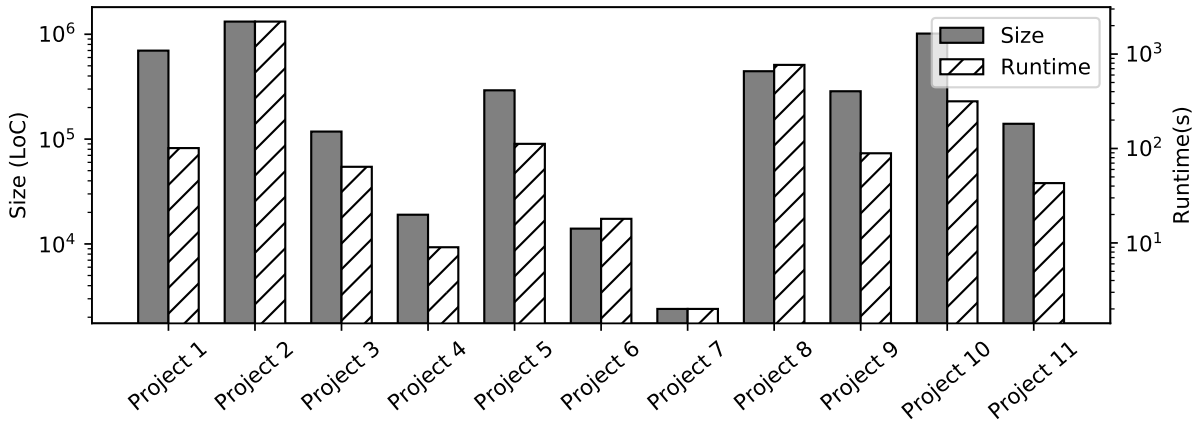


Figure 3.4: Runtime performance of Parfait for screening the eleven real-world codebases. The size shows how many lines of code these codebases have.

are security-relevant and use Java cryptographic APIs.

Runtime and Precision

Scalability is always one of the most important concerns. We list the runtime performance and the size of these scanned projects in Fig. 3.4. The project sizes vary from 2K to 1321K. The detection is run on the machine with Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz, 128G memory, and Oracle Linux Server release 6.9 operating system. The results show that Parfait achieves excellent scalability. The analysis can be finished within 10 minutes for the majority of these projects including those with millions of lines of code (Project 10).

Fig. 3.5 demonstrates the precision results of Parfait and CryptoGuard on the eleven real-world projects. Compared with CryptoGuard, Parfait successfully identified more true positive cases with fewer false positives. Parfait reported 42 vulnerabilities and all of them are manually verified as true positives. The precision reaches 100%. We show several real-world vulnerabilities found by Parfait in Section 3.4.2 CryptoGuard reported 69 vulnerabilities. However, there are 47 false positives among them. The precision is 31.88%. We noticed

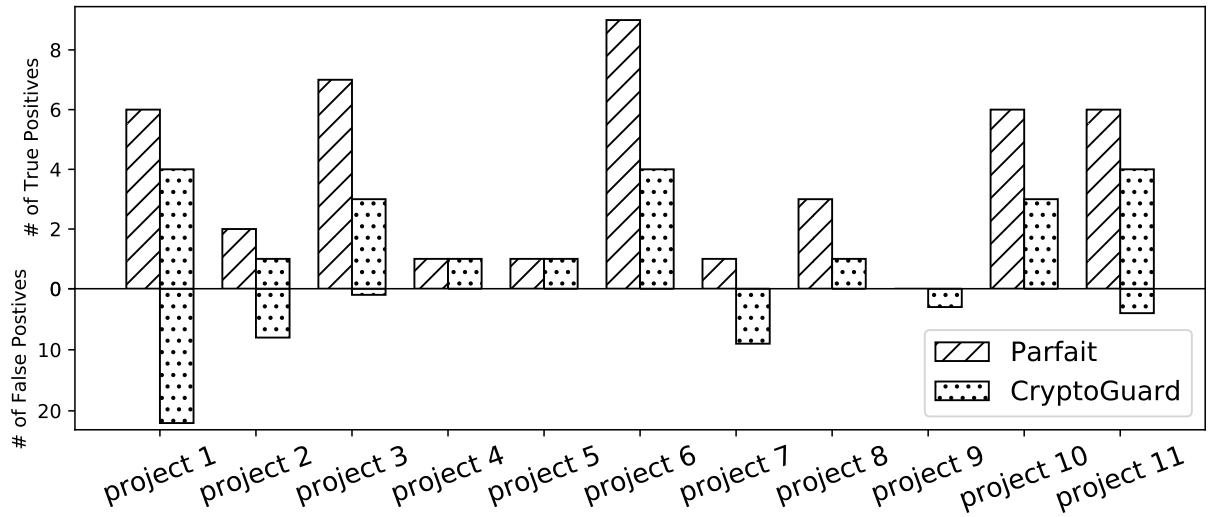


Figure 3.5: The number of vulnerabilities reported by Parfait and CryptoGuard in the eleven real-world industrial applications. The upper area of the x axis shows the true positive alerts while the bottom area of the x axis shows the false positive alerts. Nine of them are internal codebases of Oracle. Two of them are open-source projects.

that all the false positive cases of CryptoGuard are caused by the same issue, that is, how CryptoGuard detects weak Pseudo-random Number Generator (PRNG) vulnerabilities. We noticed that all the false positives of CryptoGuard are caused by the same issue, that is, how CryptoGuard identifies weak PRNG cases. We will discuss it in the comparison between CryptoGuard and Parfait.

Comparison with CryptoGuard. As we introduced, CryptoGuard and Parfait leverage identical refined dataflow analysis at a high level to detect cryptographic vulnerabilities. Here, we analyze the differences between them in detection results.

Detection for Weak PRNG. A major difference between Parfait and CryptoGuard is the way they identify weak PRNG vulnerabilities. After manual analysis, we noticed that all the false positives of CryptoGuard shown in Fig. 3.5 are weak PRNG cases. To make it more clear, we break down the reported cases into weak PRNG cases and other types of vulnerabilities, as shown in Fig. 3.6. Overall, there are 48 weak PRNG vulnerabilities and 21 other types

of vulnerabilities reported by CryptoGuard. Among the 48 weak PRNG cases, only 1 of them is verified as a true positive case. As a contrast, Parfait reported 0 weak PRNG case, which indicates that Parfait missed at least 1 weak PRNG vulnerability. This suggests that CryptoGuard tends to have a more conservative approximation on weak PRNG vulnerability detection while Parfait reports this type of vulnerability in a more precise approximation.

Listing 3.1 shows a false positive weak PRNG identified by CryptoGuard. The Java class `Random` is not strong enough, therefore, an alternative class `SecureRandom` that is cryptographically strong is recommended to use. However, our manual verification confirmed that the `Random` instance is not used in a security or cryptographic context. Hence, we consider it is a false positive as there is no impact on security. CryptoGuard performs an exhaustive search in the codebase to report every `Random` usage regardless of the context. Hence, there are many false positives. On the opposite, Parfait applies a more strict criterion for alerting this type of vulnerability. Only when the `Random` instance is passed to the cryptographic APIs covered in Table 3.1, will it be reported as a weak PRNG case. However, this may miss some cases due to the limited coverage of the cryptographic APIs. It is difficult to accurately determine whether a `Random` instance is used for cryptographic purposes. Identifying more vulnerable usage patterns and involved cryptographic APIs for this type of vulnerability can be future work. To extend the current detection criteria, Parfait provides the flexibility for users to change the sinks, sources, sanitizers, and verifiers of the dataflow analysis through configuration, which makes customizing the vulnerability detection rules easy.

```
1 Random random = new Random();
2 int rnumber = random.nextInt();
```

Listing 3.1: A reported weak PRNG vulnerability that is a false positive

Exploration Depth for Callee Methods. Another difference between Parfait and CryptoGuard is the exploration depth for callee methods when performing interprocedural analysis. The

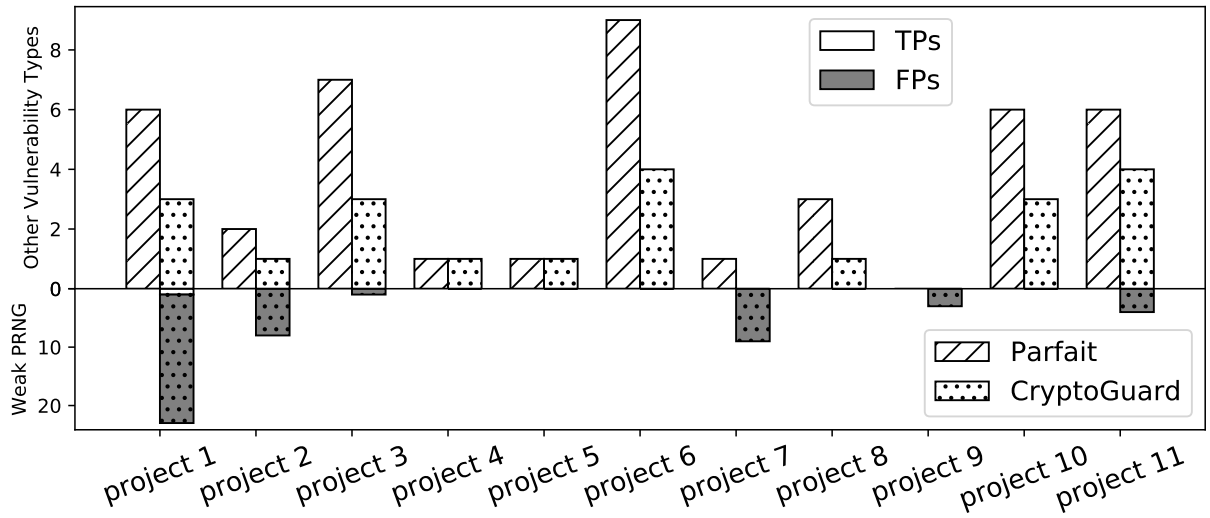


Figure 3.6: The number of vulnerabilities reported by Parfait and CryptoGuard in the eleven real-world industrial applications. We break them down into the weak PRNG vulnerabilities and the other vulnerability types. Parfait reported 0 weak PRNG vulnerability. CryptoGuard reported 48 weak PRNG vulnerabilities while only 1 of them is a true positive case. Parfait and CryptoGuard both achieves 100% precision in the other vulnerability types excluding the weak PRNG cases.

interprocedural dataflow analysis requires exploring the encountered callee methods. When meeting the recursive callee methods or the callee stack is too deep, the analysis needs to clip the call graph. CryptoGuard allows users to configure the exploration callee stack depth. To make the analysis fast, CryptoGuard set the default callee stack depth as 1. Parfait deals with this problem by summarization mechanism (see details in Section 3.3.2). This design avoids clipping the callee stack, however, the price is that the summarization becomes the most costly part. To make the summarization as a one-time cost, it is performed separately in advanced and stored for queries when encountering a callee method in the dataflow analysis. In Fig. 3.5, we observe that CryptoGuard missed 21 cases that have been reported by Parfait. This might be attributed to the limited default callee stack depth that CryptoGuard explores. It can be improved by setting a larger value of the callee stack depth.

Application Perspective vs. Library Perspective. Parfait differs from CryptoGuard in the

vulnerability definitions in some situations. An example is given in Listing 3.2. If the potentially vulnerable method is not called in the scanned codebase, the concerned field variable is left undetermined and then Parfait considers it as a non-vulnerable case. However, CryptoGuard applies a forward slicing for this field variable to find out the possible assignments in the initialization. If a constant is assigned in the initialization, CryptoGuard still considers it as a vulnerability. If the detected issues are in applications, Parfait's design is superior because it avoids overestimating the vulnerabilities. If they are in libraries, CryptoGuard's design is better as it discovers the potential buggy method although they may not be called yet.

```
1  public class PredictableCryptographicKeyABSCase1 {
2      Crypto crypto;
3      public PredictableCryptographicKeyABSCase1() throws Exception {
4          String passKey = PredictableCryptographicKeyABSCase1.getKey("pass.key");
5          if(passKey == null) {
6              crypto = new Crypto("defaultkey");
7          }
8          crypto = new Crypto(passKey);
9      }
10     //this.crypto.defaultKey-->secret key; no caller for encryptPass, terminate
11     public byte[] encryptPass(String pass, String src) throws Exception {
12         String keyStr = PredictableCryptographicKeyABSCase1.getKey(src);
13         return crypto.method1(pass, keyStr);
14         //keyStr-->secret key; this.crypto.defaultKey-->secret key
15     }
16     public static String getKey(String s) {
17         return System.getProperty(s);
18     }
19 }
20 class Crypto {
```

```
21      Cipher cipher;
22      String algoSpec = "AES/CBC/PKCS5Padding";
23      String algo = "AES";
24      String defaultKey;
25      public Crypto(String defkey) throws NoSuchPaddingException,
        NoSuchAlgorithmException {
26          cipher = Cipher.getInstance(algoSpec);
27          defaultKey = defkey;
28      }
29      //key-->secret key; this.defaultKey-->secret key
30      public byte[] method1(String txt, String key) throws UnsupportedEncodingException
        , InvalidKeyException, BadPaddingException, IllegalBlockSizeException {
31          if(key.isEmpty()){
32              key = defaultKey;
33          }
34          byte[] keyBytes = key.getBytes("UTF-8");
35          byte [] txtBytes = txt.getBytes();
36          keyBytes = Arrays.copyOf(keyBytes,16);
37          SecretKeySpec keySpec = new SecretKeySpec(keyBytes,algo); //A potential bug
38          cipher.init(Cipher.ENCRYPT_MODE,keySpec);
39          return cipher.doFinal(txtBytes);
40      }
41  }
```

Listing 3.2: A test cases considered non-vulnerable by Parfait but vulnerable by CryptoGuard. The backward analysis in Parfait terminates at Line 11 and leaves this.crypto.defaultKey as a variable due to no caller of this method.

```
1 public class DesEncrypter{
2   private byte[] salt = { (byte) 0xC9, (byte) 0xDB, (byte) 0xA3, (byte) 0x52, (byte)
      0x56, (byte) 0x35, (byte) 0xE8, (byte) 0xB0};
3   private int iterationCount = 20;
4     public DesEncrypter(final String passPhrase){
5         initDesEncrypter(passPhrase);}
6     private void initDesEncrypter(final String passPhrase){
7     ...
8     AlgorithmParameterSpec paramSpec = new PBEParameterSpec(salt,iterationCount);}}
```

Listing 3.3: A real-world vulnerability about using constant salt and insufficient iteration count (We modified the code to make the codebase unidentifiable.)

Real-world Findings

We have reported the detected vulnerabilities to corresponding developers. In terms of the open-source projects, we further find that the vulnerabilities are either in their non-production (development) mode or fixed in their latest versions. We show several real-world detected cases below.

Listing 3.3 shows vulnerabilities of using constant salt and insufficient iteration count as PBE parameters. This case represents the most common vulnerable pattern of the sensitive cryptographic materials (e.g., passwords, salts, IVs, etc) to be hard-coded in the initialization.

```
1 public String padding_salts(String salts){
2     StringBuffer sb = new StringBuffer();
3     for(int i=salts.getBytes().length; i<16; i++){
4       sb.append((byte)i&0xfe)}
5     String padded_salts = salts+sb.toString();
6     return padded_salts;}
```

Listing 3.4: A real-world vulnerability about insufficient entropy salts

Listing 3.4 is a noteworthy real-world example. It introduces a vulnerability of using salts

with insufficient entropy. When a random salt is iteratively assigned by the same variable, its value space is reduced significantly and hence makes the exhaustive attack feasible. Our analysis reports a constant number 16 at Line 3 involved in the construction of the salts. However, to accurately capture the insufficient entropy issue, symbolic execution is required.

```
1 public SecureRandom getObject() throws Exception{
2     SecureRandom rnd = SecureRandom.getInstance(algorithm);
3     if(seed != null){
4       byte[] seedBytes = FileCopyUtils.copyToByteArray(seed.InputStream());
5         rnd.setSeed(seedBytes); //manual seeding
6     }else{
7         rnd.nextBytes(new byte[1]) //self-seeding
8     }}
```

Listing 3.5: An example from CVE-2019-3795

Listing 3.5 shows a detected vulnerability in the open-source project Spring Security, disclosed as CVE-2019-3795 [3]. This vulnerability appears in Spring Security versions 4.2.x before 4.2.12, 5.0.x before 5.0.12, and 5.1.x before 5.1.5. Although not involving a hard-coded seed, the `SecureRandom` instance relies on an unreliable `InputStream` at Line 4 as the `seed`. Inspired by this real-world vulnerability, we apply a more strict rule for `SecureRandom.setSeed` to avoid unreliable seeding. Only self-seeding and manual seeding by the method `SecureRandom.generateSeed()` are considered as secure. A self-seeding (secure) will be automatically enforced if the API `nextBytes` is called immediately after the `SecureRandom` instantiation [2].

Listing 3.6 shows a reported case for bypassing certificate verification. This case disables the certificate verification by simply throwing the `UnsupportedOperationException` for all certificates. This misuse, matching a vulnerable pattern, was reported, however it is not enabled in the production code path, and hence not exploitable or requiring any remediation.

```
1 public void checkClientTrusted(X509Certificate[] certs, String authType) throws
    CertificateException{
2    throw new UnsupportedOperationException("checkclientTrusted is unsupported in "+
    this.getClass().getName());}
```

Listing 3.6: A real-world false positive case about TrustManager

Examples in CryptoAPI-Bench

We show the code snippets that are the false positives or false negatives of our tool in CryptoAPI-Bench. There are several issues that are not covered by our detection design. First, our detection does not have path sensitivity. Listing 3.7 shows an example that requires path sensitivity to handle. We reported this as a vulnerability because the constant "defaultkey" is passed to the variable `keyBytes` in Line 10 through a control flow path without entering the `if` branch. However, since the `if` condition will always be satisfied. This path is unrealizable, which causes the false alarm.

```
1     String defaultKey = "defaultkey";
2     int choice = 2;
3     byte[] keyBytes = defaultKey.getBytes();
4     //keyBytes-->key material after phiFLow
5     if(choice>1){
6         //nothing-->key material
7         SecureRandom random = new SecureRandom();
8         keyBytes = String.valueOf(random.ints()).getBytes();
9     }
10    keyBytes = Arrays.copyOf(keyBytes,16);
11    SecretKeySpec keySpec = new SecretKeySpec(keyBytes, "AES");
```

Listing 3.7: A false positive caused by path sensitivity

Moreover, our detection may miss some cases due to the string manipulation. As shown in Listing 3.8, the tool captures the source "20" that is used to generate the iteration count. Our tool will compare the value of the captured source with the minimal required interation count 1000. However, since the developer uses a `string` "20" instead of the `int`, it escapes from our detection condition.

```
1  public class LessThan1000IterationPBEABICase2 {
2      public static final String DEFAULT_COUNT = "20";
3      private static char[] COUNT;
4      private static char[] count;
5      public static void main(){ //Bug condition: "20"<1000?
6          LessThan1000IterationPBEABICase2 lt = new LessThan1000IterationPBEABICase2();
7          go2();    //"20"-->PBE iteration
8          go3();    //this.COUNT-->PBE iteration
9          lt.key2(); //this.count-->PBE iteration
10     }
11     private static void go2(){
12         COUNT = DEFAULT_COUNT.toCharArray();
13     }
14     private static void go3(){
15         count = COUNT;
16     }
17     public void key2(){ //this.count-->PBE iteration
18         ...
19         pbeParamSpec = new PBEParameterSpec(salt,
       Integer.parseInt(String.valueOf(count)));
20     }
21 }
```

Listing 3.8: A false negative case caused due to type matching

3.4.3 Discussion

We discuss the potential improvement and limitations of Parfait.

Potential Improvement. There are two potential improvements to fix the false-negative cases. First, a false negative could be caused by missing the summarization for `clinit` method. An example is shown in Listing 3.9. This deficiency is derived from the fact that `clinit` has not appeared in Parfait's call graph. A fix for this issue could be updating the call graph construction to cover the `clinit` of every class. Second, a false-negative case shown in Listing 3.8 is caused by incompatible types between the captured source (i.e., `String`) and the sensitive argument (i.e., `int`). This corner case can be improved by checking the type compatibility through the type casting in Java language.

```
1  public class PredictablePBEPasswordABICase2 {
2  public static String KEY = "sagar";
3   public static char [] DEFAULT_ENCRYPT_KEY = KEY.toCharArray(); //"sagar"-->this.
    DEFAULT_ENCRYPT_KEY happens in clinit
4   private static char[] encryptKey;
5   ...
6   public static void main(String [] args) { //this.DEFAULT_ENCRYPT_KEY-->PBE
    password
7       ...
8   }
9  }
```

Listing 3.9: A false negative case caused due to the summarization

Limitations. Our cryptographic vulnerability detection still has limitations with handling path-sensitive cases and pointer issues. We show a path-sensitive false-positive case in Listing 3.7. Furthermore, another potential cause for false positives could be pointer issues.

Due to the limitation of static analysis, there may be over-approximation in our call graph construction, which leads to potential false positives. However, path-sensitivity and pointer precision are too costly, in our experience, for large codebases. Our analysis is designed to scan large-scale industrial projects, therefore we accept the trade-off for better overall performance.

3.5 Conclusion and Future Work

We have implemented a precise and scalable cryptographic vulnerability detection in the framework of Parfait. Leveraging the refinement insights from CryptoGuard, our detection reproduced the high precision results (few or no false positives) achieved by CryptoGuard. Experiments show 100% precision for eleven real-world large-scale projects and CryptoAPI-Bench excluding the path-sensitivity cases. Benefited from the IFDS and layered framework of Parfait, our cryptographic vulnerability detection also achieves good runtime performance for large-scale codebases. The runtime for these eleven large-scale codebases ranges from 2 seconds to 36 minutes. Most of them can be screened within 10 minutes.

Based on the backward data-flow analysis, our Parfait-based cryptographic vulnerability detection covers eleven cryptographic vulnerability types. There are a few vulnerability types that require the combination of backward and forward data-flow analysis. According to our findings in Section 3.4.2, there are some complicated real-world cases that may be dealt with by other program analysis techniques, e.g. symbolic execution. That would be our future work. Besides, how to accurately handle the Weak PRNG vulnerabilities is also an interesting future direction.

Chapter 4

Neural Network based Code Embedding

In this work, we conduct a measurement study to comprehensively compare the accuracy impacts of multiple embedding options in cryptographic API completion tasks. Embedding is the process of automatically learning vector representations of program elements. Our measurement focuses on design choices of three important aspects, *program analysis preprocessing*, *token-level embedding*, and *sequence-level embedding*. Our findings show that program analysis is necessary even under advanced embedding. The results show 36.10% accuracy improvement on average when program analysis preprocessing is applied to transfer byte code sequences into API dependence paths. With program analysis and the token-level embedding training, the embedding *dep2vec* improves the task accuracy from 55.80% to 92.04%. Moreover, only slight accuracy advantage (0.55% on average) is observed by training the expensive sequence-level embedding compared with the token-level embedding. Our experiments also suggest the differences made by the data. In the cross-app learning setup and a data scarcity scenario, sequence-level embedding is more necessary and results in a more obvious accuracy improvement (5.10%).

4.1 Introduction on Code Embedding

Code embedding refers to the process of transforming the program elements into continuous vectors [25, 86, 197]. This transformation is important for deep learning, as the subsequent model training and inference are performed on the embedding vectors [43, 63, 141**?**]. Despite much progress in this area [25, 64, 68, 86, 132, 197], it is still unclear the effectiveness and advantages of different embedding designs. A side-by-side comparison would help one better design neural network based methodologies and harness their power for embedding-based applications.

Our work addresses a foundational question in AI-based software engineering. We conduct comprehensive comparative experiments to uncover the impacts of multiple embedding design choices in a specific application scenario, cryptographic API completion. We choose this scenario because API completion is a basic building block for many software engineering tasks, including code repair, and code generation. It aims to predict the next API method given the previous code sequence. Moreover, cryptographic APIs are widely known as error-prone [115, 147]. By experimenting on these challenging APIs, we observe the accuracy impacts of different embedding choices.

To produce the vectors for training, an embedding solution usually includes three key steps. First, programs are preprocessed into certain representations (e.g., byte code, control flow graphs) to deliver meaningful features. This is usually achieved by program analysis techniques. Based on the preprocessed representations, a basic embedding training can be performed. It vectorizes every single token by gathering its context information across the entire corpus, which is referred to as *token-level embedding*. Beyond embedding a single token, an extra step could be conducted to produce embedding vectors for a given sequence, which is called *sequence-level embedding* in this work. It requires an extra sequence model pretraining

compared with the basic token-level embedding. Therefore, based on our extensive literature review, we identify design choices of the three main aspects – **i)** *program analysis preprocessing*, **ii)** *token-level embedding*, and **iii)** *sequence-level embedding* to compare, as shown in Table 4.1. Such comparison is missing in the literature and needs to be systematically performed.

Our **first comparison group** focuses on the impacts of program analysis preprocessing. Program analysis is often used to process programs before embedding [23, 42, 85, 195]. This preprocessing is important as it decides what information is used for embedding training. For example, Henkel *et al.* [86] extract symbolic traces for embedding while the state-of-the-art code embeddings (e.g., GraphCodeBERT [78], inst2vec [31]) leverage data flows from graph representations to embed program elements. In our work, we compare three program representations, byte code, program slices, and API dependence paths, obtained with different program analysis strategies for embedding. We explain why the three representations are selected in Section 4.3.1.

Our **second comparison group** examines the impacts of token-level embedding. We make comparisons between token-level embedding and the one-hot encoding baseline. One-hot encoding is a basic vectorization approach that indexes N tokens and represents the i-th token by an N-dimensional vector that includes a single 1 at the i-th dimension and 0s for other dimensions. Compared with it, token-level embedding, such as word2vec [117, 118, 119], is expected to result in low-dimensional semantic-aware vectors that could benefit the downstream task training. By the experimental comparison, we observe how much accuracy improvement the token-level embedding can gain.

Our **third comparison group** learns the impacts of sequence-level embedding (also called contextualized embedding). We make comparisons between sequence-level embeddings and token-level embeddings. Compared with token-level embedding, sequence-level embedding

is more advanced because the polysemy issue is handled, by assigning different vectors for different occurrences of a token. However, it also requires an extra expensive sequence language model and pretraining process to achieve that. For example, the state-of-the-art natural language sequence-level embedding BERT [63] is obtained by pretraining the Transformer [174] neural network. Our experimental comparisons aim to answer at what level the advantage of the sequence-level embedding is over the token-level embedding.

To evaluate embeddings with different design choices, we perform API completion tasks on our Java cryptographic API benchmark. Our benchmark is composed of Java cryptographic code collected from 79,887 Android Apps. To follow FSE's open science policy and for verifiability, our Java cryptographic API benchmark is publicly available on GitHub [1].

Next, we explain our research questions along with the comparative experiments designed to answer them.

RQ1: What is the accuracy impacts of token-level embedding obtained from byte code, slices, and API dependence paths in crypto API completion, respectively? To answer this question, we pretrain three token-level embeddings, *byte2vec*, *slice2vec*, and *dep2vec* on byte code, slices, and API dependence paths, respectively. Byte code, program slices, and API dependence paths are the outcome of different program analysis preprocessing. The obtained embeddings are compared with the basic setting, one-hot encoding, with corresponding program analysis preprocessing.

RQ2: What is the accuracy impacts of sequence-level embedding obtained from byte code, slices, and API dependence paths in crypto API completion, respectively? To answer this question, we pretraining three sequence-level embeddings, *byteBERT*, *sliceBERT*, and *depBERT* on byte code, slices, and API dependence, respectively.

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

Table 4.1: The overview of our comparative settings. Each cell shows the embedding and machine learning model we use for the cryptographic API completion. We have comparisons between three program analysis preprocessed sequences, token-level embeddings vs. one-hot, and sequence-level embeddings vs. token-level embeddings.

	Program analysis preprocessing		
	Byte code	Program slices	API dependence paths
Token-level embedding	byte2vec vs. one-hot (w/ LSTM)	slice2vec vs. one-hot (w/ LSTM)	dep2vec vs. one-hot (w/ LSTM)
Sequence-level embedding	byteBERT vs. byte2vec (w/ Transformer)	sliceBERT vs. slice2vec (w/ Transformer)	depBERT vs. dep2vec (w/ Transformer)

They are finetuned for the cryptographic API completion and compared with an identical Transformer neural network without the pretraining knowledge.

RQ3: Are the embedding effective for crypto API completion on new apps? To answer this, we perform the experiments not only under the basic within-app setting, but also under the cross-app setting. In the within-app setting, sequences are extracted from Android apps and randomly split for training and testing. In the cross-app setting, new Android apps are used to test the model.

RQ4: How well is the state-of-the-art general purpose code embedding applied for Crypto API completion? Besides the program analysis and embedding choices we covered in Table 4.1, we further evaluate two state-of-the-art code embedding GraphCodeBert [78] and CodeBert [68] in our cryptographic API completion. They are two general purpose source code embedding models pretrained by Microsoft on six programming languages paired with natural language. We finetune the two pretrained models for our API completion task and forms an end-to-end comparison.

Our major findings include:

- Our findings show that program analysis preprocessing plays a significant role in

Crypto API embedding and completion. For both token-level embedding and sequence-level embedding, the API dependence paths produce higher prediction accuracy, compared with slices and byte code. With program analysis, the token-level embedding *dep2vec* achieves an accuracy 36% higher than *byte2vec*. The sequence-level embedding *depBERT* achieves an accuracy 45.86% higher than *byteBERT* without program analysis preprocessing.

- Our findings show that applying embeddings with program analysis significantly improves the task accuracy compared with the one-hot baseline (no embedding). On dependence paths, the token-level embedding *dep2vec* and sequence-level embedding *depBERT* both outperform the one-hot encoding baseline by the accuracy boost of 6% and 7%, respectively. Although sequence-level embedding is slightly (0.55%) better than token-level embedding in our experiments. Considering the expensive cost of sequence-level embedding, token-level embedding is more recommended according to it.

- Our findings show that the improvements derived from program analysis and embedding is valid for cryptographic API completion on new apps. In cross-app learning scenario, the program analysis guided embedding *depBERT* and *dep2vec* still achieve good accuracy at 95.75% and. 93.58%, respectively. Another observation is the advantage of *depBERT* over *dep2vec* is slightly more obvious by the 2.17% accuracy boost compared with 0.55% in the basic setting. The sequence-level embedding *depBERT* is most recommended to be used in the data scarce situation, as the largest improvement (5.10%) of *depBERT* compared with *dep2vec* is observed on the smallest task dataset with 26,357 dependence paths.

- The state-of-the-art general purpose source code embedding solutions GraphCodeBert and CodeBert are insufficient in our cryptographic API completion tasks with the low

accuracy 59.94%. Experiments still shows the advantage of applying program analysis preprocessing in their embedding solutions. GraphCodeBert substantially outperform its non-program-analysis counterpart CodeBert by accuracy boost of 20.07% on average. The experiments also suggest the method-level context is more recommended than the class-level context for Cryptographic API completion.

Significance of research contributions. Our work provides the first quantitative and systematic comparison of the prediction accuracy of multiple API embedding approaches for neural network based code completion. Our rigorous experiments provide new empirical results that have not been previously reported, including how various domain-specific program analyses improve data-driven predictions. These quantitative findings, together with the new cryptographic API benchmark, help guide and design more powerful and accurate code completion solutions, leading to high quality and low vulnerability software projects in practice. In addition, our measurement methodology can be generalized to other types of APIs, beyond the specific cryptographic setting.

4.2 Code Embedding Background

We provide the background of embedding and the cryptographic API completion task. We categorize embedding vectors into token-level embeddings and sequence-level embeddings.

4.2.1 Token-level Embedding

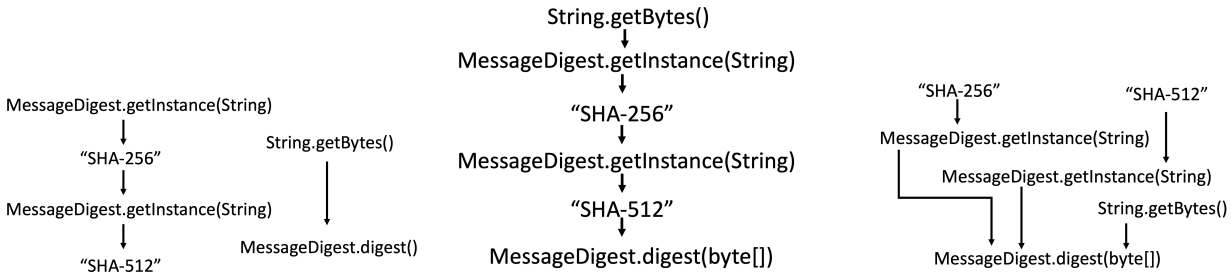
Token-level embeddings, such as word2vec [117, 118, 119], FastText [38, 92], Glove [140], assign one numeric vector to represent a token. In our work, we follow the skip-gram [117, 118] algorithm to train token-level embeddings for API methods and constants. Specifically,

a three-layer linear neural network is used to automatically learn the embeddings of all tokens in an embedding sequence corpus. The token (API method or constant) to be embedded is the input, and the tokens before and after it within a sliding window are used as the labels to train the neural network. During the embedding process, the entire embedding corpus is scanned and all the tokens and their neighbors are used for training. After that, the latent vector at the hidden layer is kept as the embedding of the input token. In this way, a token's embedding vector is determined by the statistics of its neighboring tokens in a large corpus.

4.2.2 Sequence-level Embedding

Sequence-level embedding assigns a vector for every occurrence of a token. In other words, a token is represented with different vectors when it appears in different sequences. To generate this contextualized vector, not only the token itself but also other tokens in a given sequence are used. There is a neural network based language model to take a sequence as input and output the embedding vectors of every token in this sequence. For example, the GPT family [145], BERT [63], RoBERTa [108], are sequence-level embeddings generated from Transformer neural networks. The sequence-level embedding ELMo [141] is generated from a BiLSTM neural network. This neural network is pretrained with carefully crafted tasks for producing the sequence-level embedding. Therefore, it is also referred to as a pretrained language model. The sequence-level embedding of a token is dynamically generated by the pretrained language model.

To apply the sequence-level embeddings for downstream tasks, a common way is to use the pretrained model that produces sequence-level embedding as the initial states. An extra application layer is added after the embedding layer and the entire model is fine-tuned with extra data for a specific downstream task.



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

Figure 4.1: API and constant sequences from byte code, program slices and API dependency graphs

4.2.3 Cryptographic API Completion

We evaluate different embeddings in cryptographic API completion. API completion refers to a task that suggests one or more next API methods given a preceding sequence of API elements (i.e., *API methods and constants*). We define two types of cryptographic API completion tasks, i.e., *next API completion* and *next API sequence completion*. The former aims to predict one API method in the next line while the latter produces a sequence of API methods to invoke sequentially.

4.3 Embedding Measurement Setting

We perform comparative experiments to answer our research questions. As shown in Table 4.1, we compare different design choices of *program analysis preprocessing*, *token-level embedding*, and *sequence-level embedding*.

4.3.1 Program Analysis Preprocessing Strategies

We examine the impacts of using program analysis to guide the embedding. There could be unlimited program analysis strategies to extract different program sequences. Specifically, we compare three types of program sequences: *i) byte code*, *ii) program slices*, and *iii) API dependence paths*. The byte code is from Android Apps without program analysis. The program slices are obtained by conducting interprocedural backward slicing on byte code. Moreover, the API dependence paths are extracted from API dependence graphs we construct on program slices with the dataflow dependence between the API calls. We select the three because they embody the increasing levels of program analysis guidance.

Byte code sequences. We extract the API sequences directly from Android byte code. For each method implementation, we extract the API methods and constants used in it into one sequence. There is no ordering between sequences collected from different method implementations. Based on our observation, the order of the API methods and constants in these sequences is close to their order in source code. We cover the byte code option because it reflects the effect of embedding without program analysis guidance.

Program slices. We apply a program analysis strategy, *interprocedural backward slicing*, to obtain program slices. The slicing starts from the variables used with a cryptographic API invocation. By backwardly tracing the data flows reaching these variables, all the code statements influencing the API invocation are kept while irrelevant code statements are excluded. When reaching the entry point of the current method, we jump to its callers to continue the backward tracing until the tracked data facts are empty or there is no caller found. In this way, the influencing code context beyond a local method is also collected. When meeting a self-defined method call, we replace it with its implementation code if available. An example of program slices is shown in Fig. 4.1b. A major difference between

program slices and byte code is that the irrelevant predecessors are removed by program analysis.

API dependence paths. With program analysis, the code semantic information, such as program dependencies, can be extracted. We perform the *API dependence graph construction* and extract the API dependence paths for embedding. The API dependence graphs are built through dataflow analysis. We add the data dependence edges between API calls on slices. An example of our API dependence graph is shown in Fig. 4.1c. It uses an API or a constant as a node. Two nodes having the data dependence (`def-use`) relationship are connected directly. The API dependence paths are covered in our measurement as a representative of the state-of-the-art code semantic based approaches. [31, 78].

Experimental setup of program analysis preprocessing. We implement an interprocedural, context- and field-sensitive dataflow analysis to achieve our backward slicing and API dependence graph construction. The analysis is implemented with the Java program analysis framework Soot [172]. Soot takes the Android byte code as input and transforms it into an intermediate representation (IR) Jimple. The program analysis (i.e., slicing or API dependence graph construction) is performed on Jimple IR. We use Soot 2.5.0, Java 8, and Android SDK 26.1.1.

4.3.2 Token-level Embedding Settings

We perform the token-level embedding training to produce vectors for tokens in an embedding vocabulary.

Embedding vocabulary. We collect the embedding vocabulary when performing program analysis. By scanning the program, we perform backward slicing for encountered cryptographic API calls. In this way, we extract cryptographic code from the apps. All the API

methods and constants appearing in the cryptographic code are collected as a vocabulary. For API methods, we filter those that appear less than five times. For constants, we manually collect 104 reserved string constants used as the arguments of cryptographic APIs. We also collect the constants that appear more than 100 times in the slices. Finally, we have a vocabulary of 4,543 tokens (3,739 APIs and 804 constants). The API methods include the standard APIs from Java and Android platforms, as well as some third-party APIs that cannot be inlined because of recursion or phantom methods (whose bodies are inaccessible during the analysis). Table 4.2 shows the library distribution of these API methods.

Table 4.2: The library sources of our embedding APIs

Source		**Number of embedded APIs**
Java platform	java.security	510
	javax.crypto	166
	java.io	138
	java.lang	259
	others	374
Android platform		486
Third parties		1827

We train the skip-gram embedding model [118] to obtain the word2vec-like embedding. With different program analysis preprocessing, three types of token-level embeddings, *byte2vec*, *slice2vec*, and *dep2vec* are produced.

- *byte2vec* is the baseline embedding version that applies *word2vec* [117, 118] directly on the byte code corpus.

- *slice2vec* is the embedding with the inter-procedural backward slicing as the preprocessing method.

- *dep2vec* applies API dependence graph construction to guide the embedding training.

Experimental setup for token-level embeddings. We follow the convention of the natural language embedding word2vec to set hyperparameters. The embedding vector length is 300. The sliding window size for neighbors is 5. We also applied subsampling and negative sampling to randomly select 100 false labels to update in each batch. Based on our preliminary experiments, we train embeddings with a mini-batch size of 1024. The embedding terminates after 10 epochs. Because we did not oberserve significant improvement by longer epochs and smaller batch size.Our embedding model is implemented using Tensorflow 1.15. Training runs on the Microsoft AzureML GPU clusters, which support distributed training with multiple workers. We use a cluster with 8 worker nodes. The VM size for each node is the (default) standard NC6.

4.3.3 Sequence-level Embedding Settings

We obtain sequence-level embeddings by applying the method of training the well-known natural language embedding BERT [63] on program sequences.

byteBERT vs. sliceBERT vs. depBERT. On byte code, program slices, and API dependence paths, we obtained three different versions of BERT embeddings for API elements, *byteBERT*, *sliceBERT*, and *depBERT*. These BERT-like API embeddings are produced by pretrained Transformer neural networks. We apply the masked language modeling (MLM) task to pretrain them. MLM is a task that reconstructs language sequences with masked tokens. It predicts the missing tokens for a given sequence with random masks. The masked tokens in the input sequence are either replaced by a special token `[MASK]` or an arbitrary random token in the vocabulary or kept in original sequences. We set the probabilities of the three situations as 80%, 10%, and 10%, and to follow the convention in NLP. The masked tokens are randomly selected with the probability of 30% and one sequence is lim-

Table 4.3: Overview of our datasets

Dataset	App Set ID	Number of Apps	Experiments
Basic	1	16,048	Embedding API Completion
Advanced	2	64,478	Embedding
	3	11,997	API Completion
	4	1,819	API Completion
	5	1,055	API Completion
	6	538	API Completion

ited to having two masked tokens at most. These are similar to the setting of the MLM for training BERT [63]. We discard the next sentence prediction (NSP) of BERT as there is no corresponding concept of "next sentence" between two code sequences. Three types of sequence-level embeddings are trained with identical hyperparameters. Same with the LSTM training with token-level embedding, the neural network is trained for 10 epochs with batch size 1024. When training the Transformer model, the input tokens are represented as our token-level embedding. To apply these sequence-level embedding in cryptographic API completion, the pretrained neural networks are finetuned them by the given task-specific data.

4.3.4 Dataset Overview

We conduct experiments on Android Apps we collected from the Google Play store. According to the way we split data for training and testing, we have a basic dataset and an advanced cross-app data setup. Table 4.3 gives an overview.

Basic Data Split Setting

The basic dataset is composed of 16,048 Android Apps from three categories, 5,176 Apps from the business category, 4,581 Apps from the communication category, and 6,291 Apps from the finance category. From these Apps, we extracted 707,775 API sequences from byte code, 926,781 API sequences from program slices, and 566,279 API sequences from API dependence graphs. The number of tokens in the three types of sequences is shown in Table 4.4. The tokens refer to the APIs or constants in our embedding vocabulary.

Table 4.4: Embedding corpora statistics of the basic dataset

Corpora	Byte code	Slices	Dependence paths
Number of tokens	28,887,852	12,341,912	38,817,046

For embedding, we use all of the API sequences to produce the token-level embeddings and sequence-level embeddings. For API completion tasks, we randomly split all the sequences for training and testing following the ratio 4:1.

Advanced Data Split Setting

We create an advanced dataset to enable cross-app learning and validate our findings on new apps. Under this setup, the collected apps are first split for embedding and API completion tasks, respectively. This guarantees that the apps used for API completion tasks are not seen in the embedding training phase. Our embedding experiments are conducted on 64,478 Apps (App set 2), which are much more than the Apps (App sets 3, 4, 5, 6) we used for API completion tasks. This consideration is because embeddings are often pretrained with huge data volumes and released for fine-tuning with smaller task-specific datasets in the real world. Then, the apps for API completion tasks are split into training and testing sets. This

guarantees that the apps used for testing are never seen in the training. Compared with the basic dataset, the cross-app setting is more practical and challenging. It evaluates whether the model trained on a set of apps can be applied to new apps.

In addition, to observe the impacts of the task data volume, we perform API completion training and testing on four App sets (App sets 3, 4, 5, 6) varying in data sizes. The largest one, App set 3, is a diverse App set including 11,997 Apps from 12 App categories. Besides, there are three smaller App sets (App sets 4, 5, 6) consisting of 1,819 Apps from the personalization category, 1,055 Apps from the social category, and 538 Apps from the weather categories.

4.4 Evaluation Results

We measure the accuracy of the cryptographic API completion to compare the impacts of different embedding choices. We calculate the top-1 accuracy that only considers the correctness of the top-1 prediction of the model. When deciding whether a top-1 prediction is correct, we use the ground truth from the sequence itself. For API dependence paths from a graph, there might be multiple correct answers due to the branches of the graph.

4.4.1 Comparative Experiments for RQ1

Our RQ1 is answered by comparing token-level embedding and one-hot encoding on byte code, slices, and dependence paths, respectively. The results are shown in Tables 4.5 and 4.6.

Experimental setup for cryptographic API completion tasks. We train LSTM based models for the task. For *next API completion* task, we train the LSTM based sequence model to accept a sequence of API methods or constants $(t_1, t_2, \ldots, t_{n-1})$ and output the next API t_n.

For *next API sequence completion* task, we train the LSTM based seq2seq (encoder-decoder) model to accept the first half API sequence $(t_1, t_2, \ldots, t_n)$ and predict the last half of the sequence $(t_{n+1}, t_{n+2}, \ldots, t_{2n})$.

We filter the vulnerable code using CryptoGuard [147], which is a static cryptography API misuse detection tool. Only secure code is kept to avoid data poisoning [157]. We limit the maximum number of LSTM steps to 10 and use a batch size of 1,024, a learning rate of 0.001. The highest accuracy achieved within 10 epochs is recorded. These hyperparameters are selected because no obvious accuracy improvement is observed by longer epochs, smaller batch size or learning rate. We use the stacked LSTM architecture with vanilla LSTM cells for the LSTM-based models.

Byte code vs. program slices vs. API dependence paths

Table 4.5: Accuracy of the next API completion on the basic dataset.

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 4.6: Accuracy of the next API sequence completion on the basic dataset. We use the LSTM-based sequence model with hidden layer size 256 for this task.

Byte Code		**Slices**		**Dependence Paths**	
1-hot	byte2vec	1-hot	slice2vec	1-hot	dep2vec
43.61%	44.63%	64.10%	85.02%	82.94%	89.23%

Tables 4.5 and 4.6 show the accuracy results of the *next API completion* and the *next API sequence completion*, respectively. To uncover the impact of the program analysis preprocessing, Both the token-level embedding (i.e., *byte2vec*, *slice2vec*, *dep2vec*) and the

one-hot encoding baseline are used to train the LSTM models on byte code, slices, and dependence paths.

We observe that program analysis preprocessing shows significant benefits. Table 4.5 shows the accuracy on dependence paths is 92%, which is 9% higher than on slice, and 36% higher than on byte code, with corresponding token-level embeddings. The API completion accuracy with one-hot encoding is also substantially improved by program analysis. The accuracy with one-hot encoding increases from 56% on byte code to 72% on slices, and further to 86% on dependence paths. The results of the next API sequence completion (Table 4.6) are also consistent with the conclusion. It shows that the accuracy achieved with *byte2vec* improved by 40.39% with *slice2vec*, and improved by 44.60% with *dep2vec*.

> Finding 1: For Crypto API completion with token-level embeddings, program analysis significantly improves the accuracy by 36.20%[2] on average.

Table 4.7: Accuracy of the next API completion with or without sequence-level embedding (pretrain) on the basic dataset.

Model Size	Byte Code		Slices		Dependence Paths	
	Transformer + byte2vec (w/o. pretrain)	byteBert (w. pretrain)	Transformer + slice2vec (w/o. pretrain)	sliceBert (w. pretrain)	Transformer + dep2vec (w/o. pretrain)	depBert (w. pretrain)
Small	44.38%	45.21%	83.37%	84.15%	90.96%	91.07%
Base	56.76%	57.59%	84.80%	84.83%	92.80%	93.52%

Token-level embedding vs. one-hot vectors

On each program analysis preprocessing representation, we compare the token-level embedding and the one-hot encoding baseline. We observe significant improvements by applying token-level embeddings on slices and dependence paths. However, the improvement on byte code is limited. Table 4.5 shows that *slice2vec* improves the accuracy by 11% from its one-hot

[2]dep2vec column - byte2vec column in Table 4.5

baseline. *dep2vec* improves the accuracy by 6% from its one-hot baseline. These improvements suggest that *slice2vec* and *dep2vec* capture useful information. This conclusion is also observed in the next API sequence recommendation task. *slice2vec* and *dep2vec* improve the accuracy from their baselines by around 21% and 6%, respectively. In contrast, *byte2vec* does not show any significant improvement from its one-hot baseline.

> Finding 2: For Crypto API completion on program slices and API dependence paths, token-level embedding achieves average accuracy improvement by 12.02% and 3.97%, respectively, compared with one-hot vectors.

We also observe higher accuracy achieved by longer LSTM units, which is as expected. Furthermore, the accuracy increases more rapidly from LSTM-64 to LSTM-128 compared with LSTM-256 to LSTM-512. It is likely because once the model has enough capacity, the differences caused by embedding are smaller.

Overall, the best accuracy is achieved by *dep2vec* in both tasks, an accuracy of 92.04% in the *next API completion* task and an accuracy of 89.23% in the *next API sequence completion* task. Compared with the basic one-hot encoding on byte code (no program analysis preprocessing), they achieve substantial accuracy improvements (36% and 46%, respectively) in both tasks.

4.4.2 Comparative Experiments for RQ2

Our RQ2 is answered by comparing sequence-level embedding with token-level embedding on byte code, slices, and dependence paths, respectively. When a sequence-level embedding applied, we fine-tune it (i.e., *byteBERT*, *sliceBERT*, or *depBERT*) with the task-specific training. They are compared with unpretrained Transformer networks with token-level em-

beddings. We include two Transformer neural networks with different size, Transformer-base and Transformer-small. The Transformer-base model has 12 hidden layers with the size 768 and 12 attention heads. The Transformer-small model has 4 hidden layers with the size 512 and 4 attention heads. Results are shown in Tables 4.7.

Byte code vs. program slices. vs. API dependence paths

Table 4.7 shows that program analysis preprocessing is still necessary even with sequence-level embeddings. The accuracy results on byte code sequences are low (45.21% and 57.59%) compared with program slices and API dependence paths. With the program analysis, the small and base Transformer neural networks with *depBERT* achieve the accuracy of 91.07% and 93.53%, respectively. When there is only token-level embedding, this conclusion still holds. The small and base Transformer neural networks with *dep2vec* achieve the accuracy of 90.96% and 92.80%, respectively, which are 46.58% and 36.04% higher than the *byte2vec* on byte code sequences.

> Finding 3: For Crypto API completion with sequence-level embedding, program analysis makes substantial accuracy improvement of 40.90%[3] on average.

We observe the impact of program analysis is much more significant than the model size. Although a larger neural network usually results in higher accuracy, we observe that a smaller Transformer with program analysis is better than a larger one without program analysis. In Table 4.7, the accuracy achieved by the small *depBERT* is 33.48% higher than the accuracy of the base *byteBERT*.

By comparing the Transformer with token-level embeddings in Table 4.7 and the LSTM with

[3]depBERT column - byteBERT column in Table 4.7

[4] Compare the Transformer columns in Table 4.7 with the byte2vec, slice2vec, and dep2vec columns in Table 4.5

token-level embeddings in Table 4.5, we found that the LSTM-512 achieves slightly higher accuracy than the Transformer-small with a comparable size (hidden size 512).

> Finding 4: For Crypto API completion, LSTM-512 shows 4.22% [4] accuracy advantage on average over Transformer-small (hidden size 512).

Sequence-level embedding vs. one-hot embedding

Sequence-level embeddings only show slight advantages over token-level embeddings. As shown in Table 4.7, the accuracy trained with the sequence-level embeddings is only slightly higher (0.55% on average) than the Transformer neural network with their token-level baselines. Considering the cost, sequence-level embedding is not recommended in this case.

Besides, the impact of the neural network size is also more obvious than the impact of applying sequence-level embedding. As shown in Table 4.7, the base Transformer improves the accuracy by 12.38%, 1.43% and 1.84%, on byte code, slices, and dependence paths, respectively, compared with the small Transformer.

> Finding 5: Although resulting in slight accuracy improvement (0.55% [5] on average), sequence-level embedding is not the first recommended strategy to improve the Crypto API completion, compared with program analysis and a larger model.

4.4.3 Comparative Experiments for RQ3

Cross-app learning is a practical scenario in which we expect a pretrained model can be applied to other projects unseen in the training phase. Therefore, we conduct experiments

[5] Compare between BERT columns and Transformer columns in Table 4.7

to observe whether our conclusions still hold for new apps.

Table 4.8: Accuracy of the next API completion with or without sequence-level embedding (pretrain) on dependence paths of the advanced dataset (cross-app learning).

App Set	**# of cases**	**Transformer** + dep2vec (w/o. pretrain)	**depBert** (w. pretrain)	**Improvement**
3	813,737	97.23%	98.24%	1.01%
4	97,224	98.66%	99.54%	0.88%
5	88,143	95.09%	96.78%	1.69%
6	26,357	83.34%	88.44%	5.10%
Avg.	**256,363**	**93.58%**	**95.75%**	**2.17%**

Table 4.9: Accuracy of the next API completion with or without sequence-level embedding (pretrain) on byte code sequences of the advanced dataset (cross-app learning).

App Set	**# of cases**	**Transformer** + byte2vec (w/o. pretrain)	**byteBert** (w. pretrain)	**Improvement**
3	7,275,324	79.72%	80.00%	0.28%
4	814,551	86.91%	87.21%	0.30%
5	840,381	77.46%	77.96%	0.50%
6	220,543	65.05%	67.41%	2.36%
Avg.	**2,287,700**	**77.29%**	**78.15%**	**0.86%**

Tables 4.8 and 4.9 show the API completion experiments on our advanced dataset (see Section 4.3.4) which follows the cross-app learning scenario. App sets 3, 4, 5, 6 include Apps that task-specific data is generated from. For every App category, we randomly select 80% Apps of this category to generate training data and 20% Apps to generate testing data. In another word, our training and testing data is cross-app but within a category.

Tables 4.8 and 4.9 compare sequence-level embeddings (i.e., *depBERT* and *byteBERT*) with the corresponding token-level embeddings (i.e., *dep2vec* and *byte2vec*). *DepBERT* and *byteBERT* are Transformer neural networks pretrained on App set 2 (see Table 4.3) with MLM. We use the small Transformer neural network for all the experiments.

We observe similar conclusions with the basic dataset about program analysis. The experiments on API dependence paths (Table 4.8) still show significant advantage compared with on byte code sequences (Table 4.9).

A minor difference we observe is that sequence-level embedding brings more obvious improvement than on the basic dataset. As shown in Table 4.8, the average improvement of applying the sequence-level embedding is 2.17%. This indicates that sequence-level embedding is more significant when we train our models in the cross-app scenario. We observe that the sequence-level embedding substantially improves the accuracy for the insufficient data. It achieves an accuracy 5.10% higher than the Transformer with *dep2vec*.

From Table 4.9, we also observe the improvement of applying sequence-level embedding *byteBERT* on byte code sequences. However, without program analysis, the improvements (0.86% on average) are quite small.

> Finding 6: In cross-app setting, sequence-level embedding achieves more obvious accuracy improvements (2.17% on average) compared with basic data split setting. We recommend using sequence-level embedding in cross-app learning when task data is insufficient.

4.4.4 Comparative Experiments for RQ4

Besides the design choices we covered, we further experiment on two state-of-the-art sequence-level embeddings, GraphCodeBert [78] and CodeBert [68]. GraphCodeBert and CodeBert are general purpose code embedding models pretrained by Microsoft. They adopt the Transformer-based neural architecture and pretrain it on CodeSearchNet dataset [89] which includes 2.3M functions of six programming languages paired with natural language descrip-

tion. The differences between them are their code preprocessing parts and sequence-level embedding tasks. CodeBert treats code as a sequence of tokens and is pretrained by masked language modeling (MLM). GraphCodeBert uses program analysis to extract dataflow information as input and is pretrained by two extra structure-aware tasks introduced by the authors.

Table 4.10 shows the next API completion experiments on our App sets 4, 5 and 6. We decompiled apk files into source code for the neural network inputs. For each cryptographic API call, we extract two types of source code context for it, the method-level context and the class-level context. The former extracts the previous code within the wrapper method where the target call locates while the latter collects the previous code lines found in the same class of the target call. We finetune the two models with our data for 10 epochs with batch size 16. We use this setting because no substantial improvement is observed by longer epochs or smaller batch size.

Finding 7: The state-of-the-art general purpose pretrained models only achieve a low accuracy (59.64% by GraphCodeBert on average) for cryptographic API completion. The program analysis preprocessing and the method-level context are recommended.

Table 4.10: Accuracy of the next API completion by finetuning the general purpose pretrained model GraphCodeBert and CodeBert.

App Set	GraphCodeBert		CodeBert	
	Method-level Context	Class-level Context	Method-level Context	Class-level Context
4	60.45%	39.87%	41.72%	30.82%
5	64.53%	37.83%	41.29%	31.80%
6	54.84%	35.25%	36.60%	31.32%
Avg.	**59.94%**	**37.65%**	**39.87%**	**31.31%**

We have three observations from Table 4.10. First, the best accuracy is achieved by Graph-

CodeBert with the method-level context. However, the accuracy is still at a low level, on average 59.94%. Second, GraphCodeBert substantially outperforms CodeBert in identical data and context settings. When using method-level context, GraphCodeBert has an accuracy 20.07% higher accuracy than CodeBert on average. When using class-level context, GraphCodeBert achieves 6.34% higher accuracy on average. This confirms our findings 1 and 3 that program analysis contributes a substantial improvement to the embeddings. Another observation is that method-level context is much better than class-level context. With GraphCodeBert, the method-level context outperforms the class-level context by 22.29% accuracy improvement on average. With CodeBert, the method-level context results in a 8.80% higher accuracy on average. The reason might be that the class-level context includes much more irrelevant information and makes the prediction worse.

We summarize our major findings from experiments.

- Program analysis preprocessing is very important even with advanced embedding options. With all the embedding options (sequence-level embedding, token-level embedding, or one-hot encoding), program analysis makes big improvements on the API completion accuracy. Without program analysis, the best accuracy on byte code with the most advanced *byteBERT* is only 57.59%. With the API dependence graph construction, *depBERT* on dependence paths achieves the highest accuracy of 93.52% on the basic dataset.

- Applying token-level embedding in API completion task training makes substantial improvement on program analysis process code corpora. On slices and dependence paths, the LSTM models trained with the token-level embedding *slice2vec* and *dep2vec* show significant accuracy improvements by 12% and 5%, respectively, compared with the one-hot vectors.

- The accuracy improvement of sequence-level embedding (0.55% on average) is not obvious under the basic setting. Hence, we do not recommend sequence-level embedding in that case. Meanwhile, we observer more significant improvements (5.10%) of sequence-level embedding under the cross-app scenario when the task-specific data size is small (App set 6). Therefore, we recommend it when training model for cross-app scenario and lacking enough task-specific data.

4.4.5 Analogy Tests of Token-level Embedding

We perform the analogy tests to intuitively show the quality of token-level embeddings. In natural languages, the quality of embedding is usually evaluate through analogous pairs (e.g., $men - women \approx king - queen$) [117, 118, 119]. However, there are no obvious analogous relations between the Java cryptographic APIs and constants. In our work, we define analogous pairs as two pairs of APIs or constants, (a and a') with (b and b'), having a high degree of relational similarity (i.e., analogous) in terms of some programming property. For Java cryptographic code, we identify four categories of analogous pairs as follows. We show examples in Table 4.11.

Direct Dependency. For two APIs where one always accepts the other's output, they form a pair having the direct dependency. For example, after a `KeyGenerator` instance is created by `KeyGenerator.getInstance(.)`, it always needs to be initialized through `KeyGenerator.init(.)`. The analogous relation could also be found between `KeyStore.getInstance(.)` and `KeyStore.load(.)` where the latter loads the required information to the `KeyStore` instance created by the former. We view the two pairs as analogous pairs under this category.

Semantic Symmetry. For two classes `KeyGenerator` and `KeyPairGenerator`, the former generates secret keys for symmetric cryptography while the latter generates keys for asym-

Table 4.11: Four categories of analogous pairs we define among API methods and constants. We give a representative example for each category, where two pairs (a and a' vs. b and b') have a high degree of relational similarity (i.e., analogous) in terms of some programming property. For each category, the number of analogies used in our top k evaluation (Table 4.12) is also shown.

Category		Examples of Analogous Pairs	Number of Analogies
Direct Dependency	a_1	`javax.crypto.KeyGenerator: javax.crypto.KeyGenerator getInstance(java.lang.String)`	4
	a'_1	`javax.crypto.KeyGenerator: void <init>(int)`	
	b_1	`java.security.KeyStore: java.security.KeyStore getInstance(java.lang.String)`	
	b'_1	`java.security.KeyStore: void load(java.io.InputStream,char[])`	
Semantic Symmetry	a_2	`javax.crypto.KeyGenerator: javax.crypto.KeyGenerator getInstance(java.lang.String)`	4
	a'_2	`javax.crypto.KeyGenerator: javax.crypto.SecretKey generateKey()`	
	b_2	`java.security.KeyPairGenerator: java.security.KeyPairGenerator getInstance(java.lang.String)`	
	b'_2	`java.security.KeyPairGenerator: java.security.KeyPair generateKeyPair()`	
Argument Symmetry	a_3	`"AES"`	4
	a'_3	`javax.crypto.KeyGenerator: javax.crypto.KeyGenerator getInstance(java.lang.String)`	
	b_3	`"RSA"`	
	b'_3	`java.security.KeyPairGenerator: java.security.KeyPairGenerator getInstance(java.lang.String)`	
Syntactic Variants	a_4	`javax.crypto.Cipher: byte[] doFinal(byte[])`	2
	a'_4	`javax.crypto.Cipher: int doFinal(byte[],int)`	
	b_4	`javax.crypto.Mac: byte[] doFinal(byte[])`	
	b'_4	`javax.crypto.Mac: void doFinal(byte[],int)`	

Table 4.12: Top k accuracy of 14 analogous pairs in different embedding vectors. Smaller k suggests more accurate embedding vectors that better maintain analogous relationships.

Category	Top k Accuracy		
	dep2vec	slice2vec	byte2vec
Direct Dependency	**2**	**2**	50
	2	4	14
	3	13	41
	1	2	42
Semantic Symmetry	**2**	65	**2**
	20	**3**	8
	9	204	385
	4	239	355
Argument Symmetry	**1**	94	5
	1	49	16
Syntactic Variants	15	84	**2**
	9	95	249
	2	326	191
	9	278	419

metric cryptography. There is a symmetry relationship between their APIs. For example, they both have the APIs `getInstance(String)` to create instances and APIs to generate the key.

Argument Symmetry. There is an analogous relation between API - constant pairs. For example, symmetric cipher `"AES"` can be passed to `javax.crypto.KeyGenerator: javax.crypto.KeyGenerator getInstance(java.lang.String)` as an argument. For asymmetric ciphers, `"RSA"` and API `java.security.KeyPairGenerator: java.security.KeyPairGenerator getInstance(java.lang.String)` have a similar relation.

Syntactic Variants. Some APIs share the same name but differ in their full signatures. These APIs are functionally equivalent but have different types of arguments or return values. We name them *syntactic variants*. For example, there are several APIs with the same name `doFinal(.)` of the Java class `Cipher` and Java class `MAC`.

In total, we define 14 analogy tests. Their top-k accuracy is listed in Table 4.12. We calculate the vector of the embedded object b' based on the other three vectors of a, a', and b. If the actual embedding vector of b' appears in the top k nearest list of the calculated one, we say this analogy achieves the top k accuracy.

In this small-scale analogous pairs evaluation, *dep2vec* performs the best. *dep2vec* achieves the best top-k accuracy 12 times of the 14 test cases. *slice2vec* does well in some cases. but performs poorly in syntactic variants category. This is likely because the syntactic variant APIs usually appear in different contexts in slices, making *slice2vec* fail to recognize their similarity. For other more complicated relationships like semantic symmetry or argument symmetry, the APIs and constants belonging to a pair often appear far away from each other in the code, increasing the difficulty of the test.

4.5 Case Studies and Discussion

To help interpret how program analysis and embedding vectors help the API completion, we show several case studies.

Case Study 1. This case study is on the effectiveness of the API dependence graph construction. Figure 4.2(a) shows a slice-based test case that is mispredicted by both *slice2vec* and its one-hot baseline. For digest calculation, it is common for `MessageDigest.update(.)` to be followed by `MessageDigest.digest(.)`, appearing 6,697 times in training. However, Figure 4.2(a) shows a reverse order, which is caused by the `if-else` branch shown in Figure 4.2(b). When `MessageDigest.update(.)` appears in an `if` branch, there is no guarantee which branch would appear first in slices. This reverse order is less frequent, appearing 1,720 times in training. Thanks to the API dependence graph construction, this confusion is eliminated, which predicts this case correctly.

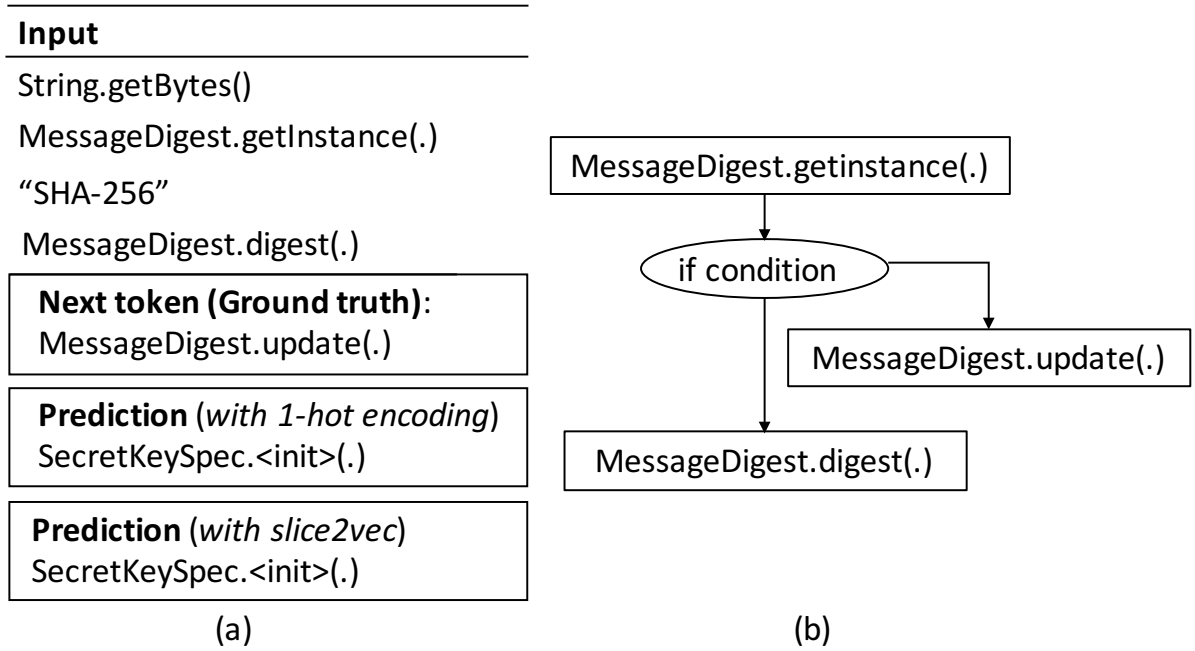


Figure 4.2: Case Study 1. (a) A slice sequence that is predicted incorrectly. (b) Model trained on API dependence graphs makes the correct prediction.

Case Study 2. This case study is on the ability to recognize new previously unseen test cases. The slices in Figure 4.3(a) and Figure 4.3(b) slightly differ in the arguments of the first API. *slice2vec* makes the correct predictions in both cases, while its one-hot baseline fails in Figure 4.3(a). `MessageDigest.getInstance(String)` appears much more frequent than `MessageDigest.getInstance(String,Provider)` in our dataset. Specifically, the former API appears 207,321 times, out of which 61,047 times are followed by the expected next token `MessageDigest.digest(.)`. In contrast, the latter API – where one-hot fails – only appears 178 times, none of which is followed by `MessageDigest.digest()`. In *slice2vec*, the cosine similarity between `MessageDigest.getInstance(String,Provider)` and `MessageDigest .getInstance(String)` is 0.68. [6] This similarity, as the result of *slice2vec* embedding, substantially improves the model's ability to make inferences and recognize similar-yet-unseen cases.

[6]For one-hot vectors, this similarity is 0.

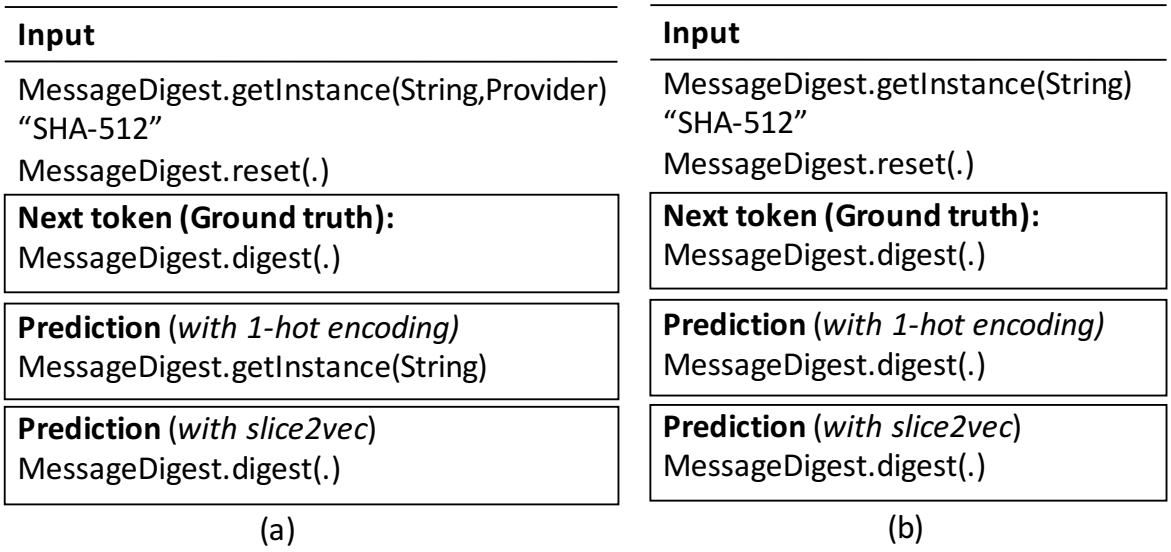


Figure 4.3: Case Study 2. (a) A test case that is correctly predicted with *slice2vec*, but incorrectly with one-hot vector. (b) A test case similar to (a), but both *slice2vec* and one-hot give the correct prediction.

Soundness. Our conclusions are sound because they are made from strict controlled experiments. For the three dimensions, program analysis, token-level embedding and sequence-level embedding, we measure the impact of a specific design by comparing the API completion trained with or without it.

Threats to Validity. An internal threat to validity is that we use identical training hyperparameters for all the API completion experiments. When applying different program analysis and embedding techniques, we train neural networks with identical training hyperparameters. We did not tune hyperparameters to find the best practices for every case. An external threat to validity comes from the dataset we use in the measurement. We only perform API completion experiments with Java cryptographic API benchmark, although the embedding method is for general purpose. We choose Java cryptographic APIs because it is complicated. Our future work will extend the benchmark with more diverse APIs.

Limitations. We briefly discuss our limitations. First, we perform a security sanitization

to filter insecure code in our dataset. However, the security sanitization relies on a static analyzer that may not be perfect. Second, we apply static analysis to extract program slices and dependence paths. However, static analysis tends to overestimate execution paths. Thus, the slices and dependence paths used for learning might not necessarily occur. Third, we do not try other embedding techniques such as ELMo [141]. We met an incompatibility issue when adapting the published ELMo code for our API completion task. The published code requires outdated libraries such as TensorFlow v1.2, and CUDA 8 while our learning environment only supports CUDA 9 or later. We will consider adding more embedding models in the future.

4.6 Related Work

There are two main branches of code embedding solutions.

Embedding without program analysis. First, a line of research develops pure data-driven solutions on general source code tokens without program analysis [19, 46, 68, 93, 94, 163]. They train neural network solutions to take as input programs that are treated as sequences of source code tokens. In [46], Buratti *et al.* claimed that the language model built on top of raw source code is able to discover abstract syntax tree (AST) features automatically.

Embedding with program analysis. Second, some studies (e.g., [23, 42, 85, 195]) leverage the program structural information through program analysis. For example, the authors of [195] learned code embedding after constructing the graph representations (e.g., control flow graphs, data flow graphs) of code. Hellendoor *et al.* [85] advocated a hybrid embedding method that considers both the graph structure and the raw sequences to overcome the size limit of graphs. To remove noises in code, Henkel *et al.* performed intra-procedural symbolic execution first and trained embedding vectors of symbolic abstractions from sym-

bolic traces [86]. However, there have not been systematic studies on how various hybrid approaches compare with a pure data-driven approach or with each other, in terms of downstream task performance.

Since there are various program intermediate representations (IRs) under program analysis, the embedding objects also vary from approach to approach. For example, Henkel *et al.* obtained embeddings for self-defined symbolic abstractions. Ding *et al.* [64] obtained embedding vectors *asm2vec* for assembly code instructions. Ben-Nun *et al.* [31] embedded LLVM IR instructions of code. Although the idea of leveraging the program structural information in embeddings is identical, these embeddings for low-level instructions, or LLVM IRs cannot be directly compared with embeddings for API elements. Our *dep2vec* and *depBERT* can be viewed as the graph based embedding approaches applying for API elements.

A line of work focuses on API embeddings and related tasks [30, 45, 50, 65, 77, 131, 132, 183]. Our work also lies in this category. Nguyen *et al.* [131, 132] use API sequences in source code to produce embeddings for Java APIs and C# APIs. Using these vectors, they successfully mapped the semantic similar Java APIs with C# APIs. Our *byte2vec* can be viewed similarly to it as our API call sequences from byte code are similar to their source code order. Chen *et al.* [50] trained the API embedding based on the API description (name and documents) and usage semantics. The obtained API embeddings are used to infer the likely analogical APIs between third party libraries. However, these solutions employ embeddings to help map analogical APIs, which is different from our task, API completion. In API completion work [128, 130, 133, 150, 162], there is either no discussion about the impacts derived from different embedding options.

4.7 Conclusions and Future Work

In this work, we systematically compared several general-purpose code embedding strategies that are useful for securing Java cryptographic implementations, which are known to be challenging to develop correctly. We performed extensive experimental evaluations, including downstream tasks on API recommendation and analogous pairs analysis. We also conducted multiple in-depth case studies to identify the reasons behind the misclassified neural network results, which are notoriously difficult to explain.

Our case studies on false positive and false negative cases pointed out several promising future directions, including adding weights to tokens with advanced neural network architectures and mechanisms to further highlight security semantics (i.e., secure vs. insecure) in embedding training beyond programming semantics (e.g., API sequences).

Chapter 5

Neural Network based Code Suggestion

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%. Multi-HyLSTM shows a significant boost from the state-of-the-art API completion tools SLANG (by 18%) and Codota (by 51%). Moreover, we show the effectiveness of our design choices through an ablation study. We have also released our dataset.

5.1 Introduction on API Recommendation

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 [43, 60, 63, 108, 187], neural network based code completion has received much attention [68, 111, 163]. Early efforts [87, 88, 149] treated programs as a sequence of source code tokens and built statistical language models on the sequential context for the next token generation. While being promising, these approaches are unable to guarantee syntax and semantic correctness and hence fail to generate high-quality code [23].

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 [104] and Github Copilot [51], 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. For example, developers often have difficulties in choosing the correct cryptographic APIs from a couple of similar ones. To improve the top-1 accuracy of the neural networks, program specific challenges need to be carefully addressed. Towards this direction, many studies [35, 42, 53, 123, 163] proposed solutions that incorporate program syntax or semantic properties. Some studies focus on representing programs as structural representations, such as syntax trees [53, 123, 163] or graphs that show the program's control flow or data flow [21, 78, 195]. Moreover, some studies applied formal language grammars (e.g., context free grammar, attribute grammar) working with neural networks to guide the code generation process [42, 121].

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 [6] only achieves 64.9% accuracy in recommending the next API method. Figure 5.1 (a) and (b) have slightly different code contexts and require two distinct next 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 program context and make correct suggestions accordingly. Another state-of-the-art neural network based API completion solution SLANG [150] also only obtains an accuracy of around 77.4%.

Our analysis finds that the root cause of the low prediction accuracy is the lack of ability to learn program dependencies. Thus, in this work, we design a neural network solution that understands program dependencies, with the help of specialized static analysis. We identify two previously unreported programming language specific challenges. The first challenge is how to recognize the 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 work. We found that the impact of global dependencies is often neglected by existing solutions, especially when they are less frequent. 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 approach of representing program context sequentially has difficulty distinguishing these APIs and is problematic. We design a new multi-path neural network

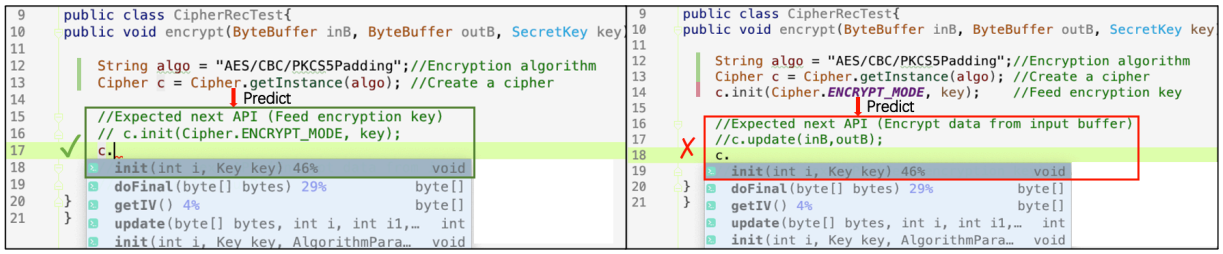


Figure 5.1: API completion and accuracy. (a) Codota's recommendation for Line 17 is correct. (b) Codota's recommendation for Line 18 is wrong. The code context for (a) and (b) differs at Line 14 but Codota give identical predictions for them.

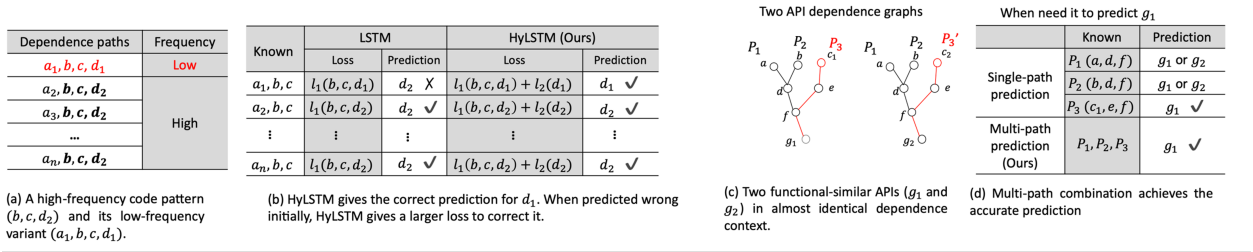
architecture to aggregate the impacts of multiple dependence paths to make predictions. Our ultimate model is named Multi-HYLSTM.

We extensively evaluate our Multi-HyLSTM design with Java cryptographic code extracted from Android apps. We choose cryptographic APIs because they are known to be error-prone [13, 115, 147]. An accurate API completion solution is important and can bring huge benefit. Our evaluation as well as case studies demonstrate that our approach Multi-HyLSTM is effective making more accurate API suggestions, substantially in 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 the state-of-the-art models (e.g., LSTM, BERT) in learning program dependencies. We performed in-depth manual analysis on 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 for learning the global dependencies and multi-path dependencies. Multi-HyLSTM includes two major features, a multi-path architecture and a global depen-



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

(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	$l_1(b, c, d_2)$	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.

(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

Figure 5.2: Examples illustrating dependence challenges in code completion and how we address them.

dence enhancing learning module named HyLTM. This neural network works together with our program context representation part, – 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 identified 774,460 Cryptographic API callsites from 64,478 Android Apps. Our evaluation include two parts. First, we compared Multi-HyLSTM with the state-of-the-art neural network based API completion tools SLANG [150] and Codota [6]. Our approach outperforms them in the top-1 completion accuracy by large margins (18% and 51%). Second, 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 have also published a large-scale Java cryptographic code dataset [1] that can be used as a benchmark to evaluate API completion model accuracy.

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

5.2 Program dependence specific challenges

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

Definition 5.1 (Global dependence)**.** We use global dependence of a program point `p` to refer to the code instructions that `p` depends on but locate outside of the wrapping method of `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 5.2 (a). The subsequence (a_1, b, c, d_1) is a more 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 cannot deal with it (Section 5.4.2).

To address this issue, we present a new sequential model HyLSTM by modifying the LSTM loss function (Section 5.3.3). 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 5.4.2).

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 5.2 (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 sequential model that takes a single path or a mixed context sequence as inputs, our architecture models each single path separately and aggregates their impacts.

5.3 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 suggest the next API method call x_n based on the given context C.

5.3.1 Overview of Our Approach

Our approach takes the advantages 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 5.1.

$$R(C) = P_1, P_2, \ldots, P_k \tag{5.1}$$

where $R(.)$ represents our program analysis preprocessing in Section 5.3.2. P_i denotes a extracted dependence path.

Next, we calculate the conditional probability as:

$$\begin{aligned} p(x_n|C) &\simeq p(x_n|(P_1, P_2, \ldots, P_k)) \\ &= M(p(x_n|P_1), p(x_n|P_2), \ldots, p(x_n|P_k)) \end{aligned} \tag{5.2}$$

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

As Equation 5.2, 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.

5.3.2 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 [147], which achieves a 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 that breaks the method boundary and collects the dependence across the entire program. when encountering a code statement invokes 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 a couple of definitions for the concepts used in our API dependence graph construction.

Definition 5.2 (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 the Node

8 in Figure 5.3.

Definition 5.3 (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 5.3.

Definition 5.4 (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.

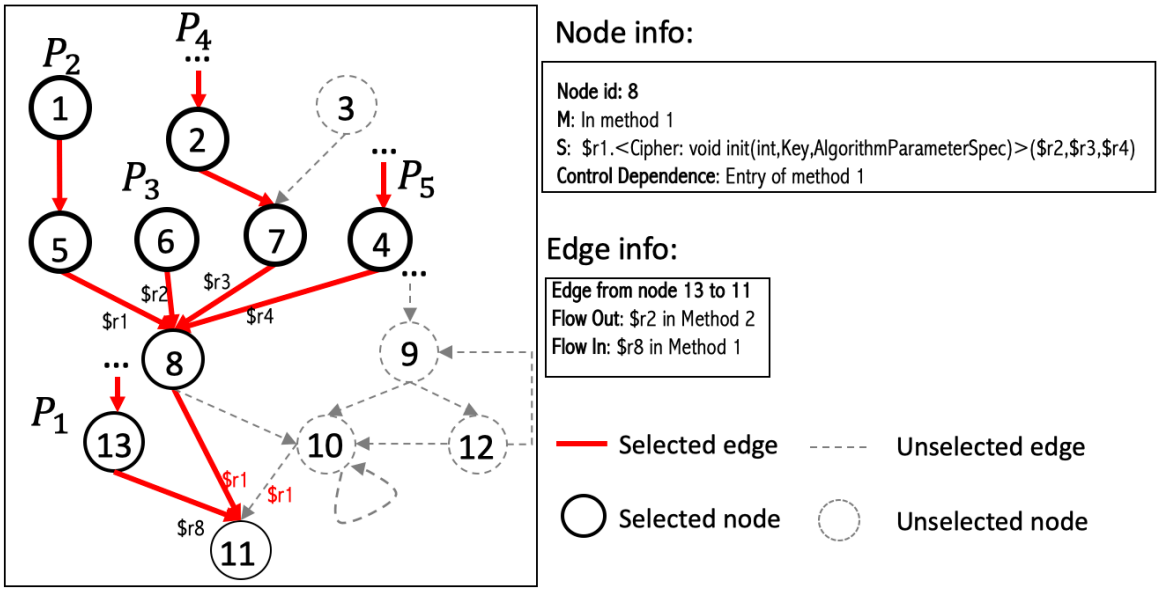


Figure 5.3: Multiple paths selection for API completion. We use the information associated with the nodes and edges to select paths. The goal is to maximize the coverage for different nearby branches with minimal paths.

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 5.5 (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.

Multi-path Selection. We extract multiple API dependence paths from the API dependence graph. To avoid path explosion, we limit the number of selected paths (to five [2] in our experiments) to minimize the cost. We use a greedy strategy to collect paths in an API dependence graph backwardly. The detailed algorithm is shown in the the pseudo code (See Algorithm 1). 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 5.3, we start from node 11. There are three edges to node 11 from nodes 8, 10, and 13. After checking the associated flow-in variables, node 8 and 10 deliver identical variable $\$r1$. Therefore, we select arbitrary one of them. In this example, nodes 8 and 13 that deliver different variables are selected. 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. Now, the selected paths are used as neural network inputs.

[2] We found five is sufficient because more paths bring negligible improvement.

5.3.3 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.

Global dependence enhancing learning

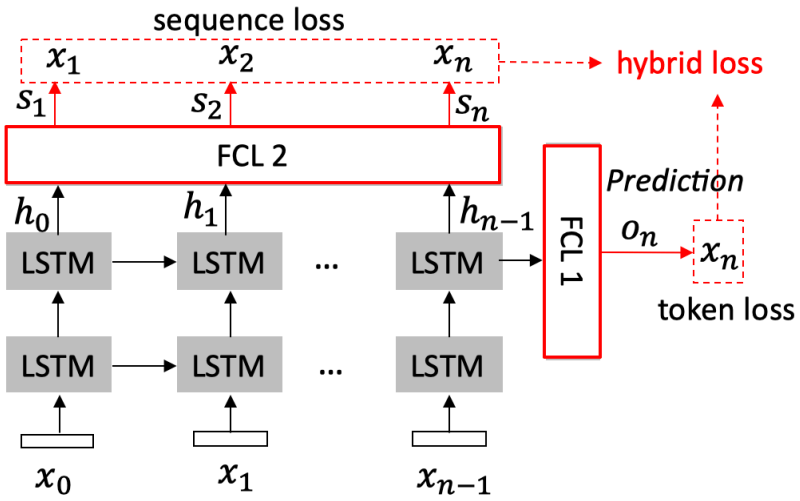


Figure 5.4: Target amplification in HyLSTM with our new hybrid loss function. Through the extra FCL_1 and o_n, similar input sequences followed by different x_n are supervised by stronger signal to highlight their differences.

The purpose of designing HyLSTM is to improve the single-path modeling. As shown in Figure 5.2(a), high-frequency suffix makes the neural network to ignore the beginning global dependencies. Intuitively, we force the model to assign larger weights to beginning tokens (e.g., a_1 in Figure 5.2) 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 the regular LSTM based language modeling in its architecture and

loss function. We illustrate the HyLSTM architecture in Figure 5.4. 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.

Let use h_i 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.

$$\begin{bmatrix} s_1 \\ s_2 \\ \dots \\ s_n \end{bmatrix} = softmax(\begin{bmatrix} h_0 \\ h_1 \\ \dots \\ h_{n-1} \end{bmatrix} W_2 + \begin{bmatrix} B_2 \\ B_2 \\ \dots \\ B_2 \end{bmatrix}) \tag{5.3}$$

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 5.4.

$$o_n = softmax(h_{n-1}W_1 + B_1) \tag{5.4}$$

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, we define a new hybrid loss combining the losses from the two projection layers in Equation 5.5. 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.

The new hybrid loss l_h is defined as:

$$l_h = \alpha L(o_n, x_n) + (1 - \alpha) \sum_{i=1}^{n} \frac{L(s_i, x_i)}{n} \tag{5.5}$$

where $L(\cdot)$ is the cross entropy loss between the output and label. We set the hybrid weight α 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 5.4.

Our multi-path architecture

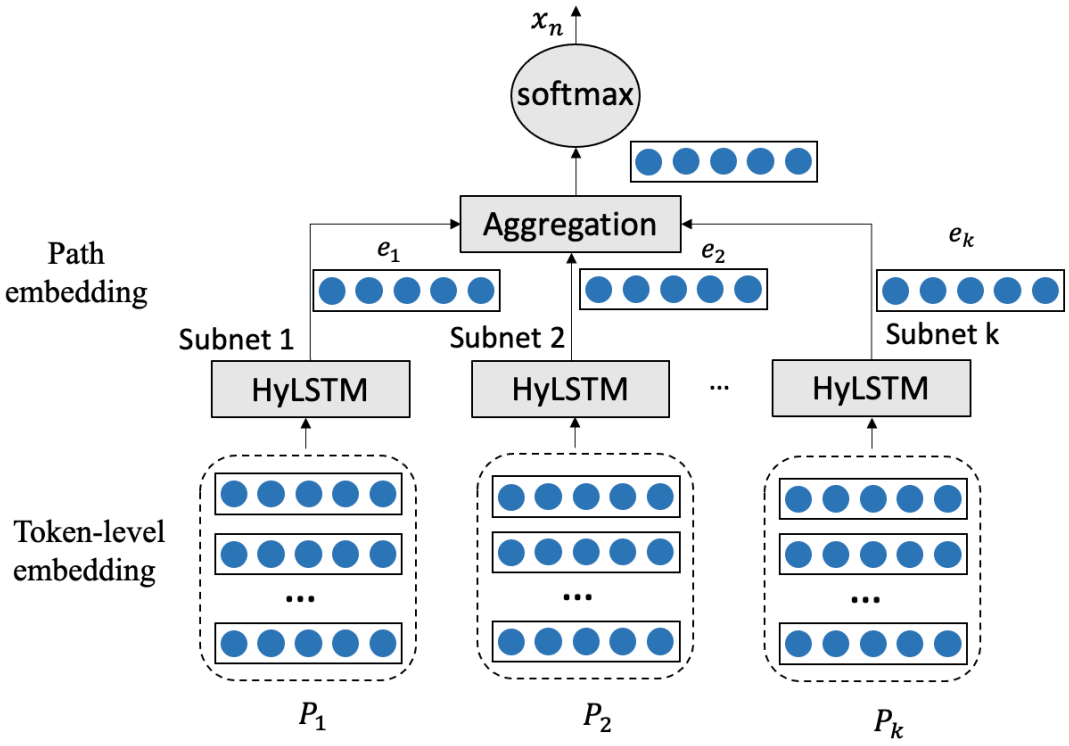


Figure 5.5: Multi-path code suggestion architecture based on aggregating path embeddings

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 5.5, 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.

$$e_i = h_{n-1}W_1 + B_1 \tag{5.6}$$

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:

$$p(x_n|C) = softmax(\sum_{i=1}^{k} e_i/k) \tag{5.7}$$

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 5.4.2.

5.4 Experimental Evaluation

We conduct the 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 [147] and identify 774,460 Java cryptographic API method call-sites 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 [150] and Codota [6]. 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 5.6). 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 [6]. Codota can be used as a plugin in most of the mainstream IDEs. We manually evaluate Codota plu-gin in the IntelliJ IDE on 245 cryptographic API callsites (see Table 5.1) 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 5.5) 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 [123], NSG [121] and the large-scale pretrained programming language model CODEGPT [111]. 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's 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) is not designed for cryptographic APIs, we think their data-driven approach should make them generalize to our domain specific data.

5.4.1 Comparison with Existing Tools

We compare our Multi-HyLSTM with two state-of-the-art API completion tools, SLANG [150] and Codota [6].

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 5.6 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 [3]. Our approach shows significant advantages over

[3]Raychev et al choose 3-gram and hidden layer size 40 for RNN in [150]

SLANG under 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 achieves the best accuracy at 91.41% under Multi-HyLSTM with hidden layer size 512, achieving an improvement by 18% compared with SLANG.

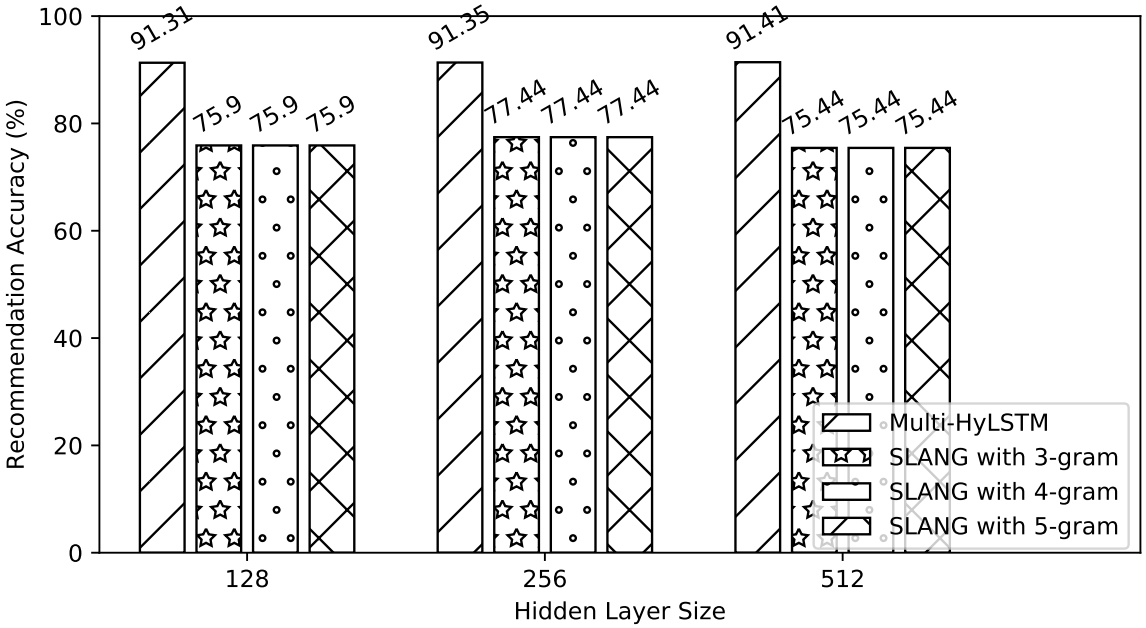


Figure 5.6: The top-1 accuracy of SLANG and our approach on API completion

Table 5.1: The top-1 accuracy of Codota and our Multi-HyLSTM on 245 randomly selected Java Cryptographic API invocation test cases. Codota gives recommendation 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	**Test Cases**	**Apps**	**Codota**	**Multi-HyLSTM (Our approach)**	
				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%**

Comparison with Codota. Codota is a commercial AI code completion plugin, which is

adopted by the 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 removing the method name after the `dot`. Table 5.1 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 5.1) rises to 97.96% if the return type is given, resulting in 51% improvement.

5.4.2 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.

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 5.4, HyLSTM has two parallel projection layers. One produces the token-level loss and the other produces the sequence-level loss. HyLSTM uses a hybrid loss combining both of them. The three models use identical LSTM cells with a hidden layer size of 256.

Table 5.2: The accuracy of HyLSTM for the next API recommendation. Acc.(A) refers to the in-set accuracy for all the test cases. Acc.(K) is the in-set accuracy for the test cases with known features. Acc.(U) is the in-set accuracy for the test cases with unknown features.

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

Table 5.2 shows the *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. Besides, we further analyze the capabilities of the three models by breaking down the test cases into two groups, the test cases with known features and the test cases with unknown 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 with 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.

Table 5.3: 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 5.2. 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, 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

The two groups are both important but reflect different capabilities. The cases with known features is benefited from our program analysis preprocessing. After the program analysis, different source code can result in identical API dependence paths, which significantly reduces the prediction difficulties. However, it is usually more challenging for a neural network to handle unknown dependence paths as the knowledge may not have appeared during training. From Table 5.2, we observe that HyLSTM outperforms the LSTM with token-level loss in terms of both test groups, especially for the test cases with unknown features. HyLSTM substantially improves the accuracy from 43.21% to 56.94%, resulting in a 31.78% improvement. Compared with the LSTM model with sequence-level loss, although HyLSTM sacrifices a little (0.19%) in the accuracy of the test cases with unknown features, it results in a higher overall accuracy. This is because the test cases with known features are the majority group. Therefore, we think HyLSTM achieves an excellent trade-off between the two groups.

Test on new Apps. We want to clarify that our excellent performance on the test cases with known features does not suggest overfitting because our model can successfully handle test cases from 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

Table 5.4: The average 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. (A)	**92.37%**	90.36%	83.99%
Acc. (K)	**99.34%**	98.91%	91.77%
Acc. (U)	**76.40%**	70.78%	66.17%

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 5.3 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 5.7 shows the *in-set accuracies* on the four new test sets. The average results of them are displayed in Table 5.4. Results show that our HyLSTM outperforms the two regular LSTM models on all the four new test sets. On average, HyLSTM achieves an accuracy of 92.37%, which is 9.98% higher than the LSTM model with the sequence-level loss and 2.22% higher than the LSTM with the token-level loss. When looking into the test cases with unknown features, we observe an obvious advantage (7.94% higher) of our HyLSTM over the LSTM model with the token-level loss. An interesting finding is that the accuracy for the test cases with unknown features substantially increases when testing with the new Apps. Our HyLSTM achieves the accuracy for the test cases with unknown features at 76.40%, which is much higher than 56.94% on the original test set.

Case Study 1. This case verifies that our HyLSTM is better at identifying the global dependence that is low-frequency. Figure 5.8 (a) shows a test case predicted incorrectly by regular

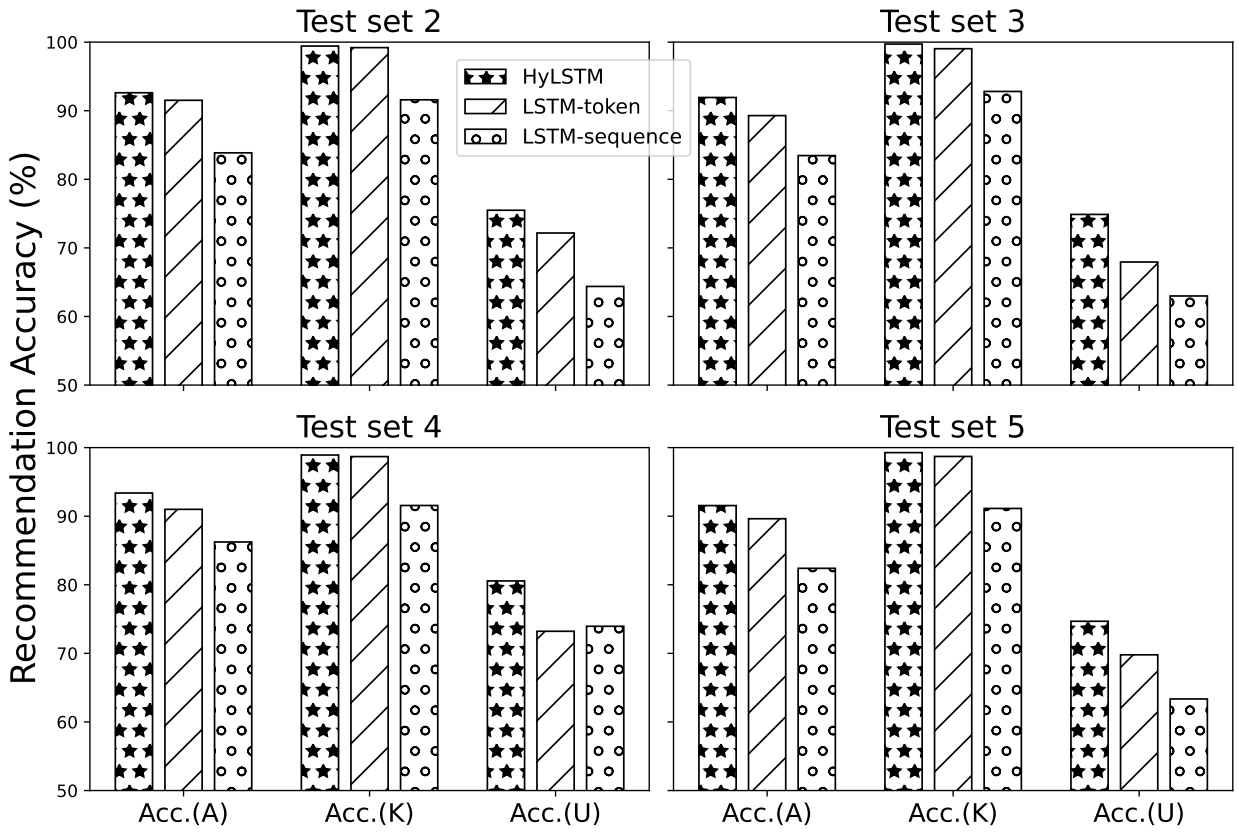


Figure 5.7: The accuracy of HyLSTM and two regular LSTM models tested with four new App sets. LSTM-sequence represents the LSTM model with sequence-level loss while LSTM-token represents the LSTM model with token-level loss.

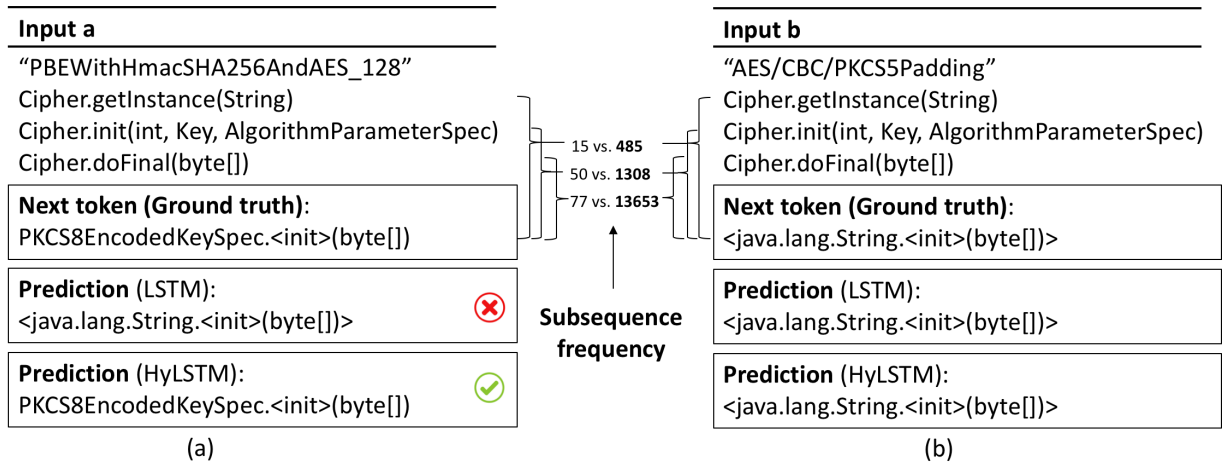


Figure 5.8: Case Study 1, (a) A test case that is predicted incorrectly by regular LSTM and correctly by HyLSTM. (b) A test case that follows the frequent short pattern thus got correct prediction by both regular LSTM and our HyLSTM.

LSTM, but correctly predicted by our HyLSTM. The wrong prediciton of LSTM is due to the more frequent shorter patterns in Figure 5.8 (b). In contrast, HyLSTM successfully differentiates the similar inputs (a) and (b).

Multi-path vs. sequential architecture

We compare our multi-path architecture with their single-path counterparts in API completion. In Table 5.5, Multi-HyLSTM are compared with its single-path version HyLSTM, and other alternative approaches Multi-BERT, where DepBERT replaces HyLSTM. DepBERT is the neural network of BERT pretrained on our dependence paths corpus with the masked language modeling task. To be fair, we also pretrained our HyLSTM on the same dataset. All of these pretrained models are finetuned by our API completion task.

Table 5.5: Comparison between multi-dependence suggestion and sequential suggestion. A, K, and U stand for in-set accuracy for all cases, known cases, and unknown cases, respectively.

	Multi-HyLSTM	**HyLSTM** (path embedding)	**DepBERT**	**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%

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.

Improvement from path embedding. The single-path pretraining can benefit the accuracy, especially for unknown cases. Compared with the basic HyLSTM in Table 5.2, 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.

	Input paths:	Top 3 suggestions by single path
Path 1	"AES" SecretKeyFactory.getInstance(String) SecretKeyFactory.generateSecret(KeySpec)	**Cipher.init(int, Key)** ✗ Mac.init(Key) Cipher.init(int,Key,AlgorithmParameterSpec)
Path 2	"AES/CBC/PKCS7Padding" Cipher.getInstance(String)	**Cipher.init(int,Key,AlgorithmParameterSpec)** ✓ Cipher.getBlockSize() Cipher.doFinal(byte[])
Path 3	16 SecureRandom.nextBytes(byte[]) IvParameterSpec.<init>(byte[])	**Cipher.init(int,Key,AlgorithmParameterSpec)** ✓ AlgorithmParameters.init(AlgorithmParameterSpec) 0
	Next token (Ground truth): Cipher.init(int,Key,AlgorithmParameterSpec)	**Prediction (Multi-path)**: Cipher.init(int,Key,AlgorithmParameterSpec) ✓

Figure 5.9: 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.

Case Study 2. Figure 5.9 demonstrates how the multi-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.

We summarize our experimental findings as follows:

- Our Multi-HyLSTM substantially outperforms the state-of-the-art academic API completion solution SLANG and commercial solution Codota. Multi-HyLSTM achieves an excellent top-1 accuracy of 91.41%, a 18.04% improvement over SLANG. In a manual analysis for 245 test cases compared with Codota, Multi-HyLSTM achieves the top-1 accuracy at 97.96%, a 50.94% improvement over Codota.

- Our multi-path architecture excels at recognizing unseen cases. Multi-HyLSTM and

Multi-BERT improve the in-set accuracy for unknown cases by 11.53% and 36.50% compared to HyLSTM and DepBERT, 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 [147]. Another limitation is that there might be difficulties to apply static analysis on incomplete source code that the code developers are writing in IDEs. The real-world application scenario requires to enable the partial program analysis that can work with the incomplete source code.

5.5 Related Work

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

Treating programs as text. Many studies [17, 48, 61, 95, 163] 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 [17] or more powerful

network models (e.g., LSTM, Transformers, GPT-2) [61] 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 [95, 163].

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 [114, 152]. However, PCFG is found insufficient due to the limited context coverage of ASTs. Bielik *et al.* [34] 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 [107, 188].

Extracting semantic information as context. To generate code following the program semantics, a couple of studies [23, 42, 78, 127] represent programs as graphs. For example, Allamanis [23] build a graph that uses AST as the backbone and add 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) [96]) are incorporated to guide the generation process on graphs or program sketches [42, 121].

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 [128, 130, 134, 150]. 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 object usage model (GORUM) to represent interactions between different objects and associated methods [133].

They built Hidden Markov models for the state of objects and predict methods [130, 134]. However, these methods may require building endless Markov models for different object types. Raychev *et al.* [150] 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 [18, 19, 23, 151]. Allamanis *et al.* defined two tasks VARNAMING [18] and VARMISUSE [23] 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 [51, 57, 68, 163]. These approaches often rely on large language models [43] 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 [137] 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 [154].

5.6 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.

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

Chapter 6

Summary and Future Work

6.1 Conclusion

My thesis work aims to help developers to write secure code by deep-learning based approaches. The ultimate goal is to develop an automatic or semi-automatic secure code generation system. Specifically, we target the Java cryptographic code because Java cryptographic code is reported to be error-prone and result in many vulnerabilities [11, 69, 75, 115, 146]. Cryptographic API misuses, such as exposed secrets, predictable random numbers, and vulnerable certificate verification, seriously threaten software security [27, 72]. We secure the Java cryptographic code by achieving the code completion/suggestion tasks with neural networks. Given the previous code lines, our model generate code by recommending the next API method to use. Trained with the secure code filtered by our vulnerability detector, our model is able to produce secure suggestions for Java cryptographic APIs.

We identified several programming language-specific challenges that require the special attention to solve. First, the API methods and constants are required to be represented as numeric vectors that can be accepted by neural networks. The self-learned representation vectors, known as embedding [117, 118, 119], is significant for the task accuracy [54, 139, 165]. How to design a general-purpose embedding method for programs that captures the code semantics is an important question. Second, we found the LSTM based sequence model is unable to identify the infrequent API sequence patterns, especially when they shares a com-

mon suffix with certain high-frequency API sequences. The high-frequency API sequences would dominate the model and result in wrong prediction for these infrequent API sequences. Finally, we also found there are some functionally-similar but non-interchangeable API methods that hinder the recommendation accuracy. How to make accurate predictions among these similar API methods is another challenge for code generation.

Our dissertation solves these problems with our specialized neural network design and program analysis guided embedding techniques. To guarantee the neural network is trained with secure code, we focus on detecting the cryptographic vulnerabilities. We identify 12 Java cryptographic classes and 18 methods that are often misused by developers. The detection relies on an interprocedual flow-, context-, filed-sensitivity dataflow analysis to extract all the security-critical values used with these API methods. The values that violate security requirements are reported. We realize the cryptographic vulnerability detection under the support of the Oracle bug checker Parfait [56]. The main challenge is to achieve the scalability and accuracy required in the industrial environment. To achieve these requirements, we optimize our dataflow analysis by Parfait's layered framework and specialize the analysis with the refinement insights presented in our prior work CryptoGuard [147].

To reveal the accuracy impacts of code embedding, we extensively compare multiple embedding choices with different program analysis preprocessing. We presented the program analysis-guided embedding strategies to obtain the dependence-aware embeddings. The byte code is processed by the backward interprocedural program analysis to extract the program slices. Based on the program slices, we further construct the API dependence graph to extract the dependence paths of API methods and constants. We train the embedding vectors by applying the skip-gram model on these dependence paths.

We further improve the accuracy of the secure code suggestion by identifying the program dependence specific challenges. Based on the cryptographic vulnerability detector and our

embedding solution, we further invent the highly accurate API method suggestion solution, referred to as Multi-HyLSTM, with the specialized neural network design. Specifically, we invent a new multi-path neural network architecture that is appropriate to predict the API method determined by dependencies from multiple paths. For each path, we introduce a low-frequency long sequence enhancing technique, called HyLSTM, to improve the capability of identifying the infrequent API sequences.

6.2 Future Work

To further help developers to repair cryptographic vulnerabilities in real world, an interesting future work could be the multi-location code repair. From our preliminary experience interacting with developers, one major reason hindering them to apply the suggested fixes is the consistency issues introduced by the one-location code change. Cryptographic functionalities often involve multiple parts working together. A simple update on one of them can cause the functionality failure due to mismatched configurations. To aid this situation, it is important to provide a code repair suggestion including multiple locations that are required to be changed consistently.

Although many existing detection tools [7, 8, 97, 126] offer repair suggestions for cryptographic vulnerabilities, studies show that the suggested fixes in the bug reports are still faraway from the practical demand of developers [70, 191]. Our previous disclosure interactions with open source developers show that many Apache open source developers feel difficult to fix the security issues based on the given suggestions.

The state-of-the-arts [112, 113] generate pattern-based simple code changes to fix different types of cryptographic API misuses. For example, the suggested fix for using a broken cipher algorithm is to replace it with a stronger algorithm (e.g., `AES/CBC/PKCS7Padding`). The

recommended solution for using a constant secret key is to replace the constant key with the randomly generated key bytes. However, these simple suggestions are insufficient to solve the problem due to several reasons. First, the change of a cryptographic API usage may cause new errors due to the inconsistency with other cryptogrpahic settings. In Java, a cryptographic operation often involves multiple classes or interfaces(e.g, `Cipher, SecretKey , etc.`). A change on one of them may require corresponding changes on the other classes. For example, when the cipher algorithm is changed, other algorithm-specific parameters (e.g., keys) should be adjusted accordingly. Second, many cryptographic operations are paired. The security requirement may not meet its semantics. For example, a randomly generated key is considered as secure, however, it is not allowed to randomly generate a key for decryption. These challenges requires us to treat the cryptographic repair more carefully. The consistent code changes in multiple locations should be applied simultaneously.

Our completed works can be the building blocks to achieve the multi-location code repair. The repair system includes three key components, i.e., 1) program analysis, 2) consistency localization, and 3) code variation. The Java program is input to our program analysis process to determine whether it has a cryptographic vulnerability. The program analysis process is also responsible to generate a report with the vulnerable code location and suggested simple fixes for the cryptographic vulnerability. The vulnerable code is passed to the second component, called consistency localization process. The secure code is stored in a secure codebase that can be a training set for next steps. In the consistency localization process, we first apply the simple fix suggestion accompanied with the vulnerability at the vulnerable location. Then, this component is responsible to localize the possible code locations that requires consistency changes. Next, we generate an appropriate code change at one possible consistency location. It is called code variation. Due to there might be multiple locations that needs to be changed. We repeat the three steps for multiple times. Each time

after a variation is applied, the updated code is entered to the program analysis again to extract the vulnerability and dependence information. The consistency locations are ranked and generated for the next step variation on code.

Bibliography

[1] Class Random. https://docs.oracle.com/javase/8/docs/api/java/util/Random.html, 2017. [Online; accessed 29-Jan-2018].

[2] Class SecureRandom. https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html, 2017. [Online; accessed 29-Jan-2018].

[3] NVD: CVE-2019-3795 Detail. https://nvd.nist.gov/vuln/detail/CVE-2019-3795, 2019. [online; Last Modified: 05/20/2019].

[4] *CWE Top 25 Most Dangerous Software Errors*, 2020. URL https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.

[5] *OWASP Top Ten*, 2020. URL https://owasp.org/www-project-top-ten/.

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

[7] Find Security Bugs. https://find-sec-bugs.github.io/, 2021.

[8] SpotBugs: Find bugs in Java Programs. https://spotbugs.github.io/, 2021.

[9] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305. IEEE, 2016.

[10] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the Usability of Crypto-

graphic APIs. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171. IEEE, 2017.

[11] Robert Ackerman. Too few cybersecurity professionals is a gigantic problem for 2019. *Tech Crunch*, 2019.

[12] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17. ACM, 2015.

[13] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 49–61. IEEE, 2019.

[14] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*, 2020.

[15] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation, 2021.

[16] Ferran Alet, Javier Lopez-Contreras, James Koppel, Maxwell Nye, Armando Solar-Lezama, Tomas Lozano-Perez, Leslie Kaelbling, and Joshua Tenenbaum. A large-scale benchmark for few-shot program induction and synthesis. In *International Conference on Machine Learning*, pages 175–186. PMLR, 2021.

[17] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive

scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE Press, 2013.

[18] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293. ACM, 2014.

[19] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49. ACM, 2015.

[20] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100. PMLR, 2016.

[21] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles Sutton. Learning continuous semantic representations of symbolic expressions. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 80–88. JMLR. org, 2017.

[22] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[23] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations (ICLR)*, 2018.

[24] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[25] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[26] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *International conference on machine learning*, pages 245–256. PMLR, 2020.

[27] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering*, 45(12):1170–1188, 2018.

[28] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[29] Gareth Ari Aye, Seohyun Kim, and Hongyu Li. Learning autocompletion from real-world datasets. *arXiv preprint arXiv:2011.04542*, 2020.

[30] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. AutoPandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[31] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. *Advances in Neural Information Processing Systems*, 31, 2018.

[32] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129*, 2016.

[33] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. Neuro-symbolic program corrector for introductory programming assignments. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 60–70. IEEE, 2018.

[34] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: probabilistic model for code. In *International Conference on Machine Learning*, pages 2933–2942. PMLR, 2016.

[35] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow. *If you use this software, please cite it using these metadata*, 58, 2021.

[36] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

[37] Eric Bodden. Inter-procedural Data-flow Analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 3–8, 2012.

[38] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[39] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the association for computational linguistics*, 5:135–146, 2017.

[40] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS*

2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017, pages 71–85. ACM, 2017. doi: 10.1145/3052973.3053004. URL https://doi.org/10.1145/3052973.3053004.

[41] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.

[42] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2019.

[43] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[44] Lutz Büch and Artur Andrzejak. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104. IEEE, 2019.

[45] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. SAR: learning cross-language API mappings with little knowledge. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 796–806, 2019.

[46] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641*, 2020.

[47] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1616–1637, 2018.

[48] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. Syntax errors just aren't natural: improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 252–261, 2014.

[49] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neuralmachine translation. *arXiv preprint arXiv:1810.00314*, 2018.

[50] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Long Xiong Ong. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering*, 2019.

[51] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[52] Mengsu Chen, Felix Fischer, Na Meng, Xiaoyin Wang, and Jens Grossklags. How reliable is the crowdsourced knowledge of security implementation. In *Proceedings of the 41st International Conference on Software Engineering*, pages 536–547. IEEE Press, 2019.

[53] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems*, 31, 2018.

[54] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061*, 2019.

[55] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

[56] Cristina Cifuentes and Bernhard Scholz. Parfait: Designing a Scalable Bug Checker. In *Proceedings of the 2008 workshop on Static analysis*, pages 4–11, 2008.

[57] Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. PyMT5: multi-mode translation of natural language and Python code with transformers. *arXiv preprint arXiv:2010.03150*, 2020.

[58] Dan Cornell. Remediation statistics: what does fixing application vulnerabilities cost. *Proceedings of the RSAConference, San Fransisco, CA, USA*, 2012.

[59] Peng Cui, Xiao Wang, Jian Pei, and Wenwu Zhu. A survey on network embedding. *IEEE transactions on knowledge and data engineering*, 31(5):833–852, 2018.

[60] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[61] Hoa Khanh Dam, Truyen Tran, and Trang Pham. A deep language model for software code. *arXiv preprint arXiv:1608.02715*, 2016.

[62] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

[63] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[64] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[65] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. Unsupervised learning of API aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 745–759, 2019.

[66] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[67] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61, 2012.

[68] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[69] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136. IEEE, 2017.

[70] Felix Fischer, Huang Xiao, Ching-Yu Kao, Yannick Stachelscheid, Benjamin Johnson, Danial Razar, Paul Fawkesley, Nat Buckley, Konstantin Böttinger, Paul Muntean, et al. Stack overflow considered helpful! deep learning security nudges towards stronger cryptography. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 339–356, 2019.

[71] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 254–265, 2016.

[72] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49, 2012.

[73] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE international joint conference on neural networks*, volume 2, pages 729–734, 2005.

[74] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[75] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[76] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[77] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642, 2016.

[78] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[79] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *AAAI*, pages 1345–1351, 2017.

[80] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Deep reinforcement learning for programming language correction. *arXiv preprint arXiv:1801.10467*, 2018.

[81] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. Improved automatic summarization of subroutines via attention to file context. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 300–310, 2020.

[82] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher P Reale, Rebecca L Russell, Louis Y Kim, and Peter Chin. Learning to repair software vulnerabilities with generative adversarial networks. *arXiv preprint arXiv:1805.07475*, 2018.

[83] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[84] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017.

[85] Vincent J Hellendoorn, C Sutton, Rishabh Singh, and P Maniatis. Global relational models of source code. In *International Conference on Learning Representations*, 2020.

[86] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas 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*, pages 163–174. ACM, 2018.

[87] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[88] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[89] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[90] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[91] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. Treebert: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*, pages 54–63. PMLR, 2021.

[92] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[93] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR, 2020.

[94] Rafael-Michael Karampatsis and Charles Sutton. SCElmo: Source code embeddings from language models. *arXiv preprint arXiv:2004.13214*, 2020.

[95] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1073–1085. IEEE, 2020.

[96] Donald E Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[97] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. CogniCrypt: supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936. IEEE, 2017.

[98] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does Cryptographic Software Fail? A Case Study and Open Problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, pages 1–7, 2014.

[99] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[100] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each.

In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[101] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54, 2012.

[102] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*, 2017.

[103] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*, 2016.

[104] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[105] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*, 2018.

[106] Defu Lian, Kai Zheng, Vincent W Zheng, Yong Ge, Longbing Cao, Ivor W Tsang, and Xing Xie. High-order proximity preserving information network hashing. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1744–1753, 2018.

[107] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. Neural code completion. 2016.

[108] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[109] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[110] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. *ACM SIGPLAN Notices*, 51(1):298–312, 2016.

[111] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[112] Siqi Ma, David Lo, Teng Li, and Robert H Deng. CDRep: Automatic repair of cryptographic misuses in Android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 711–722. ACM, 2016.

[113] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H Deng. VuRLE: Automatic vulnerability detection and repair by learning from examples. In *European Symposium on Research in Computer Security*, pages 229–246. Springer, 2017.

[114] Chris Maddison and Daniel Tarlow. Structured generative models of natural source code. In *International Conference on Machine Learning*, pages 649–657, 2014.

[115] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango-Argoty. Secure coding practices in java: Challenges and vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 372–383. IEEE, 2018.

[116] Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocky. Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*, 8(67), 2012.

[117] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[118] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[119] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, 2013.

[120] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[121] Rohan Mukherjee, Yeming Wen, Dipak Chaudhari, Thomas Reps, Swarat Chaudhuri, and Christopher Jermaine. Neural program generation modulo static analysis. *Advances in Neural Information Processing Systems*, 34:18984–18996, 2021.

[122] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural

sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698*, 2017.

[123] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. *International Conference on Learning Representations (ICLR)*, 2018.

[124] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. Source attribution of cryptographic api misuse in android applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 133–146, 2018.

[125] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, pages 935–946, 2016.

[126] Kristen L Newbury. Automated hotfixes for misuses of crypto apis. 2020.

[127] Anh Tuan Nguyen and Tien N Nguyen. Graph-based statistical language model for code. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 858–868. IEEE, 2015.

[128] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny 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*, pages 511–522, 2016.

[129] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A Stitch in Time: Supporting Android Developers in Writing Se-

cure Code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1065–1077, 2017.

[130] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. Learning api usages from bytecode: a statistical approach. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 416–427. IEEE, 2016.

[131] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. Mapping API elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 756–758. IEEE, 2016.

[132] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. Exploring API embedding for API usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 438–449. IEEE, 2017.

[133] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien 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*, pages 383–392, 2009.

[134] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. A statistical semantic language model for source code. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pages 532–542, 2013.

[135] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114, 2016.

[136] Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Inferring crypto API rules from code changes. *ACM SIGPLAN Notices*, 53(4):450–464, 2018.

[137] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[138] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. Usability smells: An analysis of developers' struggle with crypto libraries. In *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*, 2019.

[139] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

[140] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[141] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, page 2227–2237, 2018.

[142] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 447–456. ACM, 2010.

[143] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion

with bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):1–31, 2015.

[144] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[145] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[146] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng 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 (CCS)*. ACM, 2019.

[147] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Daphne 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*, pages 2455–2472. ACM, 2019.

[148] Musfiqur Rahman, Dharani Palani, and Peter C Rigby. Natural software revisited. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 37–48. IEEE, 2019.

[149] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the "naturalness" of buggy code. In *2016*

IEEE/ACM 38th International Conference on Software Engineering (ICSE), pages 428–439. IEEE, 2016.

[150] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *ACM Sigplan Notices*, volume 49, pages 419–428. ACM, 2014.

[151] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from " big code". *ACM SIGPLAN Notices*, 50(1):111–124, 2015.

[152] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. *ACM Sigplan Notices*, 51(1):761–774, 2016.

[153] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[154] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[155] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 311–322. IEEE, 2018.

[156] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[157] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocom-

plete me: Poisoning vulnerabilities in neural code completion. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[158] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[159] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 75–80. IEEE, 2014.

[160] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. Flow2vec: Value-flow-based precise code embedding. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020.

[161] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. A grammar-based structural CNN decoder for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 7055–7062, 2019.

[162] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2727–2735, 2019.

[163] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel 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*, pages 1433–1443, 2020.

[164] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion.

In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 329–340. IEEE, 2021.

[165] Bart Theeten, Frederik Vandeputte, and Tom Van Cutsem. Import2vec: Learning embeddings for software libraries. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 18–28. IEEE, 2019.

[166] Ke Tian, Danfeng Yao, Barbara G. Ryder, Gang Tan, and Guojun Peng. Detection of repackaged android malware with code-heterogeneity features. *IEEE Trans. Dependable Secur. Comput.*, 17(1):64–77, 2020. doi: 10.1109/TDSC.2017.2745575. URL https://doi.org/10.1109/TDSC.2017.2745575.

[167] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014.

[168] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM*, pages 832–837, 2018.

[169] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–312. IEEE, 2019.

[170] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating accurate assert statements for unit test cases using pretrained transformers. *The 42nd International Conference on Software Engineering*, 2020.

[171] Lewis Tunstall, Leandro von Werra, and Thomas Wolf. *Natural language processing with transformers*. " O'Reilly Media, Inc.", 2022.

[172] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[173] Clara Vania and Adam Lopez. From characters to words to in between: Do we capture morphology? *arXiv preprint arXiv:1704.08352*, 2017.

[174] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[175] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *Proceedings of the 29 th USENIX Security Symposium (USENIX) Security*, volume 20, 2020.

[176] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

[177] Ben Wang and Aran Komatsuzaki. Gpt-j-6b: A 6 billion parameter autoregressive language model, 2021.

[178] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[179] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):1–32, 2018.

[180] Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, pages 3034–3040, 2017.

[181] Sam Wiseman and Alexander M Rush. Sequence-to-sequence learning as beam-search optimization. *arXiv preprint arXiv:1606.02960*, 2016.

[182] Ya Xiao, Yang Zhao, Nicholas Allen, Nathan Keynes, Danfeng Yao, and Cristina Cifuentes. Industrial experience of finding cryptographic vulnerabilities in large-scale codebases. *Digital Threats: Research and Practice*, 2020.

[183] Wenkai Xie, Xin Peng, Mingwei Liu, Christoph Treude, Zhenchang Xing, Xiaoxin Zhang, and Wenyun Zhao. Api method recommendation via explicit matching of functionality verb phrases. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1015–1026, 2020.

[184] Frank F Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, 2020.

[185] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[186] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376. ACM, 2017.

[187] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. XLNET: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

[188] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

[189] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[190] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. Learning to represent programs with heterogeneous graphs. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, pages 378–389. IEEE, 2022.

[191] Ying Zhang, Mahir Kabir, Ya Xiao, Na Meng, et al. Data-driven vulnerability detection and repair in java code. *arXiv preprint arXiv:2102.06994*, 2021.

[192] Ying Zhang, Md Mahir Asef Kabir, Ya Xiao, Danfeng Daphne Yao, and Na Meng. Automatic detection of java cryptographic api misuses: Are we there yet. *IEEE Transactions on Software Engineering*, 2022.

[193] Ziwei Zhang, Peng Cui, Xiao Wang, Jian Pei, Xuanrong Yao, and Wenwu Zhu. Arbitrary-order proximity preserved network embedding. In *24th ACM SIGKDD In-*

ternational Conference on Knowledge Discovery & Data Mining, pages 2778–2786, 2018.

[194] Gang Zhao and Jeff Huang. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 141–151, 2018.

[195] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[196] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1296–1310. IEEE, 2019.

[197] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs, 2019.