Go-Clone: Graph-Embedding Based Clone Detector for Golang

Cong Wang
Tsinghua University
Beijing, China

Jian Gao
Tsinghua University
Beijing, China

Yu Jiang
Tsinghua University
Beijing, China

Zhenchang Xing
Australian National University
Australia

Huafeng Zhang
Huawei Technologies
Hangzhou, Zhejiang, China

Weiliang Yin
Huawei Technologies
Hangzhou, Zhejiang, China

Ming Gu
Tsinghua University
Beijing, China

Jiaguang Sun
Tsinghua University
Beijing, China

ABSTRACT

Golang (short for Go programming language) is a fast and compiled language, which has been increasingly used in industry due to its excellent performance on concurrent programming. Golang redefines concurrent programming grammar, making it a challenge for traditional clone detection tools and techniques. However, there exist few tools for detecting duplicates or copy-paste related bugs in Golang. Therefore, an effective and efficient code clone detector on Golang is especially needed.

In this paper, we present Go-Clone, a learning-based clone detector for Golang. Go-Clone contains two modules – the training module and the user interaction module. In the training module, firstly we parse Golang source code into llvm IR (Intermediate Representation). Secondly, we calculate LSFG (labeled semantic flow graph) for each program function automatically. Go-Clone trains a deep neural network model to encode LSFGs for similarity classification. In the user interaction module, users can choose one or more Golang projects. Go-Clone identifies and presents a list of function pairs, which are most likely clone code for user inspection. To evaluate Go-Clone's performance, we collect 6,110 commit versions from 48 Github projects to construct a Golang clone detection data set. Go-Clone can reach the value of AUC (Area Under Curve) and ACC (Accuracy) for 89.61% and 83.80% in clone detection. By testing several groups of unfamiliar data, we also demonstrates the generility of Go-Clone. The address of the abstract demo video: https://youtu.be/o5DogtYGbeo

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**; **Software creation and management**.

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3338996

KEYWORDS

code clone detection, deep neural network, code similarity, go programming language

ACM Reference Format:
Cong Wang, Jian Gao, Yu Jiang, Zhenchang Xing, Huafeng Zhang, Weiliang Yin, Ming Gu, and Jiaguang Sun. 2019. Go-Clone: Graph-Embedding Based Clone Detector for Golang. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293882.3338996

1 INTRODUCTION

As an increasingly popular programming languages used in industry, Golang (short for Go programming language) is a fast and compiled language, which has good execution efficiency on concurrency programming. However, there exist few tools for detecting code duplicates in Golang.

Clone detection techniques have been well developed to assist in detecting code duplicates and copy-paste related bugs[5]. For example, Li et al.[4] present CCLearner, a token-based clone detection approach leveraging deep learning. Their approach applies deep learning on known code clones and non-clones to train models. Koschke et al.[3] propose a clone detection approch using abstract syntax suffix trees. Their approach can find syntactic clones in linear time and space. Jian Gao et al.[1] present Vulseeker, a semantic learning based vulnerability seeker for cross-platform binary.

While the clone detection are successful in traditional programming language such as C and Java, there is no effective clone detector at function level for Golang. The most related works are Dupl[7] and Code Climate[2]. Dupl detects clones on suffix tree for serialized abstract syntax trees (AST). It ignores values of AST nodes and operates with their types. It focuses on clones among code's basic blocks. Code Climate is commercial software, which supports clone detection. Duplication is one of Code Climate engines, which uses a relatively simple algorithm to find similar code snippets. They parse Golang files into abstract syntax trees. When looking for duplication, they compare nodes in AST. Existing Golang clone detection tools are both based on abstract syntax trees. They detect clones at block level, and their methods are sensitive for code structure changes such as code line insertion and deletion which result in Type-3 clones.

In this paper, we present Go-Clone, a learning-based clone detector for Golang. Go-Clone contains two modules: the training module and the user interaction module [1]. In the training module, firstly we parse Golang source code into llvm IR by gollvm[6]. Secondly, LSFG[1] (labeled semantic flow graph) is calculated for each program function automatically. LSFG contains feature vectors of basic blocks, control flow, and data flow information. Then, Go-Clone encodes LSTFs into a dense vector using a deep neural network model. In the user interaction module, users can choose one or more Golang projects. Go-Clone works out a list of function pairs, which are most likely clone code.

To evaluate Go-Clone's performance, we collect 6,110 commit versions from 48 Github projects to construct a Golang clone detection data set. Go-Clone can reach the value of AUC (Area Under Curve) and ACC (Accuracy) for 89.61% and 83.80% in clone detection. By testing several groups of unfamiliar data, we also demonstrates the generility of Go-Clone. Furthermore, Go-Clone is robust to subtle changes in code structures. We demonstrate this robustness with a cloned code pair from our empirical study, which has subtle differences in code structures.

2 GO-CLONE DESIGN

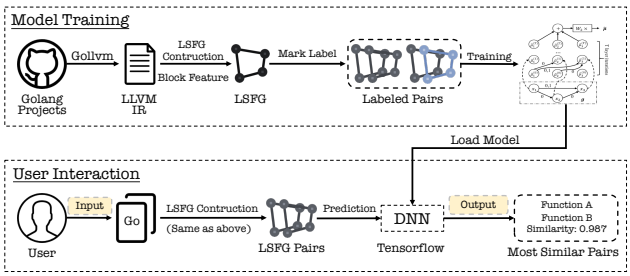


Figure 1: Framework of Go-Clone

Framework Design. Fig. 1 shows the overall framework of Go-Clone. Go-Clone contains two modules: the training module and the user interaction module. In the training module, firstly we parse Golang source code into llvm IR by gollvm[6]. Secondly, LSFG[1] (labeled semantic flow graph) is calculated for each program function automatically. We mark two same functions in different commit version as clone pairs. Otherwise, two functions are non-clone pairs. Then, Go-Clone trains LSFG pairs in a deep neural network model. In the user interaction module, users can choose one or more Golang projects. Same as pretreatment steps in training model, we convert Golang programs into LSFGs. Then LSFGs are put into the trained model to obtain their vector representations. Finally, Go-Clone works out a list of function pairs, which are most likely clone code. The cosine distance between the embedding vectors[1] of two Golang functions are used to measure the function similarity. Details of the Go-Clone's design are illustrated below.

[1]Code of the user interaction module is now open source. You can access it at https://github.com/wangcong15/go-clone

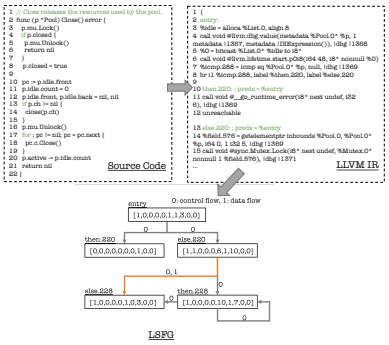


Figure 2: Example: From Source Code To LSFG

LSFG Construction. To calculate the similarity between functions, it is challenging to construct a united expression for every Golang function. In order to fully express the characteristics of the program, the desired united expression should meet three conditions: (1) contains a feature vector for each basic block, (2) reserves control flow information, (3) reserves data flow information. To that end, we propose LSFG, short for "labeled semantic flow graph". Nodes of LSFG are feature vectors for each basic block (introduced next), while edges are control flow and data flow between basic blocks. If two blocks write and read a shared memory location respectively, we create a data flow edge for them. Control flow edges are labeled 0, and data flow edges are labeled 1. We use gollvm[6] to convert the code into llvm IR. IR is a universal language that sits between the high-level program and the low-level backend. IR contains adequate information to construct LSFG. We present an example in Fig.2. The example is a code snippet from a real project. For function "Close", 22 lines of source code turns to 62 lines of IR code. Based on IR's grammar, the function is divided into five blocks. LSFG has six edges. One of the edges is labeled "0,1", while the others are labeled "0" only. "0,1" means that there are both data and control flows between two nodes, while "0" means there is only control flow. The unique data flow edge (red color in Fig. 2) corresponds to the shared memory of variable *p.idle.count*.

Table 1: Dimensions of Block's Feature Vector

Index	Category Name	Example	Remark
1	Terminator Instructions	ret / br / invoke	by llvm
2	Binary Operations	add / sub / mul	by llvm
3	Bitwise Binary Operations	shl / lshr / ashr	by llvm
4	Vector Operations	extractelement	by llvm
5	Aggregate Operations	extractvalue	by llvm
6	Memory Access / Addressing	alloca / store	by llvm
7	Conversion Operations	bitcast .. to	by llvm
8	Other Operations, except for 8/9	icmp / select	by llvm
9	Concurrency Operations	chansend1	Golang
10	Exception Instructions	panic / defer	Golang

Block Feature. Golang, as a language, has its own grammar characteristics. Therefore, it is a problem to customize the design of block feature for Golang. By referring to feature design in previous works[1][4], we decide to use a customized feature vector for Golang. As shown in Table. 1, a block's feature vector has ten dimensions, eight of which are IR instruction categories, while the other two are especially for concurrency and exception instructions.

The reason for this design is mainly to take into account the distinctive features of Golang in concurrency and exception development. For example, the first basic block in Fig. 2 is labeled as "entry". There is one terminator instruction (Line.8-br), one memory access (Line.3-alloca), one conversion operation (Line.5-bitcast) and three other operations(Line.4-call, Line.6-call, Line.7-icmp). Therefore, the block feature of "entry" is [1,0,0,0,0,1,1,3,0,0].

Mark Clone or Non-clone Labels. . To our knowledge, there are no public clone detection datasets for Golang. Therefore, we need to label clone and non-clone function pairs. We collect programs in different commit versions from Github's projects. The procedure converting Golang into llvm IR is at package level, so it is hard to avoid identical functions (completely same) between different commit versions. Therefore we preprocess the code to remove identical functions. We mark the two same functions (by function names) with certain amount of code changes in different commit versions as clone pairs. Otherwise, two functions are non-clone pairs if they are different functions. Possibly there might exist clone pairs of two different functions. We have considered this situation, but its impact could be very small, for the following two reasons: (1) The amount of this situation is very small compared to all non-clone pairs. Randomly, we pick only a small proportion of non-clone pairs (equal to clone pairs), so this situation is unlikely to bring negative impact on our model. (2) We demonstrate Go-Clone's robustness in the part of evaluation. Our model can detect duplicate code in different functions.

Training and Prediction. Go-Clone uses the semantics-aware DNN model presented in[1]. The training procedure is divided into batches. Each batch has ten pairs of training data. The purpose of batches is to improve parameter optimization speed. Model training is done for many times (iteration). After every iteration, we evaluate the model's performance on validation data, and decide whether we need to change the model hyperparameters for the model and restart the model training. After many experiments, we set the number of network layer as five. The embedding size is 64, which means the network can convert LSFG into a 64-dimension vector. This vector is called an embedding vector. When the values of ACC and AUC are stabilized, we save the model parameters. To predict similarities between functions, Go-Clone loads the trained parameters into the model. The trained model calculate the embedding vectors for LSFGs. The more similar the embedding vector is, the more likely the source code is duplicate. We calculate the cosine distance between the embedding vectors[1] to measure the function similarity.

Tool Usage. Go-Clone is a command line tool, implemented in Golang and Python. Golang is used to extract each IR of function. Python is applied for other tasks, such as training, prediction, etc. The overall tool kit contains three instructions: (1) Go2IR: Convert Golang source code into llvm IR files. (2) Go-CloneE: Extract IR code and convert into LSFG. (3) Go-CloneF: Finish clone detection and print clone pairs.

3 EVALUATION

To our knowledge, there are no public clone detection datasets for Golang. Therefore, at the beginning of this section, we describe the construction of our dataset based on 48 projects on Github. We calculate AUC and ACC to evaluate Go-Clone's performance in each training iteration. Then we present the results in clone detection experiment.

3.1 Experiment Setup

Dataset Construction. Manually building data sets is a very time-consuming task. We construct a Golang clone detection data set automatically. Firstly, we collect 6,110 commit versions from Github's 48 projects. All these programs are pre-processed to remove identical functions (completely same). After that, we mark two same functions in different commit version as clone pairs. Otherwise, two functions are non-clone pairs. We randomly pick a subset of the non-clone pairs because the number of non-clone pairs is explosive. Then, the entire data set includes 86,532 function pairs, in which the proportion of clone pairs and non-clone pairs is 1:1. The ratio of training, validation and test data is 10:1:1.

RQ1: Accuracy of clone detection. Randomly, we extract 5,000 training pairs, 500 validation pairs, and 500 test pairs. In these three groups, the proportion of clone pairs and non-clone pairs is 1:1. For the deep neural network, batch size is 10. Meanwhile, in each iteration, all these data are trained or tested.

RQ2: Generality of Go-Clone. In RQ1, Go-Clone is trained by 5,000 training pairs. To prove the generality that Go-Clone works well in detecting clones on other data, we design an experiment. We divide the additional test data by 500 pairs per group. In each group, the proportion of clone pairs and non-clone pairs is still 1:1. Go-Clone has never been exposed to these testing pairs. These groups test the generality of the trained model in RQ1. We collect and compare the results.

3.2 Result

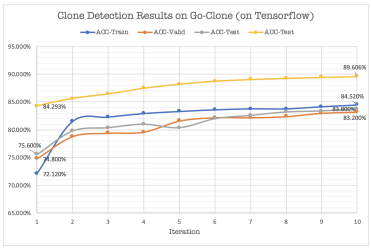
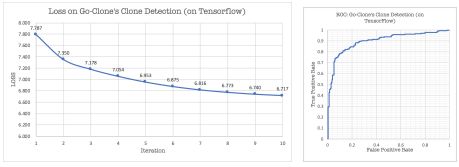
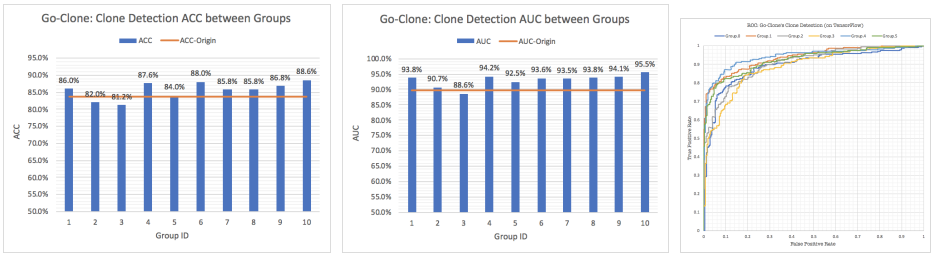


Figure 3: Clone Detection Results on Go-Clone



(a) Loss: Go-Clone's Clone Detection (b) ROC Graph

Figure 4: (a) Loss of Training (b) ROC Curve



(a) ACC between Groups (b) AUC between Groups (c) ROC Curve

Figure 5: (a)(b) Go-Clone's original performance (horizontal line) is intermediate. (c) ROC graphs: Compared to the other 5 test groups, Go-Clone's original performance (in navy blue) is intermediate.

RQ1: AUC and ACC increase during training iterations. Fig. 3 shows four measures in the 10 training iterations. ACC-Train is the accuracy rate for 5000 pairs of training data. The DNN model initializes without any experience, so ACC-Train starts with a low value (72.12%) in the first iteration. After ten iterations, ACC-Train reaches 84.52%, increased by 12.40%. From Fig. 3, we can see that ACC-Valid rises steadily from 75.00% to 83.20%. ACC-Test is the accuracy rate for 500 pairs of test data. In the previous iterations, ACC-Test experiences small fluctuations and finally stabilizes at around 83.80%. AUC-Test is another measure to evaluate the quality of the model. AUC reaches 89.61% after ten iterations. The lower the loss, the better a model. The loss of training decreases steadily from 7.787 to 6.717 (Fig. 4(a)). Meanwhile, Fig. 4 shows the ROC curve of Go-Clone's clone detection performance. As the threshold changes, the true positive rate increase much faster than the false positive rate.

RQ2: Go-Clone is generic. In RQ2, we pick the other test data by 500 pairs per group from the projects that do not contain the 6000 function pairs in the RQ1. Our trained model has not been exposed to this new test data during the model training. Fig. 5(a) shows ACC-Test between test groups. The horizontal lines indicate the original value (83.2%). From the figure, we can see that Go-Clone's original ACC-Test is intermediate among the ten groups. The best performance is 88.6% in Group.10, which is even 3.4% higher. Similarly, Fig. 5(b) shows AUC-Test between test groups. Nine groups of test data have higher AUC-Test than Go-Clone's original performance. Fig. 5(c) shows the comparison between ROC curves of RQ1 and the first new five groups. Go-Clone's original performance (in navy blue) is intermediate. We can conclude that Go-Clone's model also has good performance when meeting unfamiliar data. Therefore, Go-Clone is generic.

Compatible with subtle changes in code structures. An example of cloned code is shown in (Fig. 6), which is detected by Go-Clone. The bodies of the two program functions have 83 and 73 lines of code, respectively. As shown in the figure, statements beyond red rectangles are entirely dissimilar. The two functions set up a session for servers and clients. However, it is straightforward to tell that these two codes are similar through human observation (Even the comments are surely duplicate). Both Golang functions come from a real project. Although two functions have a lot of changes in code structures, Go-Clone can detect them.

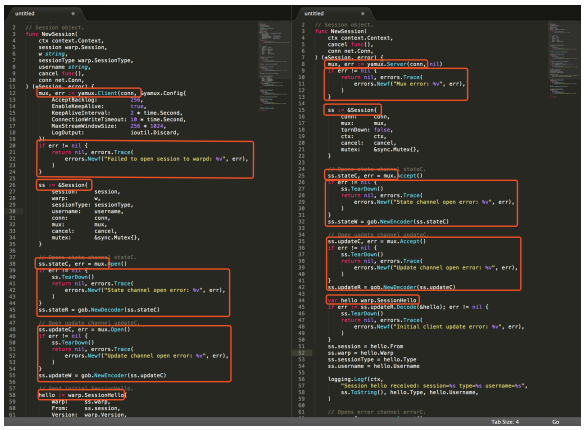


Figure 6: Example of Cloned Code Detected by Go-Clone.

4 CONCLUSION

In this paper, we have presented Go-Clone, a learning-based clone detector for Golang. Go-Clone can work out a list of function pairs, which are most likely clone code. Go-Clone can reach the value of AUC and ACC for 89.61% and 83.80%, respectively. Also we have proved its generality. Based on Go-Clone, we could do exciting things, such as vulnerability search, copy-paste bug search, etc.

REFERENCES

[1] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 896–899.

[2] Bryan Helmkamp, Chris Hulton, and Devon Blandin. 2018. Code Climate. https://docs.codeclimate.com/docs/duplication. [Online; accessed 18-Sept-2018].

[3] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*. IEEE, 253–262.

[4] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 249–260.

[5] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 389–400.

[6] Than McIntosh. 2018. gollvm - Git at Google. https://go.googlesource.com/gollvm/. [Online; accessed 20-Sept-2018].

[7] Mibk. 2018. Dupl. https://github.com/mibk/dupl. [Online; accessed 18-Sept-2018].