Find Me if You Can: Deep Software Clone Detection by Exploiting the Contest between the Plagiarist and the Detector

Yan-Ya Zhang, Ming Li
National Key Laboratory for Novel Software Technology, Nanjing University
Collaborative Innovation Center of Novel Software Technology and Industrialization
Nanjing 210023, China
{zhangyy, lim}@lamda.nju.edu.cn

Abstract

Code clone is common in software development, which usually leads to software defects or copyright infringement. Researchers have paid significant attention to code clone detection, and many methods have been proposed. However, the patterns for generating the code clones do not always remain the same. In order to fool the clone detection systems, the plagiarists, known as the clone creator, usually conduct a series of tricky modifications on the code fragments to make the clone difficult to detect. The existing clone detection approaches, which neglects the dynamics of the "contest" between the plagiarist and the detectors, is doomed to be not robust to adversarial revision of the code. In this paper, we propose a novel clone detection approach, namely ACD, to mimic the adversarial process between the plagiarist and the detector, which enables us to not only build strong a clone detector but also model the behavior of the plagiarists. Such a plagiarist model may in turn help to understand the vulnerability of the current software clone detection tools. Experiments show that the learned policy of plagiarist can help us build stronger clone detector, which outperforms the existing clone detection methods.

Introduction

Software clones are usually introduced when people reuse the code by copy-paste operations (Roy and Cordy 2007) to shorten software development time. To avoid the infringement of copyright, they intend to camouflage the reused code. Such code clones not only violate the intellectual properties of the software being cloned but also lead to the injection of software defects easily due to the lack of detailed knowledge about the cloned code. Thus, software clone detection, aiming to find those plagiarized code automatically has attracted significant attention.

Many clone detection models have been proposed, most of them hand-craft certain similarity between two code fragments by exploiting either lexical information or syntactical information of the code. For example, NICAD (Roy and Cordy 2008) considers the similarity of code fragments in lexical level, Deckard (Jiang et al. 2007) computes the code syntax similarity based on Abstract Syntax Tree. Recent studies suggest to learn semantic features from the code pairs for detecting software clones. For example, CDLH (Wei and Li 2017) formalizes the clone detection as a supervised learning to hash problem and learns deep features for functional clone detection, CDPU (Wei and Li 2018) addresses the clone detection in a positive-unlabeled (PU) perspective to leverage the unlabeled data for functional feature learning.

All of these approaches have shown their effectiveness in software clone detection. However, the clone detection is not a task in a closed and static environment. The plagiarists can easily acquire the techniques of clone detection tools, and by studying the tools they can camouflage the cloned code with particularly designed strategies. Therefore, the software clone detectors may have to deal with new patterns of code cloning. The current data for training the clone detector is always insufficient to identify the new clone patterns, and the clone detector is doomed to be not robust to such adversarial revision of the code.

In fact, there exists a contest between the plagiarists and the clone detectors, where the former will modify the cloned code in order to fool the detectors, and the latter tries to learn the patterns of the modifications made by the plagiarists. Only if such contest is considered in learning the clone detector, the model can be robust to adversarial code revisions in real world.

In this paper, we propose a novel Adversarial Clone Detection approach, namely ACD, to mimic the adversarial process between the plagiarist and the detector. The detector in ACD is an AST-based LSTM to learn the pattern of the code clones. Unlike the existing clone detection approaches, ACD also models the plagiarist based on reinforcement learning, which tries to camouflage the cloned code to make the code pairs appear to be dissimilar and difficult to be detected through a series modifications on the original source code learned using policy gradient (Sutton et al. 2000) and Monte-Carlo search. The plagiarist model and detector model are trained adversarially (Goodfellow et al. 2014), where the generated code clones are used to improve the robustness of the detector while the detection results are used in turn to find better policy for fooling the detector. ACD can not only build a strong clone detector but also model the behavior of the plagiarists. Such a plagiarist model may in turn help to understand the vulnerability of the current software clone detection tools.

Experiments on software clone detection benchmarks indicates that our approach not only can improve the overall detection performance by identifying the clone pairs with some camouflage which are not easily detected by existing clone detection approaches, but also provide some insight on the behavior of the plagiarist, which may help to understand the potential vulnerability of the current clone detectors.

The contribution of our paper lies in three folds:

- We are the first to highlight dynamic nature of software clone detection and the importance of explicitly modeling the contest between the plagiarist and the detector.
- We propose novel adversarial clone detection approach, namely ACD, which is able to mimic the adversarial process in software clone detection by simultaneously learning the a plagiarist model and a detector model.
- The proposed approach can not only build a strong software clone detector which is capable to identify the plagiarized code fragments but also can provide some insight on the vulnerability of the current software clone detection tools from the plagiarist model.

The rest of the paper is organized as follows. We first present the ACD approach, and then we report the experimental results. Finally, we discuss some related work and conclude the paper.

Adversarial Clone Detection

Given n code fragments $\{C_1, \cdots, C_n\}$ where C_i is the i-th raw code fragment, and pairwise labels to indicate whether two code fragments belong to a clone pair of not: $y_{i,j} = 1$ if (C_i, C_j) is a clone pair, $y_{i,j} = -1$ if not, and $y_{i,j} = 0$ if their relation is unlabeled. Moreover, we use C_{i^*} to indicate the modified code after implementing a series of changes on C_i. Our goal is to learn a plagiarist P_θ that can produce deceptive cloned code, and train a detector D_ϕ which maps any pairs of code fragments to $\{-1, 1\}$ to decide whether they belong to a clone pair. Figure 1 summarizes the overall architecture of ACD. The detector D_ϕ is trained by the mixing data from original code $(\{C_1, \cdots, C_n\})$ and modified code $(\{C_{i1^*}, \cdots, C_{im^*}\})$. At the same time, the plagiarist P_θ is updated by employing a policy gradient and MC search on the basis of the expected end reward coming from the detector D_ϕ. As the detector works on the pairwise data, we use fn to represent the true cloned code that is predicted as non-cloned one which means the cloned code generated by our plagiarist can fool the detector D_ϕ. The reward is estimated by the false negative rate FNR, and the specific formulation is given in the next subsection.

The Plagiarist Model

The purpose of plagiarist is to build a model that can act like the real-world plagiarists to produce different cloned code that implement the same functionality. To the best of our knowledge, changing the structure of the code is likely to fool the clone detection model (Mann and Frew 2006). There are three operations to alter the structure of the code, insertion, replacement and deletion, however deletion may change the functionality when some meaningful lines are deleted, therefore, we choose insertion and replacement as optional changes.

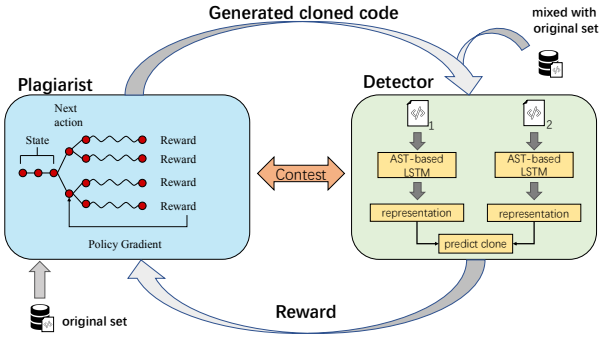

Figure 1: The architecture of ACD.

- **Insertion**: add redundant always-true if statement;
- **Replacement**: extract judging condition and assign it to a new boolean variable, replace the judging condition by this variable.

The specific operations are shown in Figure 2. Additionally, to explore the effect of position where we modified the code, we find eight potential positions to do the changes (as shown in Table 1) via constructing ASTs of the dataset, these positions are very common in the structure of code. The combination of different changes and positions constitutes a modification space $\mathcal{M}$ which contains sixteen distinctive modifications (as shown in Figure 3).

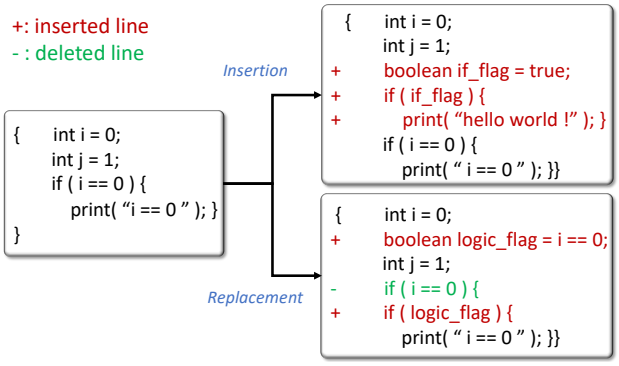

Figure 2: Two types of change: insertion and replacement.

Table 1: Positions to do changes in JAVA and C. Decl stands for Declaration.

JAVA	For	Decl	Do	While	Try	Assignment	Return	If
C	For	Decl	Do	While	Func Call	Assignment	Return	If

In order to generate interpretable cloned code that can be compiled correctly and keep the same functionality, we define some rules to make modifications. First, when we build statement trees from code, we mark the tree nodes by whether it can be the position to do change or not. For example, as shown in Figure 4, we cannot do modifications

Figure 3: The above figure shows the sixteen modifications combined by different types of change and positions in JAVA.

before the “else” statement, if we apply insertion at “else” statement position, it will change the code meaning which is contrary to our intention, and if we apply replacement at the same position, it will cause compiling error directly. Hence, these nodes will be marked as unchangeable when we build the statement trees. Secondly, considering the judging condition statement may contain some variables, in order to keep the functionality unchanged, we have to ensure that the declaration of the new boolean variable which replaces the judging condition inserted into a position after the last assignment of these variables. Besides, to avoid compiled error, the new boolean variable should be inserted into a position before the judging condition statement; After we determine the interval where the new boolean variable can be inserted, we just need to choose a position that allowed to place it.

In the process of generating cloned code, the original code C_i is considered to be the initial state s_0, and the modified code after t step is referred to as state s_t. At each step, ACD first builds a statement tree from current code (as shown in Figure 5), then the plagiarist P_θ offers the probability of which change to be chosen and where it is placed according to the tree, once the modification has been chosen, we apply it on the current code to acquire a new code $C_{i^*}^t$. After the T step, the completely modified cloned code $C_{i^*}^T$ will be generated.

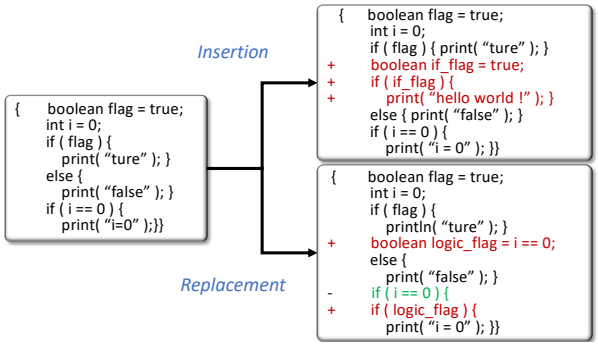

Figure 4: Wrong position to do changes.

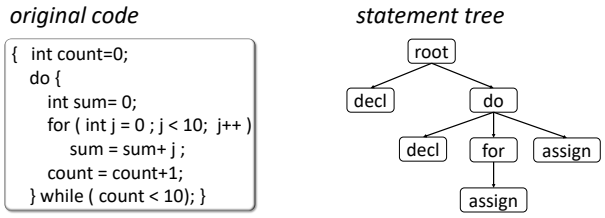

Figure 5: Build statement tree from the original code.

The Detector Model

In this paper, we reuse the AST-based LSTM in CDLH as our discriminative model since it can learn the latent features that characterizing the functionality of the code by simultaneously considering both the lexical information and syntactical structure of code fragments. Specifically, the detector D_ϕ learns a non-linear representation mapping ϕ which transforms code fragments $\{C_i\}^n$ to d-dimensional representation $\{z_i\}^n$.

During the training process, we randomly pick a batch of code fragments $\{C_{i1}, \cdots, C_{im}\}$, where $i1, \cdots, im \in \{1, 2, \cdots, n\}$ as the input of plagiarist, after that the corresponding cloned code $\{C_{i1^*}, \cdots, C_{im^*}\}$ are produced. The cloned code will keep the relation that the original one has (i.e. $y_{i^*,j} = y_{i,j}$, $y_{i^*,j^*} = y_{i,j}$), meanwhile the modified and original code will be labeled as clone pair, set $y_{i^*,i} = 1$ and the detector will be trained on the hybrid new dataset.

ACD via Policy Gradient

Since the detector only evaluate the entire generated cloned code, we follow the (Sutton et al. 2000), when we can not acquire an intermediate reward, the goal of plagiarist (policy) $P_\theta(m_t|C_{i^*}^{t-1})$ is to generate a cloned code from the start state s_0 to maximize its expected end reward.

$$J(\theta) = \mathbb{E}[R_T|s_0, \theta] = \sum_{m_T \in \mathcal{M}} P_\theta(m_T|s_0) \cdot Q_{D_\phi}^{P_\theta}(s_0, m_T), \tag{1}$$

where R_T is the reward for a completely modified code given by the detector D_ϕ. $Q_{D_\phi}^{P_\theta}(s, a)$ is the action-value function which is the expected accumulative reward. Additionally, we use the REINFORCE algorithm (Williams 1992) to estimate the action-value function. As the detector works on the pairwise data, we use FNR as the reward.

$$FNR = \frac{fn}{(tp + fn)}, \tag{2}$$

where fn represents the number of code C_j that is mislabeled as non-cloned with generated code by detector $D_{\phi(C_{i^*}, C_j)}$, $j \in \{1, 2, \cdots, n\}$, and tp stands for the number of code that is labeled as cloned pair correctly. Naturally, we have:

$$Q_{D_\phi}^{P_\theta}(a = m_T, s = C_{i^*}^{T-1}) = FNR. \tag{3}$$

As the detector can only provide a reward for a completely modified code, and we not only care about whether the modification we applied at present is best, but also should consider the future outcome it brings. Thus, we apply Monte Carlo search with a roll-out policy P_β to sample the unknown last $T - t$ modifications on the basis of the current state. In our experiment, the roll-out policy is set the same as the plagiarist. After N-time Monte Carlo search, we represent the N modified code of original code C_i as $\{C_{i^*}^1, \cdots, C_{i^*}^N\}$, where $C_{i^*}^n$ is the code after operating $M_{1:T}^n = \{m_1^n, \cdots, m_T^n\}$ modifications. Accordingly, we represent the FNR of N modified code as follow: $\{FNR^1, \cdots, FNR^N\}$. We run the roll-out policy starting from the current state till the end of the sequence for N times to get a batch of output samples. Thus, we have:

$$Q_{D_\phi}^{P_\theta}(s = C_{i^*}^{t-1}, a = m_t) = \begin{cases} \frac{1}{N} \sum_{n=1}^{N} FNR^n, t < T, \\ FNR, \quad t = T. \end{cases} \tag{4}$$

where, we find that when there is no intermediate reward, we will iteratively generate modifications starting from state $s' = C_{i^*}^{t-1}$ and roll out to the end.

It is beneficial to use the FNR from detector as a reward, because when detector is updated dynamically, the FNR will reflect the update and it can further improve the plagiarist iteratively. Once we generate a set of cloned code (size is m), we will add it to the original training dataset (size is n) and retrain the detector model as follows (Wei and Li 2017):

$$\min_\phi \sum_{i=1}^{m+n} \sum_{j=1}^{m+n} |y_{i,j}|[y_{i,j} - \frac{1}{d}\phi(C_i)\phi(C_j)]^2. \tag{5}$$

We update the plagiarist once acquiring a new detector. The long-term reward can be maximized directly via optimizing the parametrized policy. Following (Sutton et al. 2000), the gradient of the objective function $J(\theta)$ w.r.t the plagiarist's parameters θ can be derived as

$$\begin{aligned} \nabla_\theta J(\theta) = \sum_{t=1}^{T} \mathbb{E}_{M_{1:t-1} \sim P_\theta}[\sum_{m_t \in \mathcal{M}} \nabla_\theta P_\theta(m_t|C_{i^*}^{t-1}) \\ \cdot Q_{D_\phi}^{P_\theta}(C_{i^*}^{t-1}, m_t)]. \end{aligned} \tag{6}$$

Using likelihood ratios (Glynn 1990; Sutton et al. 2000; Yu et al. 2017), we build an unbiased estimation for Eq. (6):

$$\begin{aligned} \nabla_\theta J(\theta) &\simeq \sum_{t=1}^{T} \sum_{m_t \in \mathcal{M}} \nabla_\theta P_\theta(m_t|C_{i^*}^{t-1}) \cdot Q_{D_\phi}^{P_\theta}(C_{i^*}^{t-1}, m_t) \\ &= \sum_{t=1}^{T} \mathbb{E}_{m_t \sim P_\theta(m_t|C_{i^*}^{t-1})}[\nabla_\theta \log P_\theta(m_t|C_{i^*}^{t-1}) \\ &\cdot Q_{D_\phi}^{P_\theta}(C_{i^*}^{t-1}, m_t)] \end{aligned} \tag{7}$$

where $M_{1:t-1} = \{m_1, \cdots, m_{t-1}\}$ is the modifications have been applied on code C_i, and the $C_{i^*}^{t-1}$ is observed cloned code after these modifications. Given the knowledge that the expectation $\mathbb{E}[\cdot]$ can be approximated by sampling methods, we can update the plagiarist's parameters as:

$$\theta \leftarrow \theta + \alpha_h \nabla_\theta J(\theta), \tag{8}$$

where $\alpha_h \in \mathbb{R}^+$ represents the h-th step's learning rate.

In summary, Algorithm 1 shows full details of the proposed ACD. The plagiarist and the detector are trained in turns. We use detector to guide the plagiarist to produce more deceptive cloned code, and in order to keep pace with the progress of plagiarist, the detector needs to be trained with training dataset mixed with new generated cloned code.

Algorithm 1 Adversarial Clone Detection

Require: P_θ: the plagiarist model; P_β: the roll-out policy; D_ϕ: the detector model

Initialize P_θ, D_ϕ with random weights θ, ϕ.
$\beta \leftarrow \theta$
Pre-train D_ϕ
repeat
  for p-steps **do**
    Generate a sequence of modifications $M_{1:T} = (m_1, \cdots, m_T) \sim P_\theta$
    for t in $1:T$ **do**
      Compute $Q(a = m_t, s = C_{i^*}^{t-1})$ by Eq.(4)
    end for
    Update the plagiarist's parameters via policy gradient Eq.(8)
  end for
  for d-steps **do**
    Select m code fragments from dataset and feed them into the plagiarist, mix the generated cloned code with the dataset.
    Train detector D_ϕ for k epochs by Eq.(5)
  end for
  $\beta \leftarrow \theta$
until ACD converges

Experiment

In this section, we conduct experiments on real-world datasets to certify the effectiveness of ACD. We compare ACD with the state-of-the-art clone detection approaches together with self-training model to show the improvement that the ACD can bring. Besides, we design experiments to explore which kind of clone the detector is vulnerable to, finally, we study the performance variations with different parameter settings.

Experimental Setting

We conduct our experiments on two real-world datasets covering different programming languages: BigCloneBench, a widely used benchmark dataset for clone detection (Svajlenko et al. 2014) (with JAVA code fragments) and OJClone from a pedagogical programming open judge (OJ) system [1] (with C code fragments).

BigCloneBench contains projects coming from 25,000 systems, covers 10 functionalities including 6,000,000 clone pairs and 260,000 non-clone pairs. All labeled clone types are given by domain experts. We discard code fragments without any tagged true or false clone pairs and use the remaining 9,134 code fragments.

OJClone (Mou et al. 2016) is consisted of various kinds of source code submitted by students for 104 programming problems. Since we do not have experts to label the data in OJClone, we consider two different source code as a clone pair when they solve the same programming problem because they implement the same functionality. In the experiment, we select the first 15 programming problems and for each problem there are 500 source code files.

Table 2: Overall information of datasets

Datasets	Language	code fragments	AVG length	% data labeled
BigCloneBench	JAVA	9,134	28.60	0.021
OJClone	C	7,500	35.25	0.026

For BigCloneBench, a code fragment is a method, and for OJClone a code fragment is a file. We use javalang [2] to parse JAVA code to ASTs, and apply pycparser [3] to parsing C files to ASTs. Besides, to obtain word embeddings for tokens of code fragments, we use word2vec [4] to generate word embedding of length 100 for both datasets. The overall information for dataset is shown in Table 2. We use precision (P), recall (R), F1 values as performance measurement.

Performance of Clone Detection

In this section, we compare our proposed ACD with following the state-of-the-art clone detection approaches and self-training model to verify its effectiveness as a clone detection tool.

- Deckard (Jiang et al. 2007) clone detection model based on syntax
- SourcererCC (Sajnani et al. 2016) a popular lexical based clone detection
- CDLH (Wei and Li 2017) a supervised clone detection model which learns deep features in an end-to-end way

The overall precision, recall and F1 are displayed in Table 3. It is easy to find that ACD outperforms the other clone detection methods in terms of F1. We can see that ACD improves the recall a lot, considering that the data in BigCloneBench is labeled by experts, they just report the clone pairs that they happen to discover, when the clone pair does not appear or is not labeled as clone by experts, the other approaches are not capable to recognize them while ACD trained by the plagiarist can. As for dataset OJClone, it consists of code submitted by students who used to reuse the same code snippet with slight modifications like classical algorithm bubble sort and so on. These diverse lexical or syntactical modifications make the clone code hard to be detected by other approaches. However, with the help of the plagiarist, our clone detection model can cover more clone types, accordingly, the recall will improve a lot. Among the compared clone detection methods, the CDLH use almost the same deep learning model to detect clone pairs with ACD, we can observe that it beats other clone detection model, but its F1 value is still not as good as ACD's, meaning that the plagiarist can really help to build a strong clone detection model.

Apart from the state-of-the-art clone detection models, we compare the ACD with Self-training approach. In Self-training model, we still use the same detector as ACD, but at

[1] http://programming.grids.cn

[2] https://github.com/c2nes/javalang

[3] https://pypi.python.org/pypi/pycparser/

[4] http://radimrehurek.com/gensim/models/word2vec.html

each epoch, we randomly pick a batch of unlabeled pairs and use the detector to mark them, then we retrain the detector on the new training set and repeat the above operations. The performance of Self-training is shown in Table 3. In BigCloneBench, its precision is very close to 1, we can assume that the label it predicted is correct, but its F1 still cannot compare with ACD, which means the improvement brought by ACD not only thanks to the increase in the amount of labeled data, the more important reason lies in the benefit from the adversarial training.

Table 3: Precision, recall and F1 comparison of all

Approaches	BigCloneBench			OJClone		
	P	R	F1	P	R	F1
CDLH	0.92	0.74	0.82	0.47	0.73	0.57
SourcererCC	0.85	0.02	0.03	0.1	0.75	0.18
Deckard	0.93	0.02	0.03	0.99	0.05	0.10
Self-training	0.96	0.62	0.76	0.17	0.20	0.19
ACD	0.98	0.88	0.92	0.73	0.73	0.73

The vulnerability of the current software clone detection tools

In this section, we design experiments to show which kind of clone may fool the detector. Firstly, we train a detection model on original dataset and then apply it to the newly generated clone pairs. Figure 6 shows the generated code C_{i^*}, and the code written in black is the original one C_i, we can find that compared to the original one, the generated code does not have many differences in lexical. But when we use CDLH to detect clone pairs about C_{i^*}, about 459 clone pairs are labeled as non-clone incorrectly and they can be labeled properly before changed, the total number of clone pairs of this code is 1145. It is hard to explain from a lexical perspective, but when we build the AST of these two code, we can find that they differ a lot which means the alteration of structure is really helpful to camouflage clone code and this kind of clone is hard to detect for current software detection tools.

The Figure 7 displays a clone pair we find from the original dataset, it contains the similar clone type that produced by the plagiarist. Compared to code 2, the code 1 has two more insertions, we apply CDLH and ACD to detect them respectively, CDLH mislabels them as a non-clone pair while ACD marks it correctly, it also proves the traditional detection model is vulnerable to such kind of clone, while the ACD can handle it by understanding the behavior of the plagiarists.

Table 4, 5 gives the probabilities that the plagiarist learned to build deceptive clone code. We can see that the two clone types have the comparable ability to fool the detector. However when it comes to positions, the probabilities differ greatly. In JAVA, when we do modifications at control statements like "Do" and "While", it most likely to deceive the detector, whereas in C, the most deceptive position is "FuncCall" and "Decl". This consequence is interpretable, in JAVA, the structures of control statements are usually more complicated than others, so they are the most

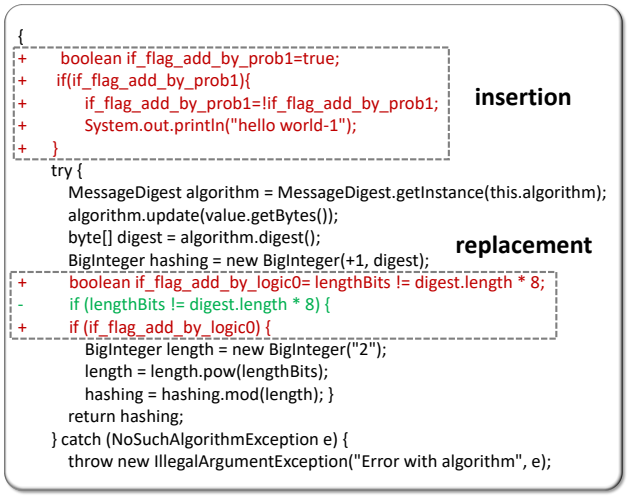

Figure 6: Clone type produced by the plagiarist that is hard to detect.

deceptive positions, while in C, by analyzing the code from dataset, we find that the "Decl" usually appears along with "FuncCall", like "int a = func(0)", and when we build the ASTs of a C code file, the AST of the called function is set as the children node of "FuncCall", so modifying code at these positions may bring pretty good effect.

Table 4: Probabilities of modifications to be chosen.

Language	Insertion	Replacement
JAVA	0.54	0.46
C	0.51	0.49

Table 5: Probabilities of position to be chosen. F, D, W, T, A, R, I, FC stand for For, Declaration, While, Try, Assignment, Return, If and Func Call statement.

	F	D	Do	W	T/FC	A	R	I
JAVA	0.1	0.12	0.19	0.27	0.07	0.09	0.06	0.1
C	0.11	0.24	0.08	0.1	0.2	0.12	0.06	0.09

Sensitivity to Hyper-Parameters

In previous experimental settings, we set the batch size of generated cloned code added to training set 512, and the number of changes applied is 8. Now, we explore how different batch size and number of changes affect on clone detection performance, measured by F1. Figure 8, 9 show the F1 values of ACD with regard to different batch size and number of changes respectively. The batch size ranges from 16 to 512 and the change number ranges from 1 to 8. The Figure 8 shows that the overall performance of ACD is not sensitive to the batch size of generated cloned code in BigCloneBench, while the F1 goes upward generally in OJClone. Moreover, we can find that the performance doesn't change a lot in general over different numbers of applied changes in this range from Figure 9.

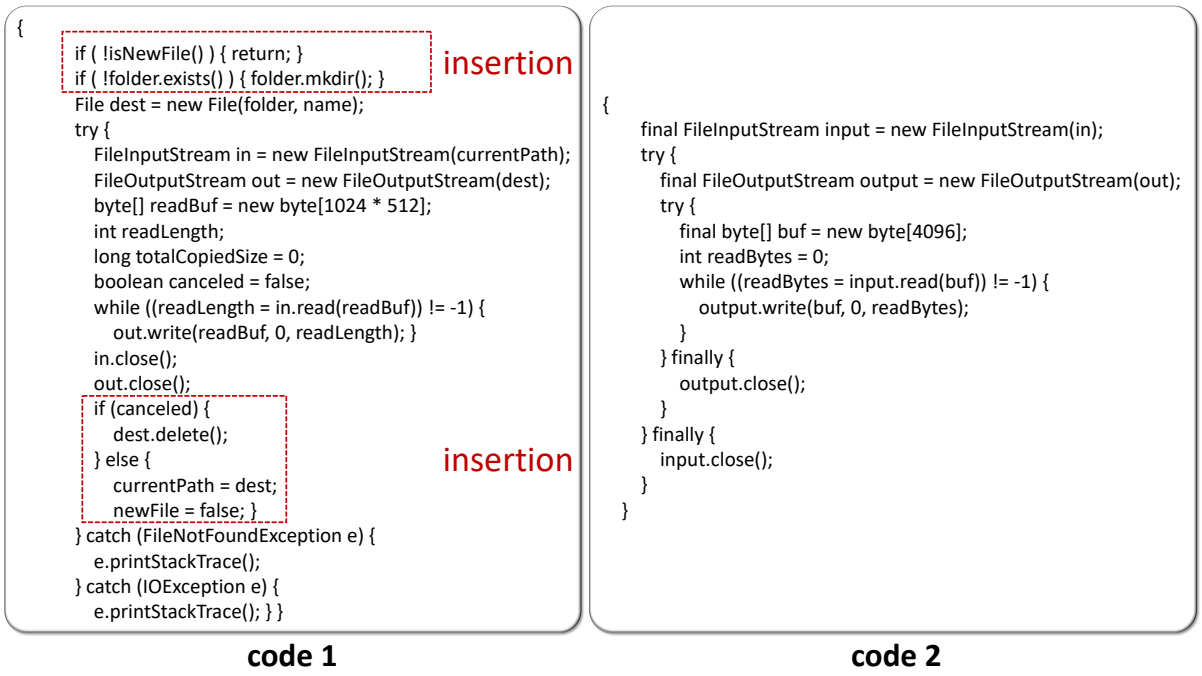

Figure 7: A clone pair from dataset which have the similar clone type produced by the plagiarist.

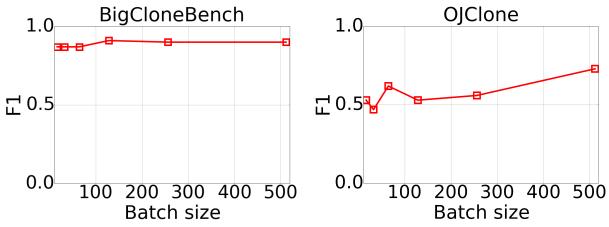

Figure 8: The influence of batch size on ACD.

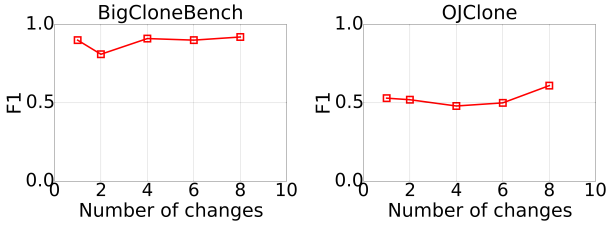

Figure 9: The influence of number of changes on ACD.

Related Work

Software Clone Detection

Copying code fragments and then reuse them by pasting with or without little modifications are common activities in software development and this behavior is called code cloning (Roy, Cordy, and Koschke 2009). Since code clone may easily lead to the injection of software defects or copyright infringement (Brixtel et al. 2010; Baker 1995), software clone detection, aiming to identify similar code fragments, has attracted significant attractions in recent years. Many clone detection approaches have been studied to find clones automatically so far. Deckard (Jiang et al. 2007) introduces AST (Abstract Syntax Tree) to measure the structure similarity of two code fragments. CCFinderX (Kamiya, Kusumoto, and Inoue 2002) and SourcererCC (Sajnani et al. 2016) treat source code as bags of tokens and compare subsequences to detect clones. NICAD (Roy and Cordy 2008) applies slight transformations to code and measures similarity by comparing sequences of text. CDLH (Wei and Li 2017) formalizes the software clone detection as a supervised learning to hash problem and learns supervised deep features in an end-to-end way for software functional clone detection. Recently, CDPU (Wei and Li 2018) has been proposed to formalize the clone detection task as a Positive-Unlabeled (PU) learning problem and leverage the unlabeled data to improve the detection performance.

Sequence Generative Model

Generative models have recently drawn significant attention, and much effort has been made to generate a structured sequence. As pointed out by (Bachman and Precup 2015), the sequence data generation can be formulated as a sequential decision making process which can be potentially solved by reinforcement learning techniques. For most practical sequence generation tasks, e.g. machine translation (Sutskever, Vinyals, and Le 2014), the reward signal is meaningful only for the entire sequence, and in the game of Go (Silver et al. 2016), the reward signal is only set at the end of the Game. In those cases, state-action evaluation methods such as Monte Carlo (tree) search have been adopted (Browne et al. 2012).

Conclusion

In this paper, we deal with the software clone detection problem by exploiting the contest between the plagiarist and the detector, which enables us to not only build strong a clone detector but also model the behavior of the plagiarists. According to human prior knowledge, the plagiarist is able to construct interpretable cloned code without altering the functionality, and such a plagiarist model may in turn help to understand the vulnerability of the current software clone detection tools. The contest with the plagiarist drives the detector to achieve better performance. Experiments on software clone detection benchmarks indicate that the learned policy of plagiarist can help us build stronger clone detector, which outperforms the existing clone detection methods.

Acknowledgement This research was supported by National Key Research and Development Program (2017YFB1001903) and NSFC (61751306).

References

Bachman, P., and Precup, D. 2015. Data generation as sequential decision making. In *Advances in Neural Information Processing Systems*, 3249–3257.

Baker, B. S. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, 86–95. IEEE.

Brixtel, R.; Fontaine, M.; Lesner, B.; Bazin, C.; and Robbes, R. 2010. Language-independent clone detection applied to plagiarism detection. In *Proceedings of 10th IEEE Working Conference on Source Code Analysis and Manipulation*, 77–86. IEEE.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1):1–43.

Glynn, P. W. 1990. Likelihood ratio gradient estimation for stochastic systems. *Communications of the ACM* 33(10):75–84.

Goodfellow, I.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; and Bengio, Y. 2014. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, 2672–2680.

Jiang, L.; Misherghi, G.; Su, Z.; and Glondu, S. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, 96–105. IEEE Computer Society.

Kamiya, T.; Kusumoto, S.; and Inoue, K. 2002. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28(7):654–670.

Mann, S., and Frew, Z. 2006. Similarity and originality in code: plagiarism and normal variation in student assignments. In *Proceedings of the 8th Australasian Conference on Computing Education*, 143–150. Australian Computer Society, Inc.

Mou, L.; Li, G.; Zhang, L.; Wang, T.; and Jin, Z. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, 1287–1293.

Roy, C. K., and Cordy, J. R. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541(115):64–68.

Roy, C. K., and Cordy, J. R. 2008. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, 172–181. IEEE.

Roy, C. K.; Cordy, J. R.; and Koschke, R. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74(7):470–495.

Sajnani, H.; Saini, V.; Svajlenko, J.; Roy, C. K.; and Lopes, C. V. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of IEEE/ACM 38th International Conference on Software Engineering*, 1157–1168. IEEE.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587):484.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 3104–3112.

Sutton, R. S.; McAllester, D. A.; Singh, S. P.; and Mansour, Y. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, 1057–1063.

Svajlenko, J.; Islam, J. F.; Keivanloo, I.; Roy, C. K.; and Mia, M. M. 2014. Towards a big data curated benchmark of inter-project code clones. In *Processings of the IEEE International Conference on Software Maintenance and Evolution*, 476–480. IEEE.

Wei, H.-H., and Li, M. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 3034–3040.

Wei, H.-H., and Li, M. 2018. Positive and unlabeled learning for detecting software functional clones with adversarial training. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2840–2846.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Springer. 229–256.

Yu, L.; Zhang, W.; Wang, J.; and Yu, Y. 2017. Seqgan: Sequence generative adversarial nets with policy gradient. In *Proceedings of the 31th AAAI Conference on Artificial Intelligence*, 2852–2858.