Challenging Machine Learning-Based Clone Detectors via Semantic-Preserving Code Transformations

Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu

Abstract—Software clone detection identifies similar or identical code snippets. It has been an active research topic that attracts extensive attention over the last two decades. In recent years, machine learning (ML) based detectors, especially deep learning-based ones, have demonstrated impressive capability on clone detection. It seems that this longstanding problem has already been tamed owing to the advances in ML techniques. In this work, we would like to challenge the robustness of the recent ML-based clone detectors through code semantic-preserving transformations. We first utilize fifteen simple code transformation operators combined with commonly-used heuristics (i.e., Random Search, Genetic Algorithm, and Markov Chain Monte Carlo) to perform equivalent program transformation. Furthermore, we propose a deep reinforcement learning-based sequence generation (DRLSG) strategy to effectively guide the search process of generating clones that could escape from the detection. We then evaluate the ML-based detectors with the pairs of original and generated clones. We realize our method in a framework named CloneGen (stands for Clone Generator). CloneGen In evaluation, we challenge the three state-of-the-art ML-based detectors and four traditional detectors with the code clones after semantic-preserving transformations via the aid of CloneGen. Surprisingly, our experiments show that, despite the notable successes achieved by existing clone detectors, the ML models inside these detectors still cannot distinguish numerous clones produced by the code transformations in CloneGen. In addition, adversarial training of ML-based clone detectors using clones generated by CloneGen can improve their robustness and accuracy. Meanwhile, compared with the commonly-used heuristics, the DRLSG strategy has shown the best effectiveness in generating code clones to decrease the detection accuracy of the ML-based detectors. Our investigation reveals an explicable but always ignored robustness issue of the latest ML-based detectors. Therefore, we call for more attention to the robustness of these new ML-based detectors.

I. INTRODUCTION

CODE reuse via copy-and-paste actions is common in software development. Such practice typically generates a large amount of similar code, which is often called code clones. According to the large-scale study conducted by Mockus [1], more than 50% of files were reused in some open-source projects. Though code cloning may be helpful under proper utilization [2], it can also become bad programming practice because of the painful maintenance costs [3]. For example, Li et al. [4] reported that 22.3% of operating systems’ defects were introduced by code cloning. Moreover, code cloning also brings intelligence property violations [5], [6], [7] and security problems [4], [8].

In retrospect, source code clone detection is an active research domain that attracts extensive attention. Multiple clone detectors have been proposed based on various types of code representations, including Textual- or Token-based [9], [10], AST-based [11], PDG-based [12], etc. These traditional detectors are primarily designed for syntax-based clone detection (clones with syntactic similarity). Besides, many of them are of limited capability (e.g., specialized for a certain type of clones) and low-efficiency [13], [14].

Recently, the latest machine learning (ML) methods have been significantly enhancing the capabilities of clone detectors. For example, FCDETECTOR [15] trains a DNN network to detect clones of functions by capturing code syntax and semantic information through AST (abstract syntax tree) and CFG (control flow graph). ASTNN [16] maintains an AST-based neural source code representation that utilizes a bidirectional model to exploit the naturalness of source code statements for clone detection. TBCCD [17] obtains structural information of code fragments from AST and lexical information from code tokens and adopts a tree-based traditional approach to detect semantic clones. CCLEARNER [18] extracts tokens from source code to train a DNN-based classifier and then uses the classifier to detect clones in a codebase. These new ML-based approaches fuse the latest ML techniques with the code features extracted from clones, thus achieving highly accurate results. Usually, they can detect most semantic clones with accuracy over 95% [15], [16], [17].
Though the new ML-based clone detectors have shown impressive accomplishments, their effectiveness heavily relies on well-labeled training data [19]. The performance of a detector trained with one dataset can be less effective in detecting code clones in another dataset. Unlike simple texts, source code contains both textual and structural information, which makes ground-truth clone pairs more versatile. Hence, building a robust prediction model for code clone detection is inherently challenging. For example, code fragments int b; and int b;=0; have the same semantics, but their CFG and AST slightly differ. Hence, rather than applying heavy code obfuscations and compiler optimizations, we can merely generate light-weight source code variations at some program locations to effectively lower the accuracy of an ML-based detector? From this point, we investigate and observe that adopting semantic-preserving transformation of a code fragment is of practical importance in validating the robustness of ML-based clone detectors.

Specifically, we present a framework, named CLONEGEN, that performs semantic-preserving code transformations to challenge ML-based clone detectors. In CLONEGEN, we have developed 15 lightweight and semantic-preserving code transformations (e.g., variable renaming, for-loop to while-loop conversion, code order swapping, etc.). In general, CLONEGEN targets the cheap yet effective transformation (or a combination of transformations) on a given code snippet to evade clone detection. To effectively guide the combinations of the 15 atomic transformations, CLONEGEN supports various heuristic strategies (i.e., Random Search, Genetic Algorithm, and Markov Chain Monte Carlo). Essentially, quickly finding the (near-)optimal solutions for evading clone detection is an optimization problem of how to combine multiple transformations. To address this, we design a deep reinforcement learning (DRL) sequence generation model (called DRLSG), which uses a Proximal Policy Optimization (PPO) strategy neural network [20] to guide the search process.

With CLONEGEN, we select unique code snippets from the widely used POJ-104 datasets [21]. Given a selected code snippet $x$, we generate semantic-preserved variants and pair $x$ with each of them. Then, we feed the formed clone pairs to the ML-based detectors to challenge whether they can still determine these code clones produced by our framework. Our approach has the following applications: 1) Source code protection. Semantically equivalent source code can be generated by CLONEGEN to evade detection of the code analysis or search tool. Using our approach in the way of lightweight code equivalent transformation (to some extent like obfuscation) can secure source code in the software supply chain. 2) Constructing datasets for assessing the ML-based clone detectors. Besides, our approach can generate new datasets to facilitate adversarial training against the ML-based clone detectors (especially, the ML-based Type IV clone detector). In such a way, we can push the boundaries of the clone detectors trained on the known datasets and improve their robustness.

Compared with existing works, our work mainly has the following differences. First, the corresponding works [22], [23], [24] are designed to evaluate traditional clone detectors and cannot effectively evaluate existing ML-based clone detectors. For example, the traditional clone detector CCALIGNER [10] achieves a near-perfect recall of 99% on datasets generated by existing clone evaluation tools [22], but only 7% recall on the datasets generated by CLONEGEN. ML-based clone detectors such as ASTNN have been shown to have better accuracy than traditional detector CCALIGNER on existing datasets. Ragkhitwetsagul et al. [25] presented a work Siamese that converts code into multiple representations, which can efficiently handle clones with multiple code modifications in the Ragkhitwetsagul’s [23] dataset OCD. Siamese performs code search on OCD with greater than 95% accuracy. For the dataset provided by Schulze et al. [24] we extracted the functions with more than 5 lines of code, and constructed 912 clone pairs. Then, we trained ASTNN with the code pairs of Type4 in the BigCloneBench [26] dataset and achieved 99% accuracy on these 912 clone pairs. Therefore, the data generated by the existing three works cannot effectively evaluate clone detectors on ML-based. Second, The code equivalence transformation operator we use has been widely used in existing methods [22], [23], [24], our contribution is not to propose these transformation operators. The difference from the previous method is that we propose DRLSG to efficiently combine these operators, assess existing ML-based detectors, and improve these ML-based detectors by adversarial learning. Existing methods simply combine these operators randomly and then transform the code, without considering efficient combination strategies between these operators. Our proposed method guides code clone code generation through reinforcement learning and heuristic search methods and is more suitable for evaluating the robustness of ML-based clone detectors. Finally, the clone code generated by CLONEGEN is designed based on semantic equivalence, which is closer to the definition of a Type IV clone and is more suitable for evaluating ML-based clone detectors that are good at detecting semantic clones. Although the partial equivalence transformation operator used in our paper is the same as existing work [22], [23]. The partial transformation used by existing tool [22] does not necessarily guarantee semantic equivalence of the code. In this study, all the transformation operators (Op1-Op14) except Op15 are strictly semantic preserving. Compared with existing methods [23], we do not propose new obfuscation techniques and we do not perform code transformations according to traditional obfuscation methods.

To evaluate the effectiveness of CLONEGEN, clone pairs generated by CLONEGEN are provided to the three state-of-the-art open-source ML-based detectors (namely, ASTNN and TBCCD), as well as a baseline detector TEXTLSTM [27]. We find that 36.5%, 42.3% and 19.0% of the clone pairs supplied by CLONEGEN can successfully evade the detection of ASTNN, TBCCD, and TEXTLSTM, respectively. Considering the original accuracy of these detectors (more than 95%) on the POJ-104 datasets, experimental results prove that CLONEGEN is notably effective in lowering the accuracy of the ML-based clone detectors via the DRLSG-guided lightweight code transformations. Meanwhile, we further evaluate the effectiveness of CLONEGEN in improving the robustness of ML-based clone detectors via adversarial training. In the best case, adversarial training improved the f-measure of the model by 31.7%. We find among the clone
pairs provided by DRLSG, 8.1% can still successfully escape from the detection of ASTNN after adversarial training.

To summarize, we make the following contributions:

- The design and implementation of CLONEGEN, which supports various heuristic strategies to guide the code transformations and generate semantic-preserving code clones to challenge the clone detectors.
- The proposal of a new DRL-based strategy that dispatches multiple lightweight yet effective code transformation operators at different program locations.
- The evaluation of CLONEGEN upon the ML-based detectors (ASTNN, TBCCD, and TextLSTM) and the traditional detectors (NiCAD and DECKARD). Results show CLONEGEN can sharply decrease the detection accuracy of these detectors.
- A further adversarial training of ML-based detectors using the clones produced by CLONEGEN, which can improve the robustness of the assessed ML-based detectors. Notably, the DRL-based strategy is most effective in lowering the detection accuracy of the adversarially trained detectors.

II. PRELIMINARIES

A. Types of Clones and Clone Detection Methods

**Definition 1:** (Clone Types) According to the definition provided by Bellon et al. [28], code clones are generally categorized into four types, namely Type I, II, III, and IV, which represent the degrees of code similarity between an original code piece and a new one, respectively.

- **Type I:** The two code snippets are mostly identical except for the comments, indentation, and layout.
- **Type II:** The differences between two code snippets are limited in variable names, function names, types, etc., in addition to differences due to Type I.
- **Type III:** The two code snippets have slight modifications such as changed, added, or removed statements, in addition to differences due to Type I and II.
- **Type IV:** Type IV only preserves semantic similarity. Thus, the two code snippets may have functionally similar but different structural patterns.

Type I–III code clones are usually referred to as *syntactic* clone, which copies the code fragments and retains a large body of textual similarity. By In contrast, if the copied code merely exposes functional similarity, such as cloning presents a Type IV clone, a so-called *semantic* clone [28].

Over years, various clone detectors have appeared in the literature [13], [14]. Typically, they convert software code under detection into specific representations and develop methods to distinguish the similarities against the generated representations. Before the emergence of ML-based detectors, traditional detectors (e.g., CCAIGNER [10], DECKARD [11], CGRAPH [12]) mainly rely on Token-, AST-, or PDG-representation to capture the code feature. Usually, they can successfully detect Type I–III syntactic clones, but fail to match Type IV semantic clones or had inefficient detection capability. ASTNN [16] splits the whole AST into small statement trees for the finger-grained encoding of the program lexical and syntactic information. FCDETECTOR [15] adopts a joint code representation of syntactic and semantic features generated from fusion embedding. TBCCD [17] applies a tree-based convolution for semantic clone detection, which leverages both the structural information from AST and the lexical information from code tokens. These ML-based detectors all achieve a stunningly high detection accuracy on the experimental datasets, often with the detection accuracy of more than 95% [15], [16], [17].

B. An Example of Clone Detection

Fig. 1 displays a code snippet *S* and the Type IV and Mutated clones on *S*. Code *S* come from the POJ-104 [21] database. It is to find the minimum number of notes of different currency denominations (e.g., \{100, 50, 20, 10, 5, 1\}) that sum up to the given amount (i.e., the input variable n). All the clones in Fig. 1 are solutions to the same problem, but with different levels of syntactic changes. Specifically, Type IV instance is more compact—it uses division instead of continuous subtraction to obtain integer quotients at line 11, and finish the calculation and printing in one for a loop.

For this motivating example, the well-trained ASTNN, TextLSTM, and TBCCD can all correctly match the Type IV code (ref. code snippets #2 in Fig. 1) with the original code (ref. code #1 in Fig. 1) – ML-based detector success in identifying the syntactic and semantic clones. By far, these ML-based detectors seem to be quite effective, but can they thoroughly address the semantic clone detection problem?

C. Motivation

With the above question, we manually create a clone code by applying several lightweight code transformations. Given the code snippet #1 in Fig. 1, after changing the variable name *k* at line 4, splitting large constants into smaller ones at line 3, swapping the code order at lines 3,4, removing part of the code comments at line 5, and converting a *for-loop* to *while-loop* at lines 7,8,9,12, we generate the code snippet #3 in Fig. 1).

Then, we construct two clone instances as \{#1, #3\} and \{#2, #3\} from pairing the new code #3 with existing code pieces in Fig. 1. Now, we run ASTNN, TextLSTM, and TBCCD on these new clone instances. We are curious to see whether a few uncomplicated code variants could lead to some different findings. Surprisingly, experimental results show that all three detectors failed to classify all two code pairs to be code clones. This interesting result implies that some cheap code-level changes could indeed nullify the DNNs in these detectors, without using dedicated adversarial samples or heavy code obfuscation (e.g., obfuscation based on encoding [29], [30], [31] and CFG-flattening [32], [33]). This observation motivates us to investigate the following questions to facilitate the automation of adversarial code clone generation:

- Can code clones from lightweight semantic-preserving transformations steadily invalidate the detection of ML-based detectors?
- What kind of transformation strategy is needed to guide the effective searching process of combining these semantic-preserving transformations?
Can we leverage the new clone instances from the semantic-preserving transformations to improve the ML-based detectors and would the improvements be explicitly beneficial?

To answer these questions, we propose and implement the clone generation framework CLONEGEN.

III. CLONEGEN: AN OVERVIEW

In this section, we describe the overall workflow for evaluating ML-based clone detectors, and explain the three technical challenges we must address.

A. The Overview of CLONEGEN

Fig. 2 illustrates the overview of CLONEGEN. In general, CLONEGEN consists of two phases: the clone generation phase and the detector evaluation phase. The former takes a source code snippet as input, performs code equivalence transformations, and outputs the mutated code snippets. The latter evaluation phase feeds the original code snippets together with the newly generated ones to a set of state-of-the-art clone detectors and outputs whether the new clone pairs could be detected by these detectors.

For example, given code #1 in Fig. 1, the steps of producing code snippet #3 by CLONEGEN are as follows. First, CLONEGEN extracts syntactic features of code snippet #1 and searches for the locations where code transformation could be applied (see Section IV-B). Second, it adopts the predefined 15 atomic transformation operators to make sure that all the performed changes preserve the semantics of the original code (see Section IV-A). Third, CLONEGEN also adopts a certain transformation strategy that properly adjusts the probability of activating an individual transformation operator (see Section IV-C). Finally, by applying the transformation operators in the order determined by the searching strategy, new code that is more likely to escape from existing detectors can be generated (see Section IV-D). Till now, the clone generation phase is finished.

B. Technical Challenges

While the high-level idea appears to be straightforward, there are three challenges to address for realizing CLONEGEN:

- **Transformation Operator:** In this work, we require equivalence transformations, which slightly modify the code syntactically but leave the code semantics intact. Therefore, the operators in this study are different from mutation operators in mutation testing [34]. Mutation testing mutates source code for inserting small faults into programs to measure the effectiveness of a test suite. So mutators in mutation testing usually change code semantics during generating new code. However, we aim to achieve semantic equivalence transformations. Towards this goal, we adopt many transformation operators proposed in [35] and [22]. In total, CLONEGEN supports 15 unique transformation operators, including variable renaming, changing syntax structures with equivalent semantics, adding junk code, deleting irrelevant code, reordering independent code, etc. Details and examples can be found in Section IV-A.

- **Encoding Schema:** Before applying transformation operators, it is required to localize potential code places applicable to proper operators. For example, we must locate all the for-loop in code before we can decide whether to execute the operator of Op2 (see Op2 in Table I). Thus, the code fragments that satisfy the transformation conditions are abstracted to execute the encoding. The abstraction

---

**Fig. 1.** Motivating example of code clone at different levels. All the clone instances are of the same semantics.

**Fig. 2.** CLONEGEN system overview.
TABLE I
THE DESCRIPTIONS AND EXAMPLES OF 15 ATOMIC TRANSFORMATION OPERATORS IN CLONEGEN

| Transformation Operator | Description | Original | Changed |
|-------------------------|-------------|----------|---------|
| Op1-ChRename            | Function name and variable name renaming | int i; | int i1; |
| Op2-ChFor               | The for-loop is transformed into a while-loop. | for(i=0;i<10;i++){
|                         | BodyA
|                         | } | i=0; while(i<10){
|                         | BodyA
|                         | i++; |
| Op3-ChWhile             | The while-loop is transformed into a for-loop. | while(i<10){
|                         | BodyA
|                         | } | for(i<10){
|                         | BodyA
|                         | } |
| Op4-ChDo                | The do-loop is transformed into a while-loop. | do{
|                         | BodyA
|                         | while(i<10){
|                         | BodyA
|                         | } | BodyA while(i<10){
|                         | BodyA |
| Op5-ChIfElseIf          | Transformation of if elseif to if else. | if[grad<60] BodyA else if[grad<60] BodyB else BodyC | if[grad<60] BodyA else if[grad<60] BodyB else BodyC |
| Op6-ChIf                | Transformation of if else to if elseif. | if[grad<60] BodyA else{ if[grad<60] BodyB else BodyC } | if[grad<60] BodyA else if[grad<60] BodyB else BodyC |
| Op7-ChSwitch            | Transformation of the Switch statement to the if elseif statement. | switch(a){ case 60: BodyA |
|                         | case 70: BodyB default: BodyC } | if(a==60) BodyA else if(a==70) BodyB else BodyC |
| Op8-ChRelation          | Transformation of relational expressions. | a>b | b>a |
| Op9-ChUnary             | Modifications to unary operations. | i++; | i=i+1; |
| Op10-ChIncrement        | Modifications to incremental operations. | i=1; | i=i+1; |
| Op11-ChConstant         | Modifying constants. | (b-a)/4 |
|                         | | |
| Op12-ChDefine           | Modifications to variable definitions. | int b=0; | int b,b=0; |
| Op13-ChAddJunk          | Adding junk code. | if(a){
|                         | Body(A) } | if(a) { BodyA |
|                         | } | if(a) return 0; |
| Op14-ChExchange         | Change the order of the statements in a block without data and control dependency. | a=b+10; c=d+10; | c=d+10; a=b+10; |
|                         | | |
| Op15-ChDeleteComments   | Deleting statements that print debugging hints and comment. | printf("test.c ");
|                         | /*comments */ | printf("test.c ");
|                         | /*comments */ |

Symbols: \( \rightarrow \) means syntax transformation, \( \downarrow \) means evaluation operation, Change function is a bijection function, Dependency is a data and control dependency statements set in a block, Comments is a comment statement set, \( \text{Printf} \) is a printf call statements set with constant arguments.

needs to be fast and scalable as we target the large codebase in evaluation. To address this issue, CLONEGEN develops a compact bitvector-based representation for encoding the search space of changing a program function. Specifically, suppose a function owns \( n_v \) variables (i.e., function name, function arguments, local variables, and used global variables), \( n_f \) for-loop statements, \( n_w \) while-loop statements, \( n_d \) do-while statements, \( n_i \) if-else statements, \( n_s \) switch statements, \( n_r \) relational expressions, \( n_u \) unary operations (e.g., \( i++ \)), \( n_c \) self-changing operations (e.g., \( i=++1 \)), \( n_d \) constant values, \( n_d \) times of variable definition, \( n_b \) code blocks (\{\ldots\}), \( n_i \) blocks of isomorphic statements that have no dependency in the control flow, and \( n_p \) print and comment statements. The length of the encoded bit vectors would be the sum of these values:

\[
I_b = \sum(n_v, n_f, n_w, n_d, n_i, n_s, n_r, n_u, n_c, n_d, n_b, n_i, n_p)
\]  

During the generation phase, if a binary bit is enabled to be 1, CLONEGEN will apply the corresponding transformation operator to the code location. However, it is still a difficult problem to have an optimal strategy of applying operators that match specific code structure for an equivalence transformation — generating semantic clones with various optimization goals. Thus, we need a transformation strategy.

- **Transformation Strategy.** As mentioned above, after defining transformation operators and encoding the search space, there is another challenge: how should we decide the chance of being enabled (i.e., 1) for each bit after encoding, to maximize the diversity of the generated semantic clones – to challenge the existing ML-based clone detectors? Generally, it appears to be a combinatorial optimization problem with a huge search space (for example, it is \( 2^{45} \) for the code snippet #1 in Fig. 1) that could be addressed by commonly-used heuristic strategies, such as Random-Search (RS), Genetic Algorithm (GA) [36], Markov chain Monte Carlo (MCMC) [37], and the recently popular Deep Reinforcement Learning (DRL), etc. These strategies usually need some goals or objective functions (e.g., the fitness function in GA). Regarding this challenge, in Section IV-C, for the commonly-used strategies (RS, GA and MCMC), we implement these strategies to guide the transformation based on some existent studies [38], [39], [40]. However, it is not straightforward to leverage DRL for this problem, for which we propose the DRLSG model in Section V.

IV. CLONEGEN: TECHNICAL APPROACH

In this section, we elaborate the major steps of CLONEGEN.

A. Transformation Operator

CLONEGEN transforms the given program source code by a set of atomic operations. These code transformations must not
change the semantics of a program. Thus, we formally define these operations as equivalence transformations.

**Definition 2:** Equivalence Transformation. Let \( C \xrightarrow{\tau} C' \) be a transformation (\( \tau \)) of a code snippet \( C \) into a code snippet \( C' \) based on the combinations of a set of atomic code transformation operators \( S_o \). The pairs \( C, C' \) form a pair of semantic clones.

Table I summarizes all the atomic transformation operators in CLONE\_GEN. The first column lists the operator names, the second column briefly describes how each atomic operation takes effect, the third column shows a simple code example before equivalence transformation and the last column gives the transformed code after the transformation. Fig. 3 Representation of successive formulations of code transformations. We have implemented these operators based on TXL.\(^1\)

Transformations based on these 15 operators defined in Table I is lightweight, as only lexical analysis in TXL is adopted. Since there are no complicated operations on CFG, PDG, or call graph (CG), the transformation only works at the syntax level. Meanwhile, the transformation manages to make semantic equivalence — all the operators guarantee to have the same semantics before and after the transformation. Even for Op4 that changes a do-while-loop statement into a while-loop, the body statement inside the do-while-loop will be executed once before entering the transformed while-loop. For Op14, we only reorder the statements without any data- or control-dependency, as such analysis is fast and easy. For Op15, we only delete comments or statements that print debugging hints or intermediate results. For the rest of the operators, it is straightforward that these operators would not change code semantics (see operational semantics defined for these operators in Fig. 3).

The generated code is syntactically validated using pycparser so that the generated code is syntactically correct. Most of our proposed operators use the statement as the basic unit of transformation, whereas the insertion of junk code uses the block of code (the enclosing code fragment) as the basic unit of transformation. The junk code \( (\text{if}(0) \ \text{return} \ 0;) \) is inserted randomly inside each block as a standalone statement, not randomly inside any statement, and therefore does not violate the syntax rules of the code, and it does not create the gaped clone. For the output deleting of Op15, our idea is to remove printf statements that contain only debugging information, the impact of which is that the user can’t see the debugging information result, but will not change the internal program logic or fail any test cases.

**B. Encoding**

Before code transformation, it is required to identify syntactic features, the places of code fragment applicable to an atomic operation in Table I. We use the term feature count to represent the number of code fragments on which CLONE\_GEN performs the corresponding atomic operations. Table II illustrates the feature and its feature count of code #1 in Fig. 1. The first and second column corresponds to the atomic operations and the denoting variables in Table I and the third column corresponds to their values. The last column lists the bitvector after encoding code #1 in Fig. 1.

\(^1\)TXL: a programming language that converts input to output by a set of conversion rules, https://www.txl.ca/txl-abouttxl.html.
TABLE II
SYNTACTIC FEATURES AND ENCODING OF CODE #1 IN FIG. 1

| Operator | Count | Value | Bitvector Encoding |
|----------|-------|-------|--------------------|
| Op1      | $n_\text{w}$ | 5     | 00100              |
| Op2      | $n_\text{f}$ | 4     | 1111              |
| Op3      | $n_\text{w}$ | 0     | n.a.              |
| Op4      | $n_{\text{do}}$ | 0   | n.a.              |
| Op5      | $n_{\text{w}}$ | 0     | n.a.              |
| Op6      | $n_{\text{w}}$ | 0     | n.a.              |
| Op7      | $n_{\text{w}}$ | 0     | n.a.              |
| Op8      | $n_{\text{w}}$ | 4     | 0001              |
| Op9      | $n_{\text{w}}$ | 4     | 0110              |
| Op10     | $n_{\text{sc}}$ | 1     | 0                 |
| Op11     | $n_{\text{ec}}$ | 18    | 100011000000001011 |
| Op12     | $n_{\text{d}}$ | 2     | 01                 |
| Op13     | $n_{\text{e}}$ | 5     | 00000              |
| Op14     | $n_{\text{e}}$ | 2     | 10                 |
| Op15     | $n_{\text{p}}$ | 0     | n.a.              |

in the second column, where an atomic operation would take effect on the features (i.e., applicable places) for bit value 1, and no effect for 0. The symbol “n.a.” means there are no available features that match the corresponding atomic operation, and the feature count must be 0. We explain how to fill the bit vector with various strategies in the next section.

C. Transformation Strategy

After the encoding step, CLONEGEN needs a transformation strategy to generate the semantic clones of diversity.

As discussed earlier, the search problem of finding an optimal sequence of applying the atomic operations can be treated as a combinatorial optimization problem. Thus, we may achieve different solutions by exploring existing optimization algorithms. In CLONEGEN, we have already implemented three commonly-used heuristic strategies, namely Random-Search (RS), Genetic Algorithm (GA), and Markov Chain Monte Carlo (MCMC). Details of each strategy are as follows:

Random-Search: The RS strategy fills the bits in a bit vector with random values 0 or 1, with equal probability. Hence, this strategy does not take into account the source code structure under detection and the features employed by different clone detectors. Hence, the RS strategy favors generating semantic clones whose equivalent code changes are attributed to randomness [41].

Genetic Algorithm: The GA [42] strategy computes the similarity between the original code and the generated new code after each random generation. The idea is to generate the code that exhibits the most textual differences from the original code. Here we convert both code versions into strings and then calculate the string editing distance [43] between them to obtain the code similarity. An editing distance reflects how many modification steps it needs to convert one string into another, and it is used as the fitness function in GA. Hence, the GA strategy favors generating semantic clones with big gaps (large editing distance) [39], [40]. The iterative evolution of the genetic algorithm prefers to select offspring with larger edit distances for preservation, so the genetic algorithm produces code with larger differences in edit distance. Instead of preferring several big code gaps, the GA approach prefers modifying every single possible statement to maximize edit distance.

Markov Chain Monte Carlo: The MCMC strategy uses the $n$-gram algorithm to calculate the probability that determines how likely a sub-sequence follows its prefix. It is observed that software programs have probabilistic natures, which assign probabilities to different sequences of words [38], [44]. For a given sequence of code snippet $s = s_1s_2\cdots s_n$, we define $P(s_n | s_{n-k+1}, \ldots, s_{n-1})$ to approximate the probability of code statement $s_n$ occurring after statements $s_{n-k+1}$ to $s_{n-1}$. Inspired by [45], we also adopt the indicator perplexity [45], to measure the probability magnitude of code occurrence. According to $n$-gram ($n = k$), the formula corresponds to:

$$H_M(s) = -\frac{1}{n} \sum_{i=1}^{n} \log p_M(s_i | s_{i-k+1}, \ldots, s_{i-1}) (\text{Perplexity})$$

According to the above formula, this strategy guides code transformations through several iterations and outputs the generated clones satisfying the above perplexity condition. Hence, the MCMC strategy favors generating semantic clones hard to understand (being highly perplexing) [38].

Though the above three strategies are in general effective, they would not take into consideration the feedback (e.g., detection results) of the assessed clone detectors. To mitigate this limitation, we specially design and implement a DRL-based method, named DRLSG, to interactively incorporate the detectors’ results for generating further complicated semantic clones (see Section V).

D. Clone Generation

As shown in Fig. 2, CLONEGEN performs a lightweight transformation phase guided by different strategies. For example, there are four for-loop in the original code of Fig. 1. The last column of Table II shows the bit vectors from the RS strategy. As stated in Section IV-B, Bit value 1 in a bit vector means that CLONEGEN must execute the code mutation operation at the corresponding code fragment, while bit value 0 means CLONEGEN should keep the original code piece unchanged. Since all bits for the operator, Op2 has the value of 1, CLONEGEN transforms the four for-loop into semantics preserving while-loop, as shown by code #3 in Fig. 1. Due to the randomness of these strategies, we could obtain a large number of code variants (e.g., code #3 to code #1 in Fig. 1). For the strategies other than RS (i.e., GA, MCMC, and DRLSG), not all code variants will be retained — for effectiveness, only those satisfying certain optimization goals (e.g., perplexity) will be retained.

Notably, our atomic operations are lightweight: according to our experiments (as described in Section VI), the transformation operations can be completed within a relatively short time. Most of our proposed operators use the statement as the basic unit of transformation, where the insertion of garbage code uses the block of code (the enclosing code fragment) as the basic unit of transformation. The rubbish code (if(0) return 0;) is inserted inside each block as a complete statement, not randomly at any
point in the code, and therefore does not break the syntax of the code.

V. DRLSG: THEORY AND DESIGN

In this section, we introduce the background knowledge of reinforcement learning and our proposed Deep Reinforcement Learning Sequence Generation (DRLSG) strategy.

A. Deep Reinforcement Learning

In recent years, deep reinforcement learning (DRL) algorithms have been used in a variety of fields [46], [47], [48], most notably AlphaGo [49] to defeat the best human players in Go, and DRL has quickly become the focus of the artificial intelligence community. This paper is based on deep reinforcement learning to generate high-quality source code for transformation sequences. Before we start presenting our approach, we briefly introduce background knowledge about DRL. The following formulation of a typical DRL process is related to the presentation given in [50].

We briefly introduce the terms commonly used in DRL:

Agent: The role of a learner and decision-maker. Environment: Everything that is composed of and interacts with something other than the agent. Action: The behavioral representation of the agent body. State: The information that the capable body obtains from the environment. Reward: Feedback from the environment about the action. Strategy: The function of the next action performed by the agent based on the state. on-policy: The policy that corresponds to when the agent learns and the agent interacts with the environment is the same. off-policy: The policy when the agent is to be learned and the agent interacting with the environment are not the same.

In reinforcement learning, a policy indicates what action should be taken in a given state and is denoted by $\pi$. If we use deep learning techniques to do reinforcement learning, the strategy is a network. Inside the network, there is a set of parameters, and we use $\theta$ to represent the parameters of $\pi$ [51]. We take the environment output state ($s$) and the agent output action (a), and string $s$ and $a$ all together, called a Trajectory ($\tau$), as shown in the following equation: $\tau = \{s_1, a_1, s_2, a_2, \ldots, s_t, a_t\}$. You can calculate the probability of each trajectory occurring:

$$p_\theta(\tau) = p(s_1) \prod_{t=1}^{T} p_\theta(a_t | s_t) p(s_{t+1} | s_t, a_t)$$ (2)

The reward function determines how many points are available for said action now based on a certain action taken in a certain state. What we have to do is to adjust the parameter $\theta$ inside the actor so that the value of $\tilde{R}_\theta = \sum \tau R(\tau)p_\theta(\tau)$ is as large as possible. We use gradient ascent because we want it to be as large as possible. We take a gradient for $\tilde{R}$, where only $p_\theta$ is related to $\theta$, so the gradient is placed at $\theta$.

$$\theta \leftarrow \theta + \eta \nabla \tilde{R}_\theta$$ (3)

$$\nabla \tilde{R}_\theta = E_{\tau \sim p_\theta(\tau)} \left[R(\tau) \nabla \log p_\theta(\tau)\right]$$ (4)

Proximal policy optimization (PPO) [20] is a variant of policy gradient. Using $\pi_\theta$ to collect data, we have to sample the training data again when $\theta$ is updated. So we want to go from on-policy to off-policy. We change on-policy to off-policy by importance sampling, from $\theta$ to $\theta'$. So now the data is sampled with $\theta'$, but the parameter to be adjusted for training is the model $\theta$.

$$\nabla \tilde{R}_\theta = E_{\tau \sim p_\theta(\tau)} \left[ \frac{p_\theta(\tau)}{p_\theta'(\tau)} R(\tau) \nabla \log p_\theta(\tau) \right]$$ (5)

B. Design of DRLSG

As shown in Fig. 4, our DRLSG model has two main components: agent and environment. The agent mainly consists of a neural network, and here we choose the openAI open source PPO model as our agent. The input to the agent is the encoding vector (state) of the code, and we train the agent to choose the corresponding transformation operator Section IV-A, to maximize the reward value. The environment consists of three main parts: the code transformation according to the action chosen by the agent, the code is encoded to obtain the vector as the state of the environment, and the reward is calculated for the current decision. Next, we describe the specific design of the DRLSG.

1) Action Space: In our DRLSG task, the agent is trained to select the action to be performed from the action space given a state. We use the 15 transformations designed as the action space of the model, and in our task, the goal of the model is to select a set of operations from the 15 transformations to transform the source code. These transformations are shown in Table 1, where we give descriptions of the specific operations and simple examples, and in Fig. 3, where we give formal definitions of the corresponding equivalent transformations.

2) Reward: The reward function is designed to guide the overall actions of the agent and reward is the key to the DRL, we need to maximize the cumulative gain of the agent. The main goal of our agent is to produce variant codes that can escape detection by MI-based clone tools. For this purpose, our reward function for each step is formalized as Algorithm 1.

---

2Stable Baselines3 is a set of reliable implementations of reinforcement learning algorithms in PyTorch, https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html.

---

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
Algorithm 1: Reward Algorithm.

Input:
- action: The current action performed by the agent.
- code_t: The original code.
- code_t−1: The current action gets the code snippet of the previous state.

Output:
- reward

1. if clone(code_t, code_t−1) then
2.   reward = ndone_R
3. else
4.   reward = clone * codeTextSim(code_t, code_t−1)
5. end
6. if action = Op13 then
7.   reward = reward * Op13_count * Op13_penalty
8. end
9. Op13_count++
10. end

As shown in Fig. 4, our reward has two main components that make up the current action to get the edit distance between code_t and code_t−1, and the current action to get the clone detection result between code_t and code_o. They constitute our reward, and we are considering a long-term reward. We implement a text-based Siamese network source code clone detection model, using two identical bidirectional LSTM (BiLSTM) networks in Siamese’s architecture. The code_t and code_o are converted into text sequences and fed into BiLSTM to obtain a representation of the code. If the TextLSTM identifies the code_t and code_o as non-clones, we assign a large positive reward (Line 1-3 in Algorithm 1) and abort the current learning, indicating that we have obtained a transformed sequence of the current code.

The edit distance indicates the minimum number of single-character editing operations required to convert from one word to another. There are three types of editing operations: insert, delete, and replace. For example, for the words “kitten” and “sitting,” the minimum single-character editing operations required to convert from “kitten” to “sitting” are: substitution of “s” for “k,” the substitution of “i” for “e,” and insertion of “g” at the end, so the edit distance between these two words is 3. We implemented a text sequence-based code text similarity based on edit distance Code Text Similarity (codeTextSim), defined as follows:

\[ \text{codeTextSim} = 1 - \frac{\text{editDistance}(\text{code}_t, \text{code}_{t-1})}{\max(\text{len(code}_t), \text{len(code}_{t-1}))} \]

where len denotes the length of the code sequence, and if the edit distance between two codes is 0, then their similarity is codeTextSim = 1.0, which means there is no difference between the two codes. If the action executed by the current state model yields code_t that does not change relative to the code_{t−1} of the previous state, then their codeTextSim = 0. We give it a negative penalty of \(\text{clone}_R \times \text{codeTextSim}\) (Line 4-6 in Algorithm 1). This is designed to avoid the rewards in the environment being too sparse, to help improve their learning efficiency and converge as quickly as possible [52], since there is no limit to the amount of garbage code that can be inserted. To control the code complexity, we use \(\text{Op13}_{\text{count}}\) to count the number of times op13 is used in the current round and give an additional negative penalty \(\text{Op13}_{\text{penalty}} \times \text{Op13}_{\text{penalty}}\) (Line 7-10 in Algorithm 1) if the action selected by the current agent is op13. In our implementation, we set \(\text{ndone}_R, \text{clone}_R,\) and \(\text{Op13}_{\text{penalty}}\) to 10, 0.5, and 0.5 empirically. When a trained DRLSG model generates a clone code, it repeatedly selects a code conversion operator to convert the code until the maximum reward is obtained or the maximum number of iterations is reached.

3) State: In DRL state represents the current state of the environment. In our model, the transformation code obtained from the current action represents our current state, which can be represented by the encoding vector of the code. For a given source code we convert it into a sequence of text sequence and then encode it with a bidirectional LSTM model, which is the same as the clone detection model mentioned in Section V-B2.

VI. Evaluation

In this section, we aim to investigate the following four research questions (RQs) through the experimental evaluation:

- **RQ1**: How effective are the different transformation strategies? How robust are the existing ML-based detectors in detecting the semantic clones generated by CLONEGEN?
- **RQ2**: How effective is our proposed DRLSG when detection results are available? Can the ML-based detectors be enhanced by adversarial training with CLONEGEN?
- **RQ3**: How effective are different types of atomic transformation operators?
- **RQ4**: How accurate are the traditional clone detectors in detecting the semantic clones produced by CLONEGEN?

A. Implementation and Experimental Setup

1) Datasets for CLONEGEN: In our evaluation, we use POJ-104 [21], a widely used database in source code study [15], [16], [17], [21], [53], [54]. POJ-104 consists of 104 folders, each of which contains 500 solutions (clones) to the same problem. The code from each POJ-104 folder to form an initial dataset \(D_1\), and apply four strategies (RS, GA, MCMC, and DRLSG) to \(D_1\) to construct four new datasets, denoted as \(D_{RS}, D_{GA}, D_{MCMC}\) and \(D_{DRL}\), respectively.

Table III shows the time consumed to transform 52,000 source code snippets in the POJ-104 dataset by four transformation strategies. For each code snippet in the 52,000, we apply the four different strategies to generate code pairs. Notably, the RS strategy generates one code variant for each snippet, while the other three strategies would generate many clone candidates and then use the corresponding optimization goal to retain the best one. Finally, we have the same size for these generated datasets: \(|D_1| = |D_{RS}| = |D_{GA}| = |D_{MCMC}| = |D_{DRL}|.

The RS strategy takes only 0.48 hours, which is the fastest. The GA and MCMC strategies need to do some code syntax analysis when transforming the code, so they are slower...
than the RS strategy. The DRLSG strategy requires an ML-based encoder to encode the source code and an ML-based detector to perform similarity analysis, and only one operator is selected for each transformation, so it is much slower.

2) Assessed Clone Detectors: There are many ML-based C++ source code clone detectors, including [15], [16], [17], [21], [53], [54], [55], [56], [57] etc. However, some of them are not open source, and some others have implementation issues when handling our dataset (e.g., FCDetector [15]). Finally, we adopt two open-source detectors ASTNN [16] and TBCCD [17]. We also apply the TextLSTM [27] model on the POJ-104 dataset. In the training/testing steps, we follow the guidelines provided in their GitHub pages, and use the suggested parameters in their papers [16], [17] to make fair comparisons. Here, TextLSTM is a model with a text embedding size of 300 and a BiLSTM hidden layer size of 256. During the training process, we divided the dataset into training, validation, and test sets according to 80%, 10%, and 10%, and we used the Adam optimizer with a learning rate of 0.001 to train 20 epochs and save the model with the best F1 during the training. The above hyperparameters are empirically determined according to our experiments.

The detailed results are shown in Table V. ASTNN and TBCCD are tested by our replication. They both perform well on POJ-104 with F1 greater than 0.96. POJ-104 has a big difference and belongs to the type IV clone, which means that the existing ML-based clone detectors have achieved good results, here the TextLSTM F1 (0.991) is the best. Table VI summarizes the time spent on setting up the three ML-based detectors. TextLSTM spends the least time on data pre-processing and ASTNN spends the least time on training and testing. TBCCD takes the maximum time, about 1.38 hours, in pre-processing and 52 hours in training and testing.

3) Baseline Transformation Strategies: As introduced in §IV-C, the search problem can be treated as a combinatorial optimization problem. Thus, we may achieve different solutions by exploring various optimization algorithms. In this paper, we also compare our proposed DRLSG with the commonly-used heuristic strategies such as RS, GA, and MCMC. We have derived the parameters for GA from the related work of [39], [40], [42]. For MCMC, we have referred black to [38], [58] for the parameter setup.

4) Evaluation Metrics: After obtaining the four new datasets, we feed them to clone detectors under test. Notably, to evaluate the robustness of the ML models inside the detectors, the instances in our dataset have two types of labels: clone and non-clone. For example, the generated dataset $D_{RS}$ consists of 52,000 clone pairs (each one from the original $D_I$ and the other one generated by the RS strategy) and also 52,000 non-clone pairs (code pairs from different folders transformed after the RS strategy). The 52,000 non-cloned pairs were randomly selected from the codes after the transformation.

We use Precision ($P$), Recall ($R$) and F1-Measure ($F_1$) to measure the performance of the ML-based detector. Let FN denote false negative, FP denotes false positive, TN denotes true negative, and TP denotes true positive, we calculate $P$, $R$, and $F_1$ as follows:

$$P = \frac{TP}{TP + FP} \quad R = \frac{TP}{TP + FN} \quad F_1 = \frac{2PR}{P + R} \quad (6)$$

We use Recall ($R$) to measure the performance of traditional tools on detecting code clones. The recall is an evaluation indicator commonly used by these detectors (e.g., [10], [59]).

5) Environment: We conduct all the experiments on an AMAX high-performance computing server. It has two 2.1GHz 24-core CPUs, four NVIDIA GeForce RTX 3090 GPUs, and 384G memory.

B. RQ1: ML-Based Detectors Versus CloneGen

In this section, we answer the RQ1, the effectiveness of CloneGen on bypassing the ML-based clone detectors. Towards this goal, we use the initial datasets $D_I$ as the training dataset for three assessed detectors. Then, for each different strategy, we use the corresponding generated dataset ($D_{RS}, D_{GA}, D_{MCMC}, D_{DRL}$) as the testing dataset.

Next, we evaluate the robustness of these detectors against the code pairs generated by CloneGen. Table IV shows the average accuracy of the three detectors, Fig. 5 shows the results of five experiments, different search strategies get corresponding sub-optimal solutions, so the results of ML-based clone detectors will fluctuate for different datasets. Generally, these ML-based detectors have varying accuracy regarding the four transformation strategies.

TextLSTM directly converts the code into a sequence of texts, and then feeds the code into an LSTM neural network to obtain the embedding vector of the code, and then feeds the vector into a shallow feedforward neural network to make a binary classification judgment, TextLSTM has the best clone detection performance on the $D_I$ with an $F_1$ of 0.991 in Table V. The $F_1$ of TextLSTM ranges from 0.502 to 0.882 for four different strategies, all of which affect TextLSTM compability to the $F_1$ on the original dataset (0.991). Among the four strategies, DRLSG (0.653) is the best in bypassing TextLSTM detection, while RS (0.865) is the worst, indicating that random transformations are more easily detected by TextLSTM.

ASTNN is a neural network built upon the AST of a program. The $F_1$ of ASTNN ranges from 0.577 to 0.735. In all four code transformation strategies, the DRLSG strategy performs the best and the RS strategy performs the worst. The result shows that different strategies can extensively bypass the AST-based detection in ASTNN.

TBCCD uses structural AST and token lexical information to generate a tree-based convolutional neural network. According to Table IV, TBCCD has better robustness than ASTNN and TextLSTM, with an average $F_1$ value of 0.852. TBCCD performs best against MCMC with an $F_1$ of 0.894, indicating that TBCCD is relatively insensitive to MCMC-guided transformation. The code clone pairs generated from the DRLSG strategy achieve $F_1 = 0.810$ on TBCCD, followed by GA (0.814) and RS (0.894). This result indicates that the token information increases the resilience of ML-based detectors against code transformation.
TABLE IV
RESULTS OF THE ML-BASED DETECTORS IN DETECTING CLONE PAIRS GENERATED BY CLONEGEN

|               | TextLSTM       | ASTNN          | TBCCD          |
|---------------|----------------|----------------|----------------|
| Test Data     | P   | R   | F1   | P   | R   | F1   | P   | R   | F1   |
| $D_{RS}$      | 0.869| 0.861| 0.865±0.017 | 0.876| 0.633| 0.735±0.023 | 0.927| 0.854| 0.889±0.003 |
| $D_{GA}$      | 0.782| 0.762| 0.772±0.065 | 0.860| 0.585| 0.696±0.021 | 0.833| 0.794| 0.814±0.046 |
| $D_{MCMC}$    | 0.858| 0.848| 0.853±0.020 | 0.873| 0.604| 0.714±0.029 | 0.862| 0.930| 0.894±0.015 |
| $D_{DRL}$     | 0.652| 0.618| 0.635±0.047 | 0.519| 0.446| 0.577±0.047 | 0.848| 0.774| 0.810±0.036 |

Fig. 5. F-measures from different experiments.

TABLE V
DETECTION ACCURACY OF THE ML-BASED DETECTORS ON POJ-104

| Tools       | P   | R   | F1   |
|-------------|-----|-----|------|
| TextLSTM    | 0.991| 0.991| 0.991 |
| ASTNN       | 0.996| 0.960| 0.977 |
| TBCCD       | 0.987| 0.994| 0.990 |

TABLE VI
TIME CONSUMPTION OF THE ML-BASED CLONE DETECTORS

| Tools    | Pre-processing(s) | Training & Testing(s) |
|----------|-------------------|-----------------------|
| TextLSTM | 656               | 21963                 |
| ASTNN    | 2412              | 7419                  |
| TBCCD    | 4991              | 187467                |

In summary, from Table IV we observe that the pure Text-based detector (TextLSTM) can be easily bypassed with a high possibility. The models based on hybrid abstractions of source code do achieve better resilience, especially the TBCCD model which relies on both Token and AST. From the CLONEGEN perspective, the DRLSG strategy has shown the best effectiveness since it performs best in testing TextLSTM and ASTNN. We count the number of equivalent transformation operations used in all transformed code in the DRLSG strategy to escape TextLSTM, ASTNN, and TBCCD, with average values of 51, 51, and 59, respectively. It is clear from the experiments that more code equivalent operations are required to escape TBCCD.

We further explore the changes in the number of tokens before and after the code change of our proposed DRLSG strategy through experiments. When a trained DRLSG model generates clone code, it repeatedly selects a code conversion operator to convert the code until the maximum reward is obtained or the maximum number of iterations is reached. Approximately, 62%, 71%, and 32% of the clone pairs generated by the DRLSG strategy are not identified by the TextLSTM, ASTNN, and TBCCD tools. Further analysis reveals that the number of tokens in the undetectable clone pairs by these tools increases by an average of 157, 156, and 175 tokens for these tools, while the number of tokens in the detectable clone pairs increases by an average of 146, 147 and 146. It has been found that the increased number of tokens after code transformation is indeed one of the factors that make the generated clone code harder to detect.

Now, we answer the RQ1: the code pairs generated by CLONEGEN can effectively bypass the detection of the state-of-the-art ML-based detectors, sharply dropping $F_1$ of these detectors from 90%+ to as low as 50%–80% in many cases. ASTNN, being most efficient, is the most fragile one with a low $F_1$ of 0.577, as clone detection is just one of the applications ASTNN supports. TBCCD has the best robustness against CLONEGEN, but it relies on very costly pre-processing and training. Last, TextLSTM seems to strike a balance in efficiency and robustness.

C. RQ2: Adversarial Training With CLONEGEN

In this section, we try to answer the RQ2 with the experimental results. That is, whether or not augmenting the training of ML-based detectors with adversarial samples can defend
CLONEGEN. The answer to RQ1 reveals the robustness issues of the ML-based detectors. A common idea of reinforcing such detectors is to fuse the spear into the shield, which means we can convert the code clone pairs from CLONEGEN to be adversarial samples to re-train the ML models.

It is believed that training data significantly affects the performance of deep learning models [19]. In general, the more complete the training data is, the better performance the model achieves. Towards this goal, we enhance the original dataset $D_1$ with the generated adversarial samples by four different strategies. Consequently, we use the following datasets as the training dataset: $D_1 + 25\% D_{RS} \cap 25\% D_{GA} \cap 25\% D_{MCMC} \cap 25\% D_{DRL}$. Here, $D'_{RS}$, $D'_{GA}$, $D'_{MCMC}$ and $D'_{DRL}$ are the newly generated datasets by applying the four different strategies on $D_1$ again ($|D_1| = |D'_{RS}| = |D'_{GA}| = |D'_{MCMC}| = |D'_{DRL}|$).

Then, we use the following datasets as the testing dataset ($D_{RS}, D_{GA}, D_{MCMC}, D_{DRL}$), which are the same as those used in §VI-B. Due to the randomness of these strategies, the newly generated datasets (e.g., $D'_{RS}$) will not be identical to the testing datasets (e.g., $D_{RS}$).

We ensure that there is no overlap between our training and test sets, as we do not change the test dataset during our experiments, we only perform equivalent transformations on the data contained in the training set in the ML-based clone detectors. In addition, the data generated by our test different strategies are constructed as $(x, x')$ where $x'$ is the code after the $x$ transformation, there are no data pairs of the type $(x, x')$ in our training dataset, so we can guarantee the fairness of the test. In addition, we calculate the code similarity according to the code editing distance (see Section V-B2) between different runs of the DRLSG strategy, and the average text similarity is 30.6% among code pairs from two runs. This ensures there are no overlapped (or very similar) clone samples between the training and testing datasets. We re-train the models of the ML-based detectors without altering any parameters.

Fig. 6 shows the results of five experiments after adversarial training. It can be seen from the experimental results that the F-measures of these ML-based clone detectors have been significantly improved and become more stable (lower standard deviation, see Table VII) after adversarial training, and the code transformations have less impact on it. We re-generate the $P, R, F_1$ values of each detector against the transformation strategies in Table VII. For example, the row $D_1$ in Table VII represents the accuracy of the retained models on the test dataset $D_1$. Compare back with the row $D_1$ in Table V, for TextLSTM, the adversarial learning improves $P, R, F_1$ from 0.991 to 0.995. For ASTNN, adversarial training slightly improves its accuracy, increasing $F_1$ from 0.977 to 0.979. For TBCCD, the new training results in a slight decrease in $P$ and $F_1$ to 0.981 and 0.983, $R$ slight increase to 0.984. The results are reasonable, as the adversarial samples are quite different from the original samples. Besides, the accuracy in Table V is very high, we cannot expect much improvement via the adversarial training.

Regarding the effectiveness of different strategies in escaping from the detection, the dataset of clone pairs generated by the DRLSG strategy (i.e., $D_{DRL}$) achieves the worst detection accuracy among the last four rows in Table VII — this result clearly shows clone pairs generated by the DRLSG strategy are least detectable, and at least Recall (8.1%) of its generated clone pairs cannot be detected by ASTNN. On the other side, the RS strategy (i.e., $D_{RS}$) seems to be the least effective among the four strategies, generating the most detectable clone pairs and resulting in high detection accuracy. Last, the strategies GA (i.e., $D_{GA}$) and MCMC (i.e., $D_{MCMC}$) show similar effectiveness, achieving accuracy somewhere in between $D_{DRL}$ and $D_{RS}$.

Regarding the robustness of different ML-based detectors in capturing various generated clones, we find that the accuracy of these detectors is generally satisfactory after adversarial training. As shown in Fig. 7, in general, adversarial training improves the robustness of these ML-based detectors by increasing their $P, R, F_1$ values. To be specific, the $F_1$ value of TextLSTM has improved by 0.194 on average among all the strategies. In the best case, it increases the $F_1$ by 0.317 on the $D_{DRL}$ dataset. For ASTNN, the average increase of $F_1$ is 0.248, and it achieves a maximum increase of 0.342 on the $D_{DRL}$ dataset. The last detector, TBCCD, achieves an increase of only 0.119 on average, maximum increase of 0.163 on the $D_{GA}$ dataset.

To further test the effectiveness of adversarial training with CLONEGEN, we test our approach (CLONEGEN) on a new dataset

Fig. 6. F-meaures of different experiments after adversarial training.
of C/C++ programs recently open-sourced by IBM Research and MIT-IBM Watson AI Lab, CodeNet Dataset [60], which is recently widely-used for various code analysis tasks. CodeNet Dataset has two C/C++ benchmark datasets, similar to the popular POJ-104, we construct 62,620 data pairs for testing. As the existing tools TBCCD and ASTNN support C, but not C++, we choose TextLSTM, which supports both C/C++, for validation. We train the TextLSTM model on the POJ-104 dataset and test it on CodeNet Dataset. As shown in Table VIII, The results show that the F1 accuracy of TextLSTM can be improved by 8.8% (74.6%-65.8%) after adversarial training. It can be seen that TextLSTM is trained on the POJ-104 dataset and achieves good results (0.991), but it is not so good on another CodeNet dataset without training (0.658), indicating poor generalization of existing tools. This is another topic related to deep learning work that we will explore in further detail in the future. It is experimentally verified that adversarial training can improve the accuracy of ML-based clone detectors, adversarial training can improve the generalization of the model, and the robustness of the model can be improved by adding transformed data. Since the available C/C++ clone datasets used in this paper are all Type IV clones, we do not discuss the specific clone types separately.

Now, we answer the RQ2: after the adversarial training with samples from CLONEGEN, the F1 values for the ML-based clone detectors significantly increase, indicating that the adversarial training has enhanced the robustness of the ML-based detectors. Meanwhile, the DRLSG strategy has exhibited the best effectiveness in generating undetectable clones as it makes TextLSTM, ASTNN, and TBCCD achieve the lowest accuracy among the four strategies.

D. RQ3: Effectiveness of Transformation Operators

In this section, we answer the RQ3 about the effectiveness of these atomic transformation operators. Different from the previous experiments using all operators, we apply and assess these operators group by group. In particular, we divide our operations into two groups [31]: (1) those highly-relevant to semantic clones (Op1-ChRename, Op2-ChFor, Op3-ChWhile, Op4-ChDo, Op5-ChIfElseIf, Op6-ChIf, Op7-ChSwitch, Op8-ChRelation, Op9-ChUnary, Op10-ChIncrement, Op12-ChDefine, and Op14-ChExchange), and (2) those of simple obfuscations (Op13-ChAddJunk, Op11-Constan, Op15-ChDelete).

To explore the impact of these operations on the clone detectors, we generate two new clone datasets: the clone dataset \((D_{Sem})\) that applies only the first group of operators and the dataset \((D_{Obj})\) that applies only the second group. To eliminate possible side-effects of different transformation strategies, we do not use any above heuristic strategies. Inside, our method is straightforward in that it aggressively sets all bits in the bit vectors to be 1 for certain transformation operators, resulting in the complete set of code transformations at all qualified code locations.

The column ‘Original’ in Table IX indicates that the model is trained on the \(D_I\) dataset, and the column ‘Adversarial’ indicates that the model is trained by adding adversarial samples \((D_I+25\%D_{RS}+25\%D_{GA}+25\%D_{MCMC}+25\%D_{DRL})\). The \(D_I\) row in Table IX shows the accuracy on the original dataset, and the \(D_{Sem}, D_{Obj}\) indicate the accuracy on the dataset generated by the two types of transformation operations.

In general, the accuracy of the model is significantly affected by the two groups of transformation operators — no matter whether on column ‘Original’ or ‘Adversarial,’ the \(F_1\) values of the second group \((D_{Sem})\) are always decreased, in contrast with the first row \((D_I)\). Between the two different groups of transformation, obfuscation-like transformations (e.g., \(D_{Obj}\)) are better at bypassing the detection of TextLSTM, ASTNN, TBCCD, with average \(F_1\) values of 0.742 and 0.945 in ‘Original’ training and ‘Adversarial’ training, respectively. Besides, transformations for semantic clones \(D_{Sem}\) are also valid in escaping from the ML-based detection, with the average \(F_1\) values of 0.877 and 0.962. Hence, the experiments prove

| Table VII |
|---|
| RESULTS OF THE ML-BASED DETECTORS WITH ADVERSARIAL TRAINING IN DETECTING CLONE PAIRS GENERATED BY CLONEGEN |
| \(D_I\) | \(D_{RS}\) | \(D_{GA}\) | \(D_{MCMC}\) | \(D_{DRL}\) |
| TestData | TextLSTM | ASTNN | TBCCD | TextLSTM | ASTNN | TBCCD |
| P | R | F1 | P | R | F1 | P | R | F1 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \(D_I\) | 0.995 | 0.995 | 0.995 ± 0.000 | 0.994 | 0.965 | 0.979 ± 0.000 | 0.981 | 0.984 | 0.983 ± 0.000 |
| \(D_{RS}\) | 0.989 | 0.989 | 0.989 ± 0.000 | 0.937 | 0.958 | 0.948 ± 0.014 | 0.960 | 0.986 | 0.973 ± 0.006 |
| \(D_{GA}\) | 0.971 | 0.970 | 0.970 ± 0.009 | 0.911 | 0.829 | 0.920 ± 0.020 | 0.960 | 0.994 | 0.977 ± 0.005 |
| \(D_{MCMC}\) | 0.989 | 0.989 | 0.989 ± 0.000 | 0.915 | 0.941 | 0.928 ± 0.017 | 0.946 | 0.988 | 0.967 ± 0.011 |
| \(D_{DRL}\) | 0.938 | 0.967 | 0.952 ± 0.006 | 0.904 | 0.934 | 0.919 ± 0.029 | 0.951 | 0.984 | 0.967 ± 0.011 |

![Fig. 7. F-measure before and after adversarial training.](image-url)
that the obfuscation transformation operators are in general more effective in affecting the accuracy of the ML-based clone detectors.

We count the distribution of different operators in all transformation sequences as shown in Fig. 8. Results show that Op11 (Modifying constants) has the largest ratio and Op13 the lowest, and the rest operators have similar ratios, ranging from 2.5% to 11%. Constant values contain rich semantic features, and modifying constants (Op11) affects both token and AST features of code snippets, and the probability of constants appearing in code fragments is higher, so the DRLSG model uses Op11 more frequently. To control the complexity of the code, the number of operations using Op13 in the DRLSG model and the model reward are negatively correlated (see Section V-B2), and thus Op13 is used less frequently.

Now, we answer the RQ3: from the experimental results, both groups of transformation operators are effective in generating clone pairs that can escape from the ML-based clone detectors. For these two groups, obfuscation-like transformation operators are more effective and lower more detection accuracy of TEXTLSTM and ASTNN. The Op11 is most prevalent in clones not detected by the clone detectors.

E. RQ4: Traditional Detectors Versus CLONEGEN

In this section, we answer the RQ4, which explores whether or not traditional clone detectors can defend CLONEGEN. While ML-based detectors have achieved desirable performance, traditional detectors are still widely used in practice, and testing traditional clone detection tools ensures the integrity of our experiments. According to the previous study [61], traditional clone detector DECKARD achieves a poor recall (0.05) on the original POJ-104 dataset and SOURCERERCC obtains a reasonable recall (0.74) on the original POJ-104 dataset.

To explore to what extent the cloned code from CLONEGEN may escape from traditional detectors, we conduct a set of experiments with four open-source detectors as SOURCERERCC [59], DECKARD [11], CCAALIGNER [10], NiCAD [62], all tools use the default settings. We have designed CLONEGEN to support transformations to C++, but some of the traditional tools only support C. We have looked carefully at the POJ-104 dataset in our experiments and found that the data in it is all C (no headers), so we can guarantee that all traditional clone detection tools will be able to parse it properly. CLONEGEN transforms code on a file-by-file basis. When testing traditional clone detectors we test on a file-by-file basis, constructing pairs of data as $\langle x, x' \rangle$, where $x'$ is the code after the transformation. Although the existing traditional clone detectors use different levels of granularity for detection, they all parse the data correctly, and as long as the traditional clone detectors report the presence of a cloned fragment in the file pair, we assume that the tool correctly identifies the $\langle x, x' \rangle$ clone pair when counting the results.

Table XI shows the experimental results under the four different strategies. Table X lists only the detection time of the four detectors, as traditional clone detectors do not require a pre-processing or training stage and could perform clone detection directly. Notably, column ‘Processing Number’ refers to the number of threads we run each detector in parallel to speed up the detection. In practice, we run 104 threads (the same as the number of folders in POJ-104). If not using parallel detection,
the traditional detectors would take up to days or even one week to finish the detection.

Traditional detectors have no embedded ML models. Instead, they detect upon the recognition of suspicious clone patterns within the source code representation like token [10], [59], text [62], tree [11], dependency graph [12] and so on. We treat them as black boxes, which are fed with the set of generated code clone pairs from CLONEGEN and output the "clones or not" decision. Finally, we calculate the overall Recall values for each detector. In fact, evaluating the precision of the tool is challenging because it requires manually marking clone pairs as clones. Regarding the unreported precision results, we want to clarify that the traditional clone detectors usually have good detection precision but unsatisfactory recall according to the study [63]. In the previous study [64], only recalls of the traditional clone detectors are evaluated and discussed. Hence, in this study, we also focus on the evaluation of recalls of traditional clone detectors.

Overall, CLONEGEN makes the majority of the traditional detectors expose low recall values, with an average value of 0.049 for \( D_{DRL} \). Besides, the DRLSG strategy is more effective than other strategies in beating the traditional detectors, since more code changes bring more code syntactic differences. This means that, under the DRLSG strategy, CLONEGEN can quickly generate many code clone pairs that can bypass these detectors with more than a 90% success rate. We examine the detectors whose recall values are below 0.10. SOURCERERCC is a token-based detector that has the lowest recall (0.001) and can be easily defeated. NiCAD uses flexible, pretty-printing, and code normalization to accurately detect intentional clones. It has the second-lowest recall (0.026). CCAligner uses a combination of sliding windows and hashes to detect large-gap type clones. Its recall is 0.07. Deckard is a tree-based clone detection tool with a 0.099 recall value.

Now, we answer the RQ4: the traditional clone detectors have no resistance to the clone pairs generated by CLONEGEN, especially to the clone pairs in \( D_{DRL} \). For the tree-based detector (i.e., Deckard) that is believed to be more robust than token-based ones, its detection recall is also less than 10% on the datasets CLONEGEN generates.

VII. DISCUSSION

A. Threats to Validity

Threats to internal validity come from the parameter setup for the traditional and ML-based detectors used in this paper. To address this issue, for the traditional detectors, we use their default settings. Since traditional detectors usually have few parameters to tune, their results are quite stable. For the ML-based detectors, we use the same parameters as those reported in their papers [16], [17]. We also fine-tune the unreported parameters and finally make the trained models reproduce similar results as those reported in the corresponding papers. Through this rigorous process, we believe that we have conducted fair comparisons between our approach and all the baseline clone detectors in this study.

Threats to external validity mainly come from two aspects. First, we just use the POJ-104C language dataset to evaluate the detectors, which is a widely-used open-source benchmark for large-scale semantic clone detection. Currently, we have only done evaluations on C/C++ programs, however, like other high-level languages (e.g., Java) that share similar language features (e.g., object-orientation), we believe the evaluation findings could also be generalized to clone detection for other high-level languages. Second, we evaluate only three ML-based detectors in this paper due to the tool availability issue. In the future, we would like to support more ML-based detectors (e.g., [54], [55] when they are publicly available) or even code plagiarism tools such as MOSS [65].

The threat in our approach is that we have only considered 15 code-equivalent transformations so far, and there are many other implementable equivalence transformations, such as expanding loops, etc. Due to the extensibility of our approach, researchers can extend our approach with more implementable equivalence transformation operators. The edit distance is not the best algorithm to measure similarity. Still, edit distance is used as the basic similarity measure by many traditional clone detectors like CCFinderX and Deckard, etc. In future, we will enhance CLONEGEN with more similarity measures discussed in the study [23], which are suitable for different code representations (e.g., AST, CFG, metrics), to guide the process of reinforcement learning more effectively.

B. Impact of Transformation Operators

As shown in Table XII, we have analyzed the impact of 15 proposed transformation operators on the four commonly-used representations of source code. The impact could be categorized into three levels. Firstly, severe impact (denoted by \( \ast \)) means that the operator breaks the original structure of the representation,
while minor impact (denoted by \( \) ) means that it only changes the node properties of the representation (e.g., changing one node in AST or CFG). Last, no impact (denoted by \( \) ) means that neither the properties nor the structure of the representation is modified.

Token-based detectors are susceptible to all the changes bought by the transformation operators, except Op1 (identifier renaming) and Op14 (statement exchanging). Hence, token-based detectors can only detect Type I and Type II clones, and our operators can easily evade the token-based detectors (e.g., SOURCECC [59], CCALIGNER [10], CCFINDERX [66]). AST-based detectors (e.g., DECKARD [11]) perform better than the token-based ones, but the AST structure is not resilient to control- or data-flow changes made by operators such as Op2 to Op14. Therefore, AST-based detectors are good at detecting Type I and II clones and these detectors could be evaded by our operators. In contrast, CFG- or PDG-based detectors (e.g., CCGRAPH [12]) are more resilient to the control- or data-flow changes. Especially for CFG-based detectors, they are more resilient to the transformation performed by Op8 to Op11 (e.g., changing while to for-loop). However, the Op4, Op7, Op12, and Op13 would have a severe impact on the four representations, as the transformations brought by these operators belong to Type-IV clones (i.e., semantic clones).

Tufano et al. [67] have implemented a clone detector that evaluates clones based on identifiers, ASTs, CFGs, and bytecode representations, but it does not support the C/C++ language and we were unable to assess it. We also try to apply FCDETECTOR, a tool combining CFG and AST features, but FCDETECTOR reports timeout for clone detection on POJ-104. Finally, we evaluate the tool TBCCD that combines token and AST information and find that the robustness of TBCCD is better than TEXTLSTM and ASTNN. The rationale is that TEXTLSTM uses only textual features and ASTNN uses only AST features. Hence, we can observe that when features of multiple representations are combined, the robustness of a clone detector could be better against CLONEGEN.

C. Lightweight Code Transformation or Obfuscators?

Compared with existing methods [23], [24], we neither propose new obfuscation techniques nor perform code transformations according to traditional obfuscation methods. Our method generates code via lightweight transformations by combining existing semantic-preserving transformation operators and search strategies to generate Type IV clones.

In this study, we use transformation operators and strategies and implement our framework CLONEGEN to perform the lightweight semantic-preserving transformation. Software obfuscators [32], [38], [68], [69], [70] can also enable equivalence transformations of source code. The reasons why we do not employ existing obfuscators in this work are threefold: 1) defeating clone detection is essentially a trade-off problem that strikes the balance between costs in code transformation and benefits from evasion. Obfuscators are generally not free of charge and comparatively time-consuming. Hence, we offer a lightweight yet effective code transformation approach. 2) Obfuscation is often utilized to protect the IP (Intellectual Property). Therefore, the code after obfuscation often has poor readability and maintainability. For example, for the code int i = 1;, after the encoding by the obfuscator [70], it will become:

```c
int o_8ff9af5e5913588b0b77f5602caf02 =
(0x0000000000000002 + 0x0000000000000201 +
0x0000000000000801 - 0x0000000000000A03);
```

3) More importantly, small rather than complicated transformations are much easier for algorithm developers to analyze and debug robustness issues. Hence, simple yet effective transformations are always favorable to algorithm developers.

VIII. RELATED WORK

A. Code Clone Detection

Generally, existing clone detectors can be classified as textual-based, token-based, structural-based code cloning detection approaches. The textual-based methods [9], [62], [71] represent code fragments in the form of strings. If the text contents of the two code fragments are similar, they are considered clones. The approaches described in [4], [10], [18], [59], [66] represent source code as a series of token sequences and use different similarity detection algorithms on the token sequences to detect code clones. The approaches described in [11], [16], [72], [73] detect code similarity by extracting the syntax of the code to obtain the semantic features of the code. Recently, some approaches [6], [55], [74] perform code clone detection by extracting semantic code features, for example using hybrid features, such as CFGs and ASTs of a program [15], [23], [75], or through learning-based approaches [15], [16], [17], [18], [53], [55], [73], [75], [76], [77], [78]. Regarding the assessment of clone detectors, BigCloneBench [79] provide a clone detection benchmark to evaluate the clone detectors, and existing studies [13], [28], [80], [81], [82], [83] focus on evaluating traditional detectors in certain aspects, and there is still a lack of studies that systematically challenge and assess the robustness of the recent ML-based clone detectors.

B. Code Mutation

Code mutations are often used in code testing. Mutation testing, generates a large number of mutants that are automatically embedded in the code to exercise a target program to detect its bugs [84], [85], [86], [87]. Mutation testing can also be used to test the effectiveness of code clone detectors. Roy et al. [62] identified and standardized potential clones, and then used dynamic clustering to perform simple text-line comparisons of potential clones. Roy et al. [22] proposed a mutation insertion method to test code clones. The idea is to reinert an artificial piece of code into a piece of source code so that different types of code clone pairs can be artificially forged and then tested against the target clone detectors. Their proposed tool is a random transformation of the code, which may change the semantics of the original code. Svajlenko et al. [64] presented a benchmark framework based on mutation analysis, which evaluates the recall rate of clone detection tools for different types of clones and the editing of specific types of clones does not
require human intervention. Unlike approaches [22], [64] that can transform code but cannot guarantee semantic preservation, our approach always generates semantic equivalent clones for robustness validation. Recently, Zhang et al. [58] only used one transformation operator (renaming variable), which is included in this work, to prove that some source code processing methods (e.g., ASTNN and Token-based LSTM models) are not sound in the code classification problem. In this paper, our approach proves that the ML-based clone detectors are not sound enough in detecting code clones after simple yet effective equivalent transformations.

C. Code Obfuscation

Equivalence transformations are also commonly used in code obfuscation. Program obfuscation is a set of semantic-preserving program mutation techniques. It is mainly used to hide the intent of a program or to protect the intellectual property of software before its release. Liu et al. [38] proposed a language-model-based obfuscation framework. It makes code refactoring tools like JSNice [88] more difficult to refactor a program. Breanna et al. [39] presented MOSSAD, a method for making code plagiarism tools by inserting junk code, which effectively defeats theft detectors such as MOSS [65]. Schulze et al. [24] proposed to apply code obfuscations to evaluate the robustness of some traditional clone detectors. They applied a few code obfuscations semi-automatically to the source code and did not consider strategies to guide code mutation. The goal of our work is to conduct a simple yet effective transformation to generate semantic clones to evade both learning-based and traditional clone detectors. Rather than applying a heavy-weight transformation (encoding [70], CFG flatten [32] or other compiler optimizations [40]), our lightweight approach makes it easier for developers to quickly discover and locate robustness issues in a clone detector.

IX. CONCLUSION

This paper presents CLONEGEN, a lightweight yet effective code transformation framework that can assess the robustness of ML-based clone detectors by automatically generating clone pairs. Several state-of-the-art ML-based and traditional clone detectors. The experimental results show that our lightweight transformations are effective in evaluating the robustness of clone detectors and can significantly reduce the performance of three recent ML-based detectors, i.e., ASTNN, TBCCD, TextLSTM. Our study reveals the robustness implications of the machine learning-based clone detectors, which calls for more robust and effective methods for data collection and model training. One possible solution is to design a hybrid source code representation to improve the capability of existing ML-based detectors. In our future work, we will consider adding more constraints during the reinforcement learning process, like those adopted in the existing work [64], to minimize the comparison of original and transformed code fragments. Our source code and experimental data are publicly available at https://github.com/CloneGen/CLONEGEN.

REFERENCES

[1] A. Mockus, “Large-scale code reuse in open source software,” in Proc. IEEE 1st Int. Workshop Emerg. Trends FLOSS Res. Develop., 2007, p. 7.
[2] C. Kapsor and M. W. Godfrey, “Cloning considered harmful” considered harmful: Patterns of cloning in software,” Empirical Softw. Eng., vol. 13, no. 6, pp. 645–692, 2008.
[3] B. Laguè, D. Proulx, J. Mayrand, E. Merlo, and J. P. Hudepohl, “Assessing the benefits of incorporating function clone detection in a development process,” in Proc. IEEE Int. Conf. Softw. Maintenance, Bars, Italy, 1997, pp. 314–321.
[4] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-Miner: Finding copy-paste and related bugs in large-scale software code,” IEEE Trans. Softw. Eng., vol. 32, no. 3, pp. 176–192, Mar. 2006.
[5] A. Aiken, “Moss: A system for detecting software plagiarism,” 2004, [Online]. Available: http://www.cs.berkeley.edu/aiken/moss.html.
[6] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “DECKARD: Scalable and accurate tree-based detection of code clones,” in Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining, Philadelphia, PA, USA, 2006, pp. 872–881.
[7] X. Wang, Y. Han, S. Zhu, and P. Liu, “Behavior based software theft Detection,” in Proc. ACM Conf. Comput. Commun. Secur., Chicago, Illinois, USA, 2009, pp. 280–290.
[8] J. F. Islam, M. Mondal, and C. K. Roy, “Bug replication in code clones: An empirical study,” in Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineeering, K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M. Fokaefs, and M. Zhou Eds., Osaka, Japan: IEEE, 2016, pp. 66–78.
[9] S. Lee, C. Ji, and J. Jeong, “SID: High performance code clone detection system for large scale source code,” in Proc. Companion 20th Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst. Lang. Appl., 2005, pp. 140–141.
[10] P. Wang, J. Svalkenjo, Y. Wu, Y. Xu, and C. K. Roy, “CCAligner: A token based large-gap clone detector,” in Proc. 40th Int. Conf. Softw. Eng., M. Chaudron, I. Cnkovic, M. Chechik, and M. Harman Eds., Gothenburg, Sweden: ACM, 2018, pp. 1066–1077.
[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “DECKARD: Scalable and accurate tree-based detection of code clones,” in Proc. IEEE 29th Int. Conf. Softw. Eng., Minneapolis, MN, USA, 2007, pp. 96–105.
[12] Y. Zou, B. Ban, Y. Xue, and Y. Xu, “CCGraph: A PDG-based code clone detector with approximate graph matching,” in Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng., Melbourne, Australia, 2020, pp. 931–942.
[13] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” Sci. Comput. Program., vol. 74, no. 7, pp. 470–495, 2009.
[14] A. Walker, T. Cerny, and E. Song, “Open-source tools and benchmarks for code-clone detection: Past, present, and future trends,” ACM SIGAPP Appl. Comput. Rev., vol. 19, no. 4, pp. 28–39, 2020.
[15] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, “Function code clone detection with syntax and semantics fusion learning,” in Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal., Sh. Khurishid and C. S. Pasareanu Eds., New York, NY, USA: ACM, 2020, pp. 516–527.
[16] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in Proc. IEEE 41st Int. Conf. Softw. Eng., Montreal, QC, Canada, 2019, pp. 738–743.
[17] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, “Neural detection of semantic code clones via tree-based convolution,” in Proc. 27th Int. Conf. Program Comprehension, Y. Guedéncue, F. Khomh, and F. Barro Eds., Quebec, Canada: IEEE/ACM, 2019, pp. 70–80.
[18] L. Li, H. Feng, W. Zhang, N. Meng, and B. G. Ryder, “CCLearner: A deep learning-based clone detection approach,” in Proc. IEEE Int. Conf. Softw. Maintenance Evol., Shanghai, China, 2017, pp. 249–260.
[19] X. Zhu, C. Vondrick, C. C. Fowlkes, and D. Ramanan, “Do we need more training data?,” Int. J. Comput. Vis., vol. 119, no. 1, pp. 76–92, 2016.
[20] I. Schulman, P. Wolski, E. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017, arXiv: 1707.06347.
[21] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, “Convolutional neural networks over tree structures for programming language processing,” in Proc. 30th AAAI Conf. Artif. Intell., D. Schuurmans and M. P. Wellman Eds., Phoenix, Arizona, USA: AAAI Press, 2016, pp. 1287–1293.
[22] C. K. Roy and J. R. Cordy, “A mutation/injection-based automatic framework for evaluating code clone detection tools,” in Proc. IEEE 2nd Int. Conf. Softw. Testing Verification Validation Workshop, Denver, CO, USA, 2009, pp. 157–166.
[23] C. Raggkhitwetsagul, J. Krinke, and D. Clark, “A comparison of code similarity analysers,” Empir. Softw. Eng., vol. 23, no. 4, pp. 2464–2519, 2018.

[24] S. Schulze and D. Meyer, “On the robustness of clone detection to code obfuscation,” in Proc. IEEE 7th Int. Workshop Softw. Clones, San Francisco, CA, USA, 2013, pp. 62–68.

[25] C. Raggkhitwetsagul and J. Krinke, “Siamese: Scalable and incremental code clone search via multiple code representations,” Empir. Softw. Eng., vol. 24, no. 4, pp. 2236–2284, 2019.

[26] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, “Towards a Big Data curated benchmark of open-source project code clones,” in Proc. IEEE Int. Conf. Softw. Maintenance Evol., 2014, pp. 476–485.

[27] X. Xingjian, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, and W.-C. Woo, “Convolutional LSTM network: A machine learning approach for precipitation nowcasting,” in Proc. Int. Conf. Neural Inf. Process. Syst., 2015, pp. 802–810.

[28] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” IEEE Trans. Softw. Eng., vol. 33, no. 9, pp. 577–591, Sep. 2007.

[29] A. Kovacheva, “Efficient code obfuscation for android,” in Proc. Int. Conf. Adv. Inf. Technol., 2013, pp. 104–119.

[30] K. Fukushima, S. Kiyomoto, T. Tanaka, and K. Sakurai, “Analysis of program obfuscation schemes with variable encoding technique,” IEEE Trans. Comput. Electron., Commun. Comput. Sci., vol. 91, no. 1, pp. 316–329, 2008.

[31] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, “On secure and usable program obfuscation: A survey,” 2017, arXiv:1710.01139.

[32] T. László and Á. Kiss, “Obfuscating C programs via control flow flattening,” Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nomineas, Sectio Computatorica, vol. 30, no. 1, pp. 3–19, 2009.

[33] C. Wang, J. Hill, J. Knight, and J. Davidson, “Software tamper resistance: Obstructing static analysis of programs,” University of Virginia, Charlottesville, VA, USA, 2000, Tech. Rep. CS-2000–12, 2000.

[34] G. Petrović and M. Ivanović, “State of mutation testing at Google,” in Proc. 40th Int. Conf. Softw. Eng. Softw. Eng. Pract., 2018, pp. 163–171.

[35] S. Danicic, “Lava: A system for mutation testing of Java programs,” 2007. [Online]. Available: http://www.doc.gold.ac.uk/mas01sd/mutants/index.html

[36] T. Bäck, Evolutionary Algorithms in Theory and Practice - Evolution Strategies, Evolutionary Programming, Genetic Algorithms. New York, NY, USA: Oxford Univ. Press, 1996.

[37] M. Richay, “The evolution of Markov chain Monte Carlo methods,” Amer. Math. Monthly, vol. 117, no. 5, pp. 383–413, 2010.

[38] H. Liu, C. Sun, Z. Su, Y. Jiang, M. Gu, and J. Sun, “Stochastic optimization of program obfuscation,” in Proc. IEEE 39th Int. Conf. Softw. Eng., Buenos Aires, Argentina, 2017, pp. 221–231.

[39] B. Devore-McDonald and E. D. Berger, “Mossad: Defeating software plagiarism detection,” Proc. ACM Program. Lang., vol. 4, no. OOPSLA, 1–28, 2020.

[40] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, “Unleashing the hidden power of compiler optimization on binary code difference: An empirical study,” in Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation, 2021, pp. 142–157.

[41] A. C. Florea and R. Andonie, “Weighted random search for hyperparameter optimization,” Int. J. Comput., Commun. Control, vol. 14, no. 2, pp. 154–169, 2019.

[42] D. Rocke, “Genetic algorithms + data structures = evolution programs (3rd, Revised and Extended Edition),” J. Amer. Stat. Assoc., vol. 95, no. 449, 2000, Art. no. 347.

[43] G. Navarro, “A guided tour to approximate string matching,” ACM Comput. Surveys, vol. 33, no. 1, pp. 31–88, 2001.

[44] A. Hindle, E. Barr, M. Gabel, Z. Su, and P. Devanbu, “On the naturalness of software,” in Proc. IEEE 34th Int. Conf. Softw. Eng., Zurich, Switzerland, 2012, pp. 837–847.

[45] C. Manning and H. Schütze, Foundations of Statistical Natural Language Processing. Cambridge, MA, USA: MIT Press, 1999.

[46] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, “Convolutional neural networks over tree structures for programming language processing,” in Proc. 30th AAAI Conf. Intell., 2016, pp. 1287–1293.

[47] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, “Generating adversarial examples for holding resistant clone detection models,” in Proc. AAAI Conf. Intell., 2020, pp. 1169–1176.

[48] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, “SourceerCC: Scaling code clone detection to big-code,” in Proc. 38th Int. Conf. Softw. Eng., L. K. Dillon, W. Visser, and L. A. Williams Eds., Austin, TX, USA: ACM, 2016, pp. 1157–1168.

[49] R. Puri et al., “CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks,” 2021, arXiv:2105.12655.

[50] H. Wei and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code,” in Proc. 26th Int. Joint Conf. Intell. Int., Melbourne, Australia, 2017, pp. 3034–3040.

[51] K. Roy and J. R. Corley, “NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” in Proc. IEEE 16th Int. Conf. Program Comprehension, R. L. Krihkaar, R. Lämml, and C. Verhoef Eds., Amsterdam, The Netherlands: IEEE/ACM, 2008, pp. 172–181.

[52] F. Farmahinifarharni, V. Saini, D. Yang, H. Sajnani, and C. V. Lopes, “On precision of code clone detection tools,” in Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reengineering, 2019, pp. 84–94.

[53] J. Svajlenko and C. K. Roy, “The mutation and injection framework: Evaluating clone detection tools with mutation analysis,” IEEE Trans. Softw. Eng., vol. 47, no. 5, pp. 1060–1087, May 2021.

[54] S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: Local algorithms for document fingerprinting,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, San Diego, CA, USA, 2003, pp. 75–86.

[55] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A multilinguistic similarity analyser,” in Proc. 29th Int. Conf. Softw. Eng., Lake Buena Vista, FL, USA, 2018, pp. 141–151.

[56] W. Wang, G. Li, S. Shen, X. Xia, and Z. Jin, “Modular tree network for source code representation learning,” ACM Trans. Softw. Methodol., vol. 29, no. 4, pp. 1–23, 2020.

[57] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “Deep learning visuomotor policies,” in Proc. 30th IEEE/ACM Int. Conf. Program Comprehension, Bethesda, MD, USA, 1998, pp. 368–377.

[58] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” in Proc. Int. Conf. Intell. Softw. Maintenance, Bethesda, MD, USA, 1998, pp. 368–377.

[59] H. Wei and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code,” in Proc. Int. Joint Conf. Intell. Int., Melbourne, VIC, Australia, 2017, pp. 3034–3040.
[74] M. Wang, P. Wang, and Y. Xu, “CCSharp: An efficient three-phase code clone detector using modified PDGs,” in *Proc. 24th Asia-Pacific Softw. Eng. Conf.*, Nanjing, China, 2017, pp. 100–109.

[75] A. Sheneamer and J. Kalth, “Semantic clone detection using machine learning,” in *Proc. IEEE 15th Int. Conf. Mach. Learn. Appl.*, Anaheim, CA, USA, 2016, pp. 1024–1028.

[76] M. White, M. Tutano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in *Proc. IEEE/ACM 31st Int. Conf. Automated Softw. Eng.*, Singapore, 2016, pp. 87–98.

[77] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, “FCCCA: Hybrid code representation for function clone detection using attention networks,” IEEE Trans. Rel., vol. 70, no. 1, pp. 304–318, Mar. 2021.

[78] Y. Wu et al., “SCDetector: Software functional clone detection based on semantic tokens analysis,” in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, Melbourne, Australia, Sep. 21–25, 2020, pp. 821–833. [Online]. Available: https://doi.org/10.1145/3324884.3416562

[79] J. Svajlenko and C. K. Roy, “Evaluating clone detection tools with bigclonebench,” in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2015, pp. 131–140.

[80] E. Burd and J. Bailey, “Evaluating clone detection tools for use during preventative maintenance,” in *Proc. IEEE 2nd Int. Workshop Source Code Anal. Manipulation*, 2002, pp. 36–43.

[81] R. Tiarks, R. Koschke, and R. Falke, “An assessment of type-3 clones as detected by state-of-the-art tools,” in *Proc. IEEE 9th Int. Work. Conf. Source Code Anal. Manipulation*, 2009, pp. 67–76.

[82] F. Van Roy and S. Demeyer, “Evaluating clone detection techniques from a refactoring perspective,” in *Proc. 19th Int. Conf. Automated Softw. Eng.*, 2004, pp. 336–339.

[83] J. Svajlenko and C. K. Roy, “Evaluating modern clone detection tools,” in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 321–330.

[84] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” IEEE Trans. Softw. Eng., vol. 37, no. 5, pp. 649–678, Sep./Oct. 2011.

[85] X. Yao, M. Harman, and Y. Jia, “A study of equivalent and stubborn mutation operators using human analysis of equivalence,” in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2015, pp. 87–98.

[86] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, “Predictive mutation testing,” IEEE Trans. Softw. Eng., vol. 45, no. 9, pp. 898–918, Sep. 2019.

[87] P. McMinn, C. J. Wright, C. J. McCurdy, and G. M. Kapfhammer, “Automatic detection and removal of ineffective mutants for the mutation analysis of relational database schemas,” IEEE Trans. Softw. Eng., vol. 45, no. 5, pp. 427–463, May 2019.

[88] V. Raychev, M. Vechev, and A. Krause, “Predicting program properties from ‘big code’,” *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 111–124, 2015.

Wewei Zhang received the BE degree from the Xidian University in 2015, majored in information security and the MS degree from the University of Science and Technology of China in 2022, majored in information security. His research interests include code clone detection, software analysis, Software reverse-engineering and machine learning.

Hongyu Zhang received the PhD degree from the National University of Singapore in 2003. He is currently a professor with Chongqing University, China. Previously, he was an Associate Professor with the University of Newcastle, Australia, a lead researcher with Microsoft Research Asia, and an Associate Professor with Tsinghua University, China. His research is in the area of software engineering, in particular, software analytics, intelligent software engineering, and software maintenance and reuse. The main theme of his research is to improve software quality and developer productivity by mining software data. He has published more than 180 research papers in reputable international journals and conferences. He received five ACM Distinguished Paper awards and several best paper awards. He has also served as a program committee member/track chair for many international conferences. He is a distinguished member of ACM, a distinguished member of CCF, and a senior member of IEEE. More information about him can be found at: https://sites.google.com/site/hongyuzhang.

Yulei Sui is a scientia Associate Professor with the School of Computer Science and Engineering, University of New South Wales. He is broadly interested in program analysis, secure software engineering and machine learning. In particular, his research focuses on building open-source frameworks for static analysis and verification techniques to improve the reliability and security of modern software systems. His recent interest lies at the intersection of programming languages, natural languages and machine learning. Specifically, his current research projects include secure machine learning, software analysis and verification for bug detection through data mining and deep learning.

Yun Xu received the PhD degree in computer software and theory in 2002. He is a professor with the Department of Computer Science, the University of Science and Technology of China (USTC). In recent years, his research focuses on multi-core parallel theory and methods, parallel application system development, biological literature mining and bioinformatics. He has published more than 50 academic papers in Bioinformatics, BMC Bioinformatics, Journal of Computer Science, Journal of Software and other journals and international academic conferences.

Weei Zhang received the BE degree from the Xidian University in 2015, majored in information security and the MS degree from the University of Science and Technology of China in 2022, majored in information security. His research interests include code clone detection, software analysis, Software reverse-engineering and machine learning.

Hongyu Zhang received the PhD degree from the National University of Singapore in 2003. He is currently a professor with Chongqing University, China. Previously, he was an Associate Professor with The University of Newcastle, Australia, a lead researcher with Microsoft Research Asia, and an Associate Professor with Tsinghua University, China. His research is in the area of software engineering, in particular, software analytics, intelligent software engineering, and software maintenance and reuse. The main theme of his research is to improve software quality and developer productivity by mining software data. He has published more than 180 research papers in reputable international journals and conferences. He received five ACM Distinguished Paper awards and several best paper awards. He has also served as a program committee member/track chair for many international conferences. He is a distinguished member of ACM, a distinguished member of CCF, and a senior member of IEEE. More information about him can be found at: https://sites.google.com/site/hongyuzhang.

Yulei Sui is a scientia Associate Professor with the School of Computer Science and Engineering, University of New South Wales. He is broadly interested in program analysis, secure software engineering and machine learning. In particular, his research focuses on building open-source frameworks for static analysis and verification techniques to improve the reliability and security of modern software systems. His recent interest lies at the intersection of programming languages, natural languages and machine learning. Specifically, his current research projects include secure machine learning, software analysis and verification for bug detection through data mining and deep learning.

Yun Xu received the PhD degree in computer software and theory in 2002. He is a professor with the Department of Computer Science, the University of Science and Technology of China (USTC). In recent years, his research focuses on multi-core parallel theory and methods, parallel application system development, biological literature mining and bioinformatics. He has published more than 50 academic papers in Bioinformatics, BMC Bioinformatics, Journal of Computer Science, Journal of Software and other journals and international academic conferences.

Shengjian Guo received the BS and MS degrees both in software engineering from Wuhan University and the PhD degree from the ECE Dept of Virginia Tech, co-advised by Dr. Chao Wang (USC) and Dr. Michael Hsiao (Virginia Tech). He is an applied scientist at AWS Proactive Security. Before Amazon, he was a security researcher with BlackRidge Technology. His research interests include automated software vulnerability detection, cache timing side-channel analysis, auto-driving security, and testing and verification of concurrent programs.

Yufei Guo received the PhD degree in computer science from the National University of Singapore in 2003. He is currently an associate professor with the University of Science and Technology of China. His research interests include software engineering and machine learning. In particular, his research focuses on building open-source frameworks for static analysis and verification techniques to improve the reliability and security of modern software systems. His recent interest lies at the intersection of programming languages, natural languages and machine learning. Specifically, his current research projects include secure machine learning, software analysis and verification for bug detection through data mining and deep learning.

Yun Xu received the PhD degree in computer software and theory in 2002. He is a professor with the Department of Computer Science, the University of Science and Technology of China (USTC). In recent years, his research focuses on multi-core parallel theory and methods, parallel application system development, biological literature mining and bioinformatics. He has published more than 50 academic papers in Bioinformatics, BMC Bioinformatics, Journal of Computer Science, Journal of Software and other journals and international academic conferences.

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.