A DETAILED INFORMATION

A.1 Introduction of BLEU and CodeBLEU Metrics

A.1.1 BLEU. The BLEU (Bilingual Evaluation Understudy) [51] metric is a widely used method for evaluating the quality of text which has been machine-translated from one language to another. Developed to assess the accuracy of machine translation outputs, BLEU compares the machine-produced translations to one or more human reference translations. It quantifies translation quality by calculating the precision of n-grams (sequences of n words) in the machine-generated text relative to the reference texts, while also incorporating a penalty for overly brief translations. This metric provides a score ranging from 0 to 1, where a score closer to 1 indicates a greater similarity between the machine translation and the human reference, suggesting higher translation quality. BLEU is praised for its simplicity and objectivity, making it a standard benchmark in the field of natural language processing for comparing the performance of different translation systems.

The overall BLEU score is calculated as:

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} \frac{1}{N} \log p_n\right), \tag{12}$$

where BP is the brevity penalty, N is the maximum n-gram length, and p_n is the precision of n-grams. The brevity penalty is used to penalize overly short translations, and the precision of n-grams is calculated as the ratio of the number of n-grams in the machine translation that appear in the reference translations to the total number of n-grams in the machine translation. The BLEU score is the geometric mean of the precision of n-grams, with the brevity penalty applied to the score. The brevity penalty is calculated as:

$$BP = \begin{cases} 1 & \text{if } c > r, \\ \exp\left(1 - \frac{r}{c}\right) & \text{if } c \leq r, \end{cases} \tag{13}$$

For each n-gram level (e.g., unigram, bigram, trigram, etc.), the precision is calculated as:

$$p_n = \frac{\sum_{\text{clip}_n} c_{\text{clip}_n}}{\sum_{\text{count}_n} c_{\text{count}_n}}, \tag{14}$$

where clip_n is the number of n-grams that appear in the machine translation and the reference translations, count_n is the number of n-grams in the machine translation, and c_{clip_n} and c_{count_n} are the corresponding counts. The BLEU score is the weighted geometric mean of the precision of n-grams, with the weights being the inverse of the number of n-grams. The weights are used to balance the contributions of different n-gram levels to the overall score, with higher weights assigned to longer n-grams. This is done to reflect the fact that longer n-grams are more informative and carry more meaning, and thus should be given more importance in the evaluation.

A.1.2 CodeBLEU. CodeBLEU [57] is an evaluation metric specifically designed for assessing the quality of code generated by machine learning models in programming tasks. It extends the principles of the BLEU metric, traditionally used in natural language processing for evaluating machine translations, to the domain of source code generation. CodeBLEU takes into account not only the syntactic accuracy by comparing n-grams between the generated code and the reference code, but also incorporates semantic and structural aspects unique to programming languages. This includes considering code abstract syntax trees (ASTs), data flow, and logical control structures to better capture the functional correctness of the generated code relative to the reference implementations. By integrating these dimensions, CodeBLEU aims to provide a more comprehensive and meaningful assessment of code generation models, reflecting both the stylistic and functional fidelity of the produced source code. This metric has become increasingly important

as the field of code generation and software engineering assisted by artificial intelligence continues to evolve. CodeBLEU can be defined as the weighted combination of four parts:

$$\text{CodeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{\text{weight}} + \gamma \cdot \text{Match}_{\text{ast}} + \delta \cdot \text{Match}_{\text{df}}. \quad (15)$$

where BLEU is calculated by standard BLEU [51], $\text{BLEU}_{\text{weight}}$ is the weighted n-gram match, obtained by comparing the hypothesis code and the reference code toekns with different weights, $\text{Match}_{\text{ast}}$ is the syntactic AST match, exploring the syntactic information of the code, and Match_{df} is the semantic data flow match, capturing the semantic information of the code. The weights α, β, γ and δ are used to balance the contributions of these four parts.

A.2 Details of Datasets and Tasks

In our experiments, we consider the following five widely-used tasks, (1) defect detection, (2) clone detection, (3) authorship attribution, (4) code translation as well as (5) code summarization.

- Defect detection: Given a source code, this task is to identify whether it contains defects that may be used to attack software systems, such as resource leaks, use-after-free vulnerabilities and DoS attack. In our experiments, we use the dataset provided by Zhou *et al.*[3] [87]. It consists of 27, 318 functions collected from two large C-language open-source projects that are popular among developers and diversified in functionality, i.e., `FFmpeg`[4] and `Qemu`[5]. The defect detection task is treated as binary classification. The positive label indicates that the current project has defects while the negative one represents the opposite case.
- Clone detection: Given two code snippets as input, clone detection task aims to check whether they are equivalent in terms of operational semantics. This paper considers the widely used clone detection benchmark `BigCloneBench` [63] in our experiment. It consists over 6, 000, 000 true clone pairs and 260, 000 false clone pairs from 10 different functionalities. In `BigCloneBench`, each code fragment is a Java method. Following the settings of Zhou *et al.* [72], we discard those unlabeled data while we use 901, 028 code fragments for training and the other 415, 416 ones for validation and testing purposes.
- Authorship attribution: The purpose of this task is to identify the author of a given code snippet. In this paper, we use the Python dataset provided by Alsulami *et al.* [3]. It is collected from the `Google Code Jam1` (GCJ)[6], an annual competition held by Google since 2008. This dataset consists of solutions to 10 problems implemented by 70 authors.
- Code translation: This task aims to migrate legacy software from one programming language in a platform to another. The training data for code translation is the code pairs with equivalent functionality in two programming languages. In this paper, we use the dataset provided by Lu *et al.* [41]. It is collected from 4 open-source projects including `Lucene`[7], `POI`[8], `JGit`[9] and `Antlr`[10]. These projects are originally developed in Java and subsequently translated into C#. This dataset consists of 11, 800 pairs of functions or methods, from which 500 pairs have been randomly selected for validation purposes and the other 1, 000 pairs are used for testing.

[3] https://sites.google.com/view/devign
[4] https://github.com/FFmpeg/FFmpeg
[5] https://github.com/qemu/qemu
[6] https://codingcompetitions.withgoogle.com/codejam
[7] https://lucene.apache.org/
[8] https://poi.apache.org/
[9] https://eclipse.dev/jgit/
[10] https://github.com/antlr/

- Code summarization: The objective of code summarization is to generate a natural language comment for a given code snippet. In this paper, we use the `CodeSearchNet` dataset [31] that consists of six programming languages, including Python, Java, JavaScript, PHP, Ruby, and Go. The data are collected from publicly available open-source non-fork GitHub repositories, with each documentation representing in the first paragraph of the code. To enhance its overall quality, we employ a filtering process according to the guidelines outlined in [41]. The statistics about the filtered `CodeSearchNet` dataset is listed in Table 3.

Table 3. Data statistics about the filtered CodeSearchNet dataset.

Language	Training	Validation	Testing
GO	167,288	7,325	8,122
Java	164,923	5,183	10,955
JavaScript	58,025	3,885	3,291
PHP	241,241	12,982	14,014
Python	251,820	13,914	14,918
Ruby	24,927	1,400	1,261

A.3 Details of Evaluation Metrics

To quantitatively evaluate the performance different methods, we consider the following four metrics in our empirical study.

- Attack success rate (ASR): the percentage of the number of successful adversarial attacks examples (denoted as N_{succ}) w.r.t. the total number of examples generated by the corresponding algorithm (denoted as N_{total}):

$$ASR = \frac{N_{\text{succ}}}{N_{\text{total}}} \times 100\%. \tag{16}$$

  More specifically, for classification tasks, adversarial examples that lead to inconsistencies between the victim model's predictions and the original classification results are considered successful. For generation tasks, following the approach in NLP of setting a threshold [17, 60], we consider adversarial examples successful when the BLEU [51] or CodeBLEU [57] scores are below 50% of the original values. The higher the ASR is, the better performance of the algorithm achieves.
- Average adversarial loss (AAL): the average value of adversarial loss.

$$AAL = \frac{1}{N_{\text{succ}}} \sum_{i=1}^{N_{\text{succ}}} f_1(\mathbf{x}'). \tag{17}$$

- Average semantic similarity (ASS): the average value of semantic similarity.

$$ASS = \frac{1}{N_{\text{succ}}} \sum_{i=1}^{N_{\text{succ}}} f_2(\mathbf{x}'). \tag{18}$$

- Average modification rate (AMR): the average ratio of the perturbation tokens w.r.t. the total number of tokens:

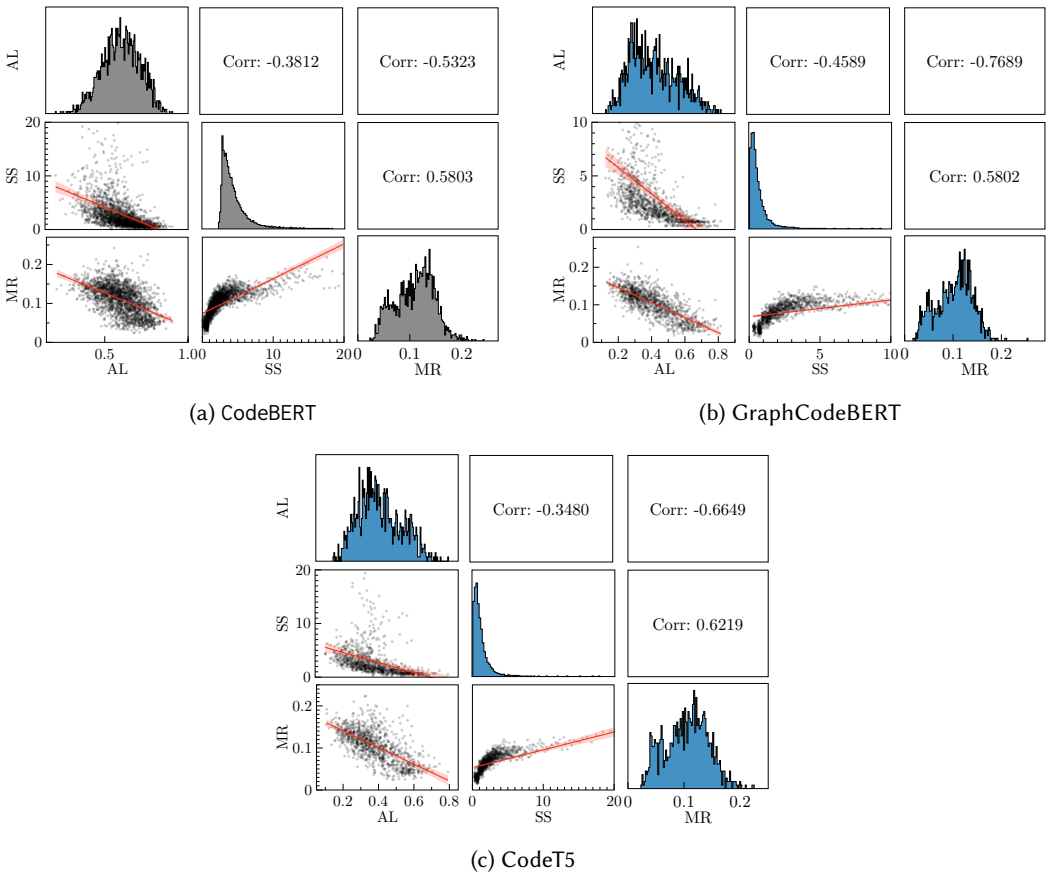
$$AMR = \frac{1}{N_{\text{succ}}} \sum_{i=1}^{N_{\text{succ}}} f_3(\mathbf{x}'). \tag{19}$$

- Average query count (AQC): the average number of queries (denoted as N_{querry}) w.r.t. the victim model to find a successful adversarial example:

$$AQC = \frac{1}{N_{\text{succ}}} \sum_{i=1}^{N_{\text{succ}}} N_{\text{querry}}. \tag{20}$$

  It is important to note that the primary goal of MOAA is to generate a diversity trade-off AEs, in contrast to existing methods which only focus on finding an AE. To ensure a fair comparison with AQC, we evaluate the efficiency of algorithms based on the number of query required to generate the first AE.

Recap from Section 2.2 that we have defined our objectives as to minimize AL, SS and MR. Consequently, we would expect algorithms that yield AAL, ASS, AMR and AQC as low as possible.



(a) CodeBERT

(b) GraphCodeBERT

(c) CodeT5

Fig. 15. Scatter plot of matrices (SPLOM) visualizes the correlation of AL, SS and MR values for CodeBERT, GraphCodeBERT and CodeT5 models on `defect detection` task.

A.4 Introduction of peer algorithms

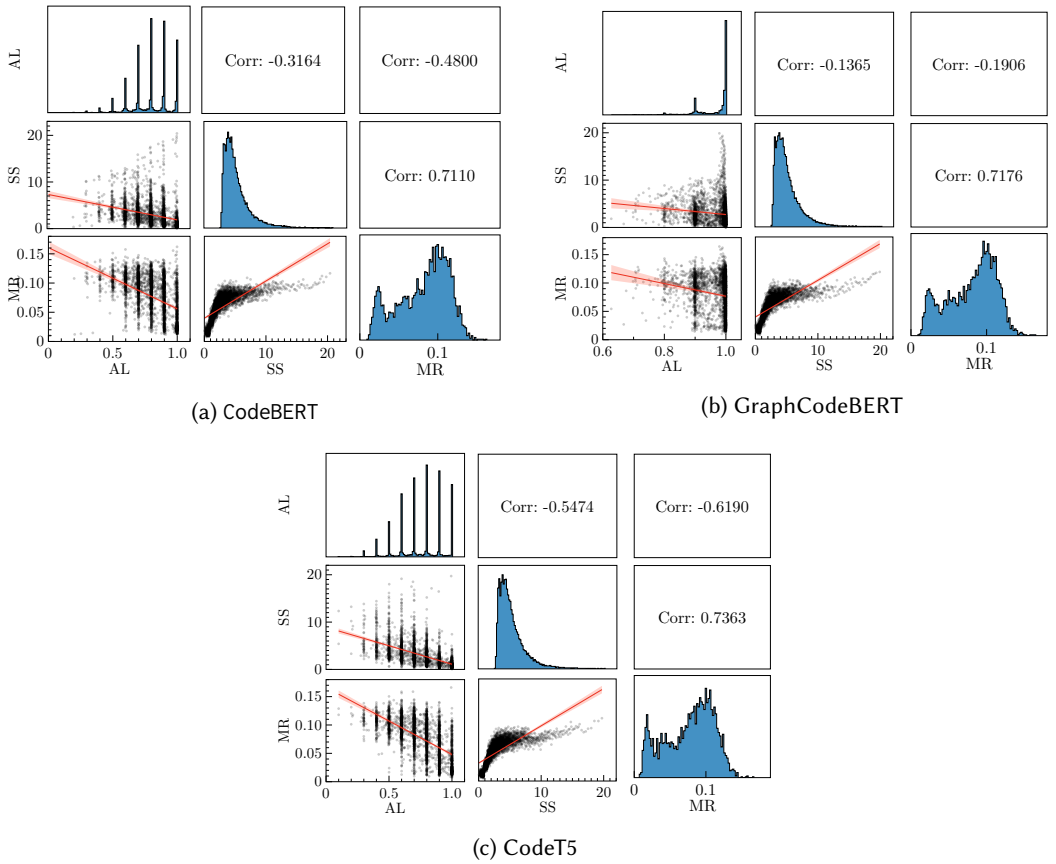
In this section, we introduce the mechanism of each algorithm we adopted as peer algorithms.

- **MHM**: Zhang *et al.* [83] formalized the process of AE generation as a sampling problem. The problem can be decomposed into an iterative process consisiting of three stages: 1) selection of the identifier to be renamed, 2) selection of substitutions and 3) acceptance or rejection decision. To generate AEs for models of code, They proposed Metropolis-Hastings Modifier (MHM), a Metropolis-Hastings sampling-based [25, 46] identifier renaming technique. This method is a black-box attack that randomly selects replacements for local variables and then strategically deciding whether to accept or reject these replacements. This decision is informed by both predicted lables and the corresponding confidence of the victim models, enabling more effective AE generation. MHM employs a pre-defined extensive collection of identifier names, from which the replacements are selected.
- **Greedy Attack**: Yang *et al.* [78] used identifier renaming as the AE generation technique and explored how to produce AEs that are natural. They defined a metric to measure the importance of identifier names in a code snippet and started to substitue identifiers with the highest importance. Greedy Attack greedily selects the replacements (out of all natural substitues), from which the generated AE makes the victtim model produce lower confidence on the ground truth label. If it fails to change the prediction results, Greedy Attack continues to replace the next identifier until all the identifiers are considered or an AE is obtained.
- **ALERT**: Yang *et al.* [78] think that finding appropriate substitutes to generate AEs is essentially a combinatorial optimization problem, whose objective is to find the optimal combination of identifiers and corresponding substitues that minimizes the victim model's confidence on the ground truth label. Thus, they design an attack based on genetic algorithms, called ALERT to solve the problem that Greedy Attack may be stuck in a sub-optimal solution. If the Greedy Attack fails to find a successful adversarial example, they apply ALERT to search more comprehensively. ALERT first represents chromosomes as a list of identifiers pairs which means replacing the identifier by the replacement, and then initialized the population. Subsequently, it performs genetic operators to generate new solution, and keep solutions with larger fitness values in the population. In the end, the algorithm returns the solution with the highest fitness value.

B EXTENDED RESULTS

B.1 Relationship between objective functions

Fig. 15, Fig. 16, and Fig. 17 shows the scatter plot of matrices (SPLOM) visualizes the correlation of AAL, ASS and AMR values for CodeBERT, GraphCodeBERT and CodeT5 models on defect detection, clone detection and authorship attribution tasks, respectively. The correlation between AL and SS is positive, while the correlation between AL and MR is negative. The correlation between SS and MR is negative. The results indicate that the three objective functions are not independent, and the relationship between them is complex. This validates the rationale of our multi-objective optimization formulation.

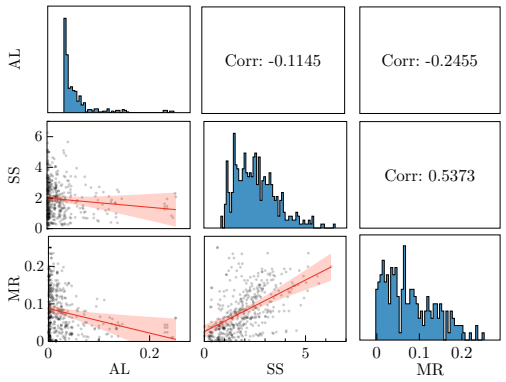


(a) CodeBERT

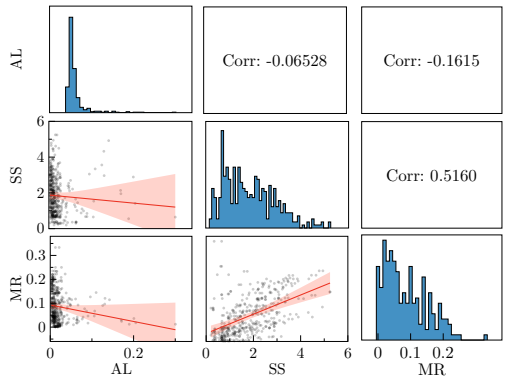
(b) GraphCodeBERT

(c) CodeT5

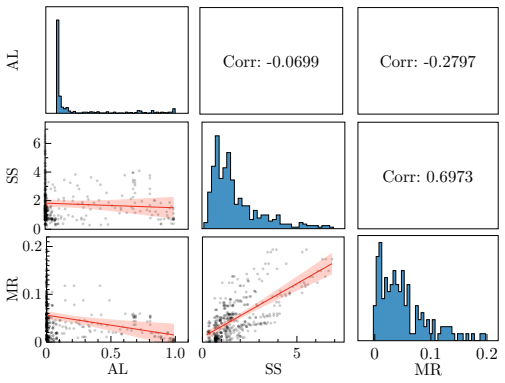
Fig. 16. Scatter plot of matrices (SPLOM) visualizes the correlation of AL, SS and MR values for CodeBERT, GraphCodeBERT and CodeT5 models on clone detection task.



(a) CodeBERT



(b) GraphCodeBERT



(c) CodeT5

Fig. 17. Scatter plot of matrices (SPLOM) visualizes the correlation of AL, SS and MR values for CodeBERT, GraphCodeBERT and CodeT5 models on `authorship attribution` task.

B.2 Comparison Results

The Table 4 shows the comparison results of ASR, AAL, ASS AMR and AQC values obtained by MOAA and the other three selected peer algorithms on `defect detection`, `clone detection`, `authorship attribution`, `code translation` and `code summarization` tasks.

In addition to attack effeiciency, MOAA is also competitive in maintaining the semantic similarity and modification rate of the generated AEs. Note that in Fig. 4, MOAA does not exhibit statistically significant differences in terms of ASS and AMR scores. This is because MOAA generates a set of AEs with diverse trade-offs between the three objectives. When calculating these metrics, we aggregate the values across the entire population, and thus the reported ASS and AMR represent the centric values of the whole population, which can be obscured by extreme AEs that solely focusing on optimizing the adversarial loss (see Fig. 18 for an example).

Fig. 18 visualizes the distribution of generated AEs by MOAA for two examples in `defect detection` dataset in the objective space. The '▲' represents the mean values of the entire population. From the right half of the figure, we can observe the mean value of f_3, i.e. MR, is influenced by the distribution of the population, resulting in an increase. Furthermore, Fig. 19 demonstrates the results for AL, SS and MR when selecting the minimum, median and mean values in the AE population genereated by MOAA. The bar charts shows that the optimal values within the population generate by MOAA significantly surpass the baseline algorithms, indicating the superiority of our algorithm.

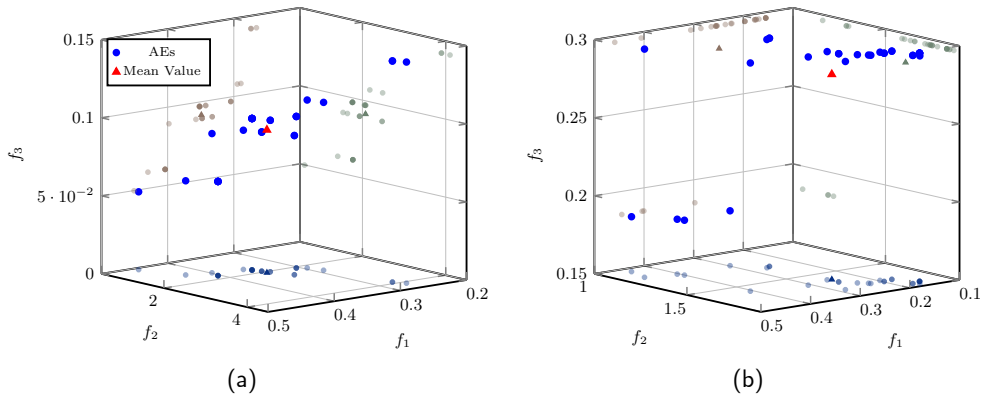


Fig. 18. Visualization of the distribution of generated AEs by MOAA for two examples in `defect detection` dataset in the objective space.

Table 4. Comparison results of ASR, AAL, ASS AMR and AQC values obtained by MOAA and the other three selected peer algorithms on defect detection, clone detection, authorship attribution, code translation and code summarization tasks.

Task	Model	Metric	MHM	Greedy Attack	ALERT	MOAA	Task	Model	Metric	MHM	Greedy Attack	ALERT	MOAA
Defect Detection	CodeBERT	ASR(%)	34.9634(1.1816)†	50.5288(2.2824)†	51.5962(2.0390)	**52.6100(2.0977)**	Code Summarization (Go)	CodeBERT	ASR(%)	73.3126(2.1967)†	80.9632(2.1015)†	86.4264(2.1147)†	**94.3883(2.0976)**
		AAL	0.4048(0.0044)†	0.4008(0.0042)†	0.4158(0.0024)†	**0.3460(0.0056)**			AAL	7.0589(0.6537)†	6.9282(0.4812)†	6.3648(0.4720)†	**0.0914(0.0070)**
		ASS	1.1752(0.0717)‡	**1.0460(0.0447)‡**	1.0904(0.0324)‡	1.7706(0.2364)			ASS	1.9247(0.0164)†	2.0397(0.0748)†	2.1460(0.0317)†	**1.8867(0.0264)**
		AMR(%)	11.0479(0.5489)†	**8.8927(0.3491)‡**	9.7054(0.2470)‡	10.2937(0.3540)			AMR(%)	**13.1255(0.8292)‡**	13.3919(0.7708)	13.3661(0.9694)	13.6364(1.3100)
		AQC	1054.4211(44.2738)†	**228.5147(3.7074)‡**	643.9245(16.9197)	647.5264(15.1203)			AQC	**1208.2164(13.3535)‡**	1329.3666(16.4602)‡	1359.2694(17.0333)‡	1708.4422(14.7381)
	GraphCodeBERT	ASR(%)	52.2113(1.6509)†	66.3950(1.5099)†	70.4752(2.0003)†	**92.6990(1.8308)**		GraphCodeBERT	ASR(%)	60.4947(1.9984)†	66.2294(1.5590)†	74.2207(1.4916)†	**91.2442(1.4846)**
		AAL	0.4371(0.0037)†	0.4260(0.0027)†	0.4241(0.0034)†	**0.2150(0.0032)**			AAL	6.9529(0.0774)†	6.9803(0.0721)†	6.6467(0.0250)†	**0.1689(0.0068)**
		ASS	1.1201(0.0352)‡	**0.9617(0.0305)‡**	1.0175(0.0265)‡	1.2172(0.0287)			ASS	1.3706(0.0031)†	1.3386(0.0073)	**1.3195(0.0098)**	1.3281(0.0024)
		AMR(%)	9.9979(0.3142)†	8.3062(0.3161)†	8.8675(0.2732)†	**7.8421(0.1591)**			AMR(%)	**13.6252(0.9475)‡**	16.1356(1.2271)	15.8896(1.7858)‡	16.3406(1.5793)
		AQC	1545.6196(49.7182)†	213.1684(4.2550)†	600.6114(32.2701)†	**144.5727(6.4235)**			AQC	1260.9188(7.6357)‡	**987.2148(6.6742)‡**	1327.1169(12.5701)‡	1536.6321(12.7392)
	CodeT5	ASR(%)	58.6701(1.1650)†	59.1292(1.8468)†	62.9706(1.9411)†	**75.5396(1.8566)**		CodeT5	ASR(%)	75.4062(2.5589)†	80.6288(2.2577)†	85.1743(2.3457)†	**95.4993(2.2593)**
		AAL	0.4383(0.0034)†	0.4528(0.0038)†	0.4527(0.0033)†	**0.3231(0.0141)**			AAL	7.1635(0.3958)†	6.9374(0.4176)†	6.9158(0.3109)†	**0.0945(0.0027)**
		ASS	**1.0152(0.0628)‡**	1.0376(0.0933)‡	1.0502(0.0724)‡	1.3728(0.0820)			ASS	1.7395(0.0754)	**1.7293(0.3826)**	1.7989(0.3910)†	1.7382(0.0227)
		AMR(%)	**9.1522(1.0717)‡**	9.1843(1.1961)	9.2658(1.1559)	9.2280(0.5606)			AMR(%)	13.6795(0.8142)	**13.4374(1.0781)‡**	13.9038(1.0768)	13.7450(1.2203)
		AQC	1201.2857(32.0083)†	269.4983(4.5513)†	468.4779(4.1553)†	**111.0769(6.6249)**			AQC	1022.4953(10.5874)	**684.7991(4.7953)‡**	1139.4655(9.4255)	909.0473(7.2948)
Clone Detection	CodeBERT	ASR(%)	9.4743(0.3979)†	24.8216(0.7862)†	**28.9126(0.9873)‡**	27.4664(1.2258)	Code Summarization (Java)	CodeBERT	ASR(%)	71.9599(2.1537)†	78.8542(2.1126)†	83.2597(2.1489)†	**91.1603(2.0348)**
		AAL	0.0137(0.0025)‡	**0.0118(0.00277)‡**	0.0140(0.0021)‡	0.0608(0.0104)			AAL	8.5247(1.1589)†	8.1258(0.9196)†	8.4839(0.9260)†	**1.6432(0.0346)**
		ASS	**1.0297(0.0614)‡**	1.4062(0.0527)‡	1.5233(0.0358)‡	3.2655(0.1381)			ASS	1.0395(0.0610)	1.0382(0.0042)	**1.0172(0.0046)**	1.2173(0.0744)
		AMR(%)	**4.4311(0.2943)‡**	7.2608(0.2271)‡	8.3348(0.2251)‡	9.9231(0.2942)			AMR(%)	10.6472(0.0906)†	10.1908(0.0808)	10.1842(0.0763)	**9.7968(0.1862)**
		AQC	**267.7727(0.5215)‡**	343.0625(6.5172)‡	2157.2774(79.6103)†	887.0185(8.3498)			AQC	**479.5237(4.0225)‡**	756.1951(4.0936)‡	889.3219(6.0982)‡	1316.6049(11.8694)
	GraphCodeBERT	ASR(%)	4.8072(0.3733)†	5.9498(0.4234)†	7.7729(0.6346)†	**25.4834(0.7816)**		GraphCodeBERT	ASR(%)	74.1837(1.9567)†	82.7395(2.3089)†	83.3326(2.4118)†	**91.6697(2.2387)**
		AAL	**0.0622(0.0199)**	0.0706(0.0115)	0.0657(0.0088)	0.0837(0.0070)			AAL	8.4296(0.6698)†	8.1521(0.9947)†	8.3189(0.7406)†	**1.7016(0.4572)**
		ASS	2.5864(0.2611)‡	**2.0150(0.09729)‡**	2.1519(0.1083)‡	3.5807(0.0723)			ASS	1.0917(0.6462)	**1.0856(0.8349)**	1.0982(0.6871)	1.1728(0.7549)
		AMR(%)	**4.6996(0.4476)‡**	10.2701(0.4624)	11.4084(0.4162)	10.2541(0.0139)			AMR(%)	10.1801(0.1246)†	9.9435(0.1538)	9.8547(0.1273)	**9.3427(0.1486)**
		AQC	575.7493(6.2874)‡	**357.5935(3.6641)‡**	1127.6145(12.2368)	1464.4348(9.8952)			AQC	1354.8251(0.6596)	**657.1928(7.3942)‡**	743.8721(6.2546)‡	1354.1853(8.4509)
	CodeT5	ASR(%)	13.3097(1.9351)†	15.0468(1.8145)†	**21.4845(1.8092)**	20.6125(1.5927)		CodeT5	ASR(%)	70.8053(2.4352)†	82.5873(2.0992)†	82.5139(1.8864)†	**90.5189(1.6097)**
		AAL	0.0463(0.0034)	**0.0172(0.0042)‡**	0.0177(0.0018)‡	0.03751(0.0032)			AAL	8.4889(0.2460)†	8.3756(0.6413)†	8.3665(0.4718)†	**1.6792(0.3255)**
		ASS	1.4551(0.07417)‡	**0.8734(0.0633)‡**	1.3787(0.0660)‡	3.0523(0.0729)			ASS	**1.1293(0.0051)**	1.1355(0.0042)	1.1298(0.0034)	1.1950(0.0072)
		AMR(%)	6.8009(0.3439)‡	**5.8782(0.4149)‡**	8.5879(0.4739)	9.0597(0.2077)			AMR(%)	10.0251(0.3275)	10.0554(0.4416)	10.0328(0.3695)	**9.8162(0.3426)**
		AQC	**224.6579(7.4395)‡**	609.6146(8.5868)	872.9426(9.2412)†	589.6741(5.8253)			AQC	1105.2413(10.5993)	**783.4504(8.2385)‡**	958.0157(9.2413)	1098.8810(15.0116)
Authorship Attribution	CodeBERT	ASR(%)	33.0841(1.5639)†	33.4579(3.9871)†	41.6822(0.5119)†	**55.6514(1.5784)**	Code Summarization (JavaScript)	CodeBERT	ASR(%)	72.8792(1.4754)†	79.4468(1.8328)†	83.7764(1.8792)†	**94.4724(2.0539)**
		AAL	0.1993(0.0091)†	0.1074(0.0105)	0.2186(0.0072)†	**0.0958(0.0038)**			AAL	7.3475(0.0031)†	7.4288(0.0022)†	7.3414(0.0025)†	**0.0945(0.0017)**
		ASS	2.5588(0.0523)†	1.4009(0.0253)	1.6340(0.0809)	**1.3739(0.0455)**			ASS	1.0251(0.042)	1.0203(0.074)	1.0279(0.062)	**1.0142(0.0351)**
		AMR(%)	13.0903(0.3598)†	7.1008(0.2668)	9.1250(0.3438)†	**5.0087(0.0842)**			AMR(%)	10.0326(0.2601)	9.7083(0.3337)	9.6685(0.2719)	**9.5337(0.1257)**
		AQC	1997.6822(22.7374)†	468.7500(5.8089)	2292.5208(15.7111)†	**379.6458(2.8910)**			AQC	**783.1970(8.0723)‡**	1131.9847(1.8096)	1228.1297(8.7176)	1022.6524(14.4693)
	GraphCodeBERT	ASR(%)	25.9615(1.5203)†	30.5769(1.2537)†	34.0385(1.9939)†	**73.5484(1.6307)**		GraphCodeBERT	ASR(%)	66.0817(2.5152)†	78.3215(0.9378)†	79.1885(1.0357)†	**88.7874(1.1068)**
		AAL	**0.0152(0.0007)‡**	0.0579(0.0030)	0.0605(0.0043)	0.0464(0.0024)			AAL	7.3072(0.0064)†	7.2462(0.0117)†	7.2539(0.0105)†	**0.1814(0.0023)**
		ASS	1.9383(0.2116)†	2.0739(0.0851)†	2.2065(0.0775)†	**1.2839(0.0442)**			ASS	1.4139(0.0893)	1.4130(0.0759)	**1.4023(0.0647)**	1.4078(0.0267)
		AMR(%)	10.1498(1.0148)†	10.1945(0.5468)†	10.6832(0.2184)†	**4.5019(0.0798)**			AMR(%)	10.1872(0.0324)‡	10.1652(0.0274)‡	**10.0051(0.0256)‡**	11.4227(0.0423)
		AQC	1011.0235(12.4738)†	669.5794(5.2947)	720.5902(5.7387)	**502.8567(4.2580)**			AQC	970.4263(5.1902)†	751.3329(3.7841)	880.2802(7.3768)	**701.4414(8.7519)**
	CodeT5	ASR(%)	60.0430(0.8487)†	77.1429(0.4318)†	77.3333(0.4259)†	**93.4615(1.0533)**		CodeT5	ASR(%)	72.4293(1.7211)†	79.3892(2.1395)†	81.4914(2.0268)†	**93.0783(1.8478)**
		AAL	0.3542(0.0023)†	0.3252(0.0022)†	0.3250(0.0028)†	**0.0456(0.0038)**			AAL	7.3166(0.0114)†	7.1265(0.0250)†	7.1176(0.0276)†	**0.09416(0.0041)**
		ASS	1.5294(0.0072)	1.6612(0.0056)	1.6643(0.0068)	**1.0619(0.0467)**			ASS	1.0376(0.0589)	**1.0211(0.0674)**	1.0236(0.0762)	1.0709(0.0341)
		AMR(%)	7.7451(0.0569)†	8.5215(0.0492)†	8.5428(0.0476)†	**4.0707(0.1280)**			AMR(%)	**9.7834(0.0219)‡**	9.8112(0.0758)‡	9.7931(0.1876)‡	10.0085(0.0953)
		AQC	948.2614(8.7978)†	**360.0697(1.4218)**	382.6635(1.7551)	410.7524(3.6198)			AQC	818.6516(4.0134)†	**178.4650(1.1522)‡**	289.5082(1.0374)	244.0604(0.9513)
Code Translation (Java→C#)	CodeBERT	ASR(%)	19.4652(1.3840)†	24.3496(1.3817)†	27.6091(1.8849)†	**37.2587(1.5966)**	Code Summarization (Python)	CodeBERT	ASR(%)	26.5319(1.6192)†	41.9822(1.8452)†	42.3215(1.7456)†	**50.3525(1.4172)**
		AAL	42.7687(1.7746)†	41.9823(2.1617)†	41.9604(2.6058)†	**31.8919(2.0897)**			AAL	9.4605(0.0583)†	9.4158(0.1463)†	9.4669(0.1768)†	**4.6512(0.0330)**
		ASS	**1.4002(0.7874)‡**	1.7456(0.7682)‡	1.7513(0.2381)‡	2.4386(0.6145)			ASS	1.9928(0.0645)†	2.0991(0.1163)†	2.0209(0.1306)†	**1.7004(0.0162)**
		AMR(%)	9.3106(0.2972)‡	**9.1800(0.3815)‡**	9.1883(0.5519)‡	12.6639(0.7013)			AMR(%)	18.4112(0.4339)	18.0986(0.2963)	18.7691(0.1200)	**17.4396(0.2149)**
		AQC	1433.7221(9.7696)†	**684.8405(2.6475)‡**	898.7626(6.1487)	908.4528(5.0057)			AQC	852.0959(4.9375)	**711.8820(3.9474)**	853.9002(8.4915)	770.0263(8.0428)
	GraphCodeBERT	ASR(%)	19.5062(1.6495)†	24.8468(1.9070)†	28.0031(1.8538)†	**32.6510(1.1514)**		GraphCodeBERT	ASR(%)	30.1248(1.7048)†	41.6237(1.2823)†	45.0556(1.2634)†	**61.5703(1.7124)**
		AAL	41.3063(2.5773)†	41.8398(2.1493)†	41.8472(2.0611)†	**32.7849(1.4689)**			AAL	7.1494(0.6472)†	7.2328(0.6517)†	7.2936(0.6423)†	**3.6214(0.7748)**
		ASS	**1.6987(0.2562)‡**	1.8506(0.5873)	1.8874(0.5308)	2.4532(0.6886)			ASS	**1.2519(0.1254)**	1.2833(0.0447)	1.2649(0.0896)	1.3251(0.0691)
		AMR(%)	**12.2491(1.0458)**	12.3073(1.0978)	12.3223(1.4532)	12.5108(1.4730)			AMR(%)	**18.6924(0.4961)‡**	18.7747(0.9233)‡	18.9937(0.7181)‡	20.0072(0.5571)
		AQC	**964.2559(9.1950)‡**	1289.8176(15.5111)	1364.2602(8.2487)	1414.5173(13.7891)			AQC	801.6217(9.9239)†	**609.8933(8.3682)**	969.5867(3.6068)†	629.7227(6.5213)
	CodeT5	ASR(%)	20.5098(1.0747)†	26.8406(1.0908)†	28.5673(1.3533)†	**36.7087(1.5843)**		CodeT5	ASR(%)	36.9622(2.1724)†	49.8849(2.0637)†	57.3779(2.3206)†	**65.1018(2.1651)**
		AAL	41.3249(1.2622)†	41.4436(1.4165)†	41.4580(1.0861)†	**28.9674(1.0467)**			AAL	7.3179(0.5612)†	7.3456(0.4658)†	7.3067(0.7206)†	**3.1588(0.8673)**
		ASS	1.6575(0.0)‡	**1.6336(0.7332)‡**	1.7240(0.1845)‡	2.4986(0.0858)			ASS	**1.3773(0.0699)‡**	1.5138(0.0572)	1.5071(0.0660)	1.6217(0.0877)
		AMR(%)	12.3851(0.0874)	**12.1058(0.0719)**	12.1407(0.0942)	13.4673(0.1248)			AMR(%)	**17.0762(0.3304)‡**	17.2729(0.4249)‡	17.4266(0.2022)‡	19.6417(0.4120)
		AQC	1115.8959(8.5798)†	**364.8792(3.1183)‡**	785.9483(8.3591)	707.5491(8.5932)			AQC	1141.6404(15.1902)†	792.7308(9.8947)†	1077.4974(11.4656)†	**543.6317(4.2029)**
Code Translation (C#→Java)	CodeBERT	ASR(%)	28.3817(1.0964)†	33.0786(1.3050)†	33.6216(1.9560)†	**42.0938(1.1529)**	Code Summarization (Ruby)	CodeBERT	ASR(%)	60.3187(1.1769)†	67.2979(1.4735)†	72.9829(1.7450)†	**89.3568(1.9701)**
		AAL	37.5315(1.0342)†	36.8309(1.2786)†	38.9662(1.4164)†	**27.6437(1.2853)**			AAL	5.8629(0.0143)†	5.4959(0.0079)†	5.8479(0.0118)†	**0.3032(0.0275)**
		ASS	1.8972(0.45552)	1.9275(0.1957)	1.9706(0.2002)	**1.8633(0.2876)**			ASS	**1.0693(0.2175)**	1.0756(0.2583)	1.0763(0.2206)	1.1286(0.0753)
		AMR(%)	**8.1517(0.9473)‡**	8.6149(1.3091)‡	8.7062(1.5883)‡	13.6125(1.2368)			AMR(%)	14.6887(0.0944)	15.6528(0.2067)†	15.7683(0.2237)†	**13.4196(0.1064)**
		AQC	1118.3492(11.5706)†	**235.6494(4.0349)**	326.1112(4.5417)	250.3528(3.9131)			AQC	998.2624(14.2460)	**763.0913(6.6805)‡**	905.4322(9.8221)	960.6993(7.6981)
	GraphCodeBERT	ASR(%)	29.6756(0.9634)†	34.8186(1.2498)†	35.9256(1.0662)†	**43.9068(1.1027)**		GraphCodeBERT	ASR(%)	65.7519(2.0489)†	72.7107(2.3153)†	76.5041(2.0043)†	**91.5836(2.3359)**
		AAL	39.2513(0.3761)†	35.9514(0.7908)†	36.0817(0.1020)†	**26.6884(0.8343)**			AAL	5.5424(0.5721)†	5.6221(0.6601)†	5.9094(0.6487)†	**0.5714(0.0079)**
		ASS	**1.5202(0.2586)‡**	1.7402(0.7869)	1.7519(0.5615)‡	2.3617(0.3879)‡			ASS	1.2667(0.0947)	1.2516(0.0881)	1.2694(0.0754)	**1.2657(0.0302)**
		AMR(%)	**7.9475(0.2641)‡**	8.1708(0.8651)‡	8.6395(0.4379)‡	13.2705(0.6994)			AMR(%)	15.2882(0.6079)	15.5355(0.4816)	15.6184(0.6568)	**14.3566(0.9554)**
		AQC	1434.9513(16.9056)†	**521.4703(5.9889)**	665.5925(8.1927)	532.9388(6.9820)			AQC	515.1050(12.8312)	**302.4738(9.9101)‡**	322.4893(10.1972)‡	683.7195(6.9023)
	CodeT5	ASR(%)	32.1931(0.9267)†	36.4419(0.9254)†	38.2741(1.1782)	**46.3932(0.9016)**		CodeT5	ASR(%)	69.2103(2.3417)†	78.0914(2.0706)†	80.9588(2.4502)†	**94.4562(2.3896)**
		AAL	38.6196(1.0257)†	36.4782(1.2261)†	36.4940(1.2745)†	**26.7596(1.8524)**			AAL	5.9790(0.0048)†	5.9635(0.0085)†	5.9655(0.0057)†	**0.5203(0.0083)**
		ASS	1.8047(0.3979)	1.9286(0.3356)	1.9347(0.7024)	**1.7264(0.6530)**			ASS	1.2781(0.057)	1.2947(0.073)	**1.2676(0.0896)**	1.3233(0.0992)
		AMR(%)	**8.6995(0.3583)‡**	9.3126(0.5787)‡	9.3171(0.1983)‡	14.5387(0.7553)			AMR(%)	13.8472(0.1820)	14.6355(0.7457)	14.9510(0.7319)	**13.7727(0.08830)**
		AQC	828.7284(3.8754)†	**401.1386(7.2451)‡**	587.8209(7.1875)	636.3316(6.8927)			AQC	**1119.9139(14.5362)‡**	1348.7448(10.3948)	1591.6591(3.7099)	1401.9790(5.7011)

† denotes the performance of MOAA is significantly better than other peers according to the Wilcoxon's rank sum test at a 0.05 significance level;

‡ denotes the corresponding algorithm significantly outperforms MOAA.

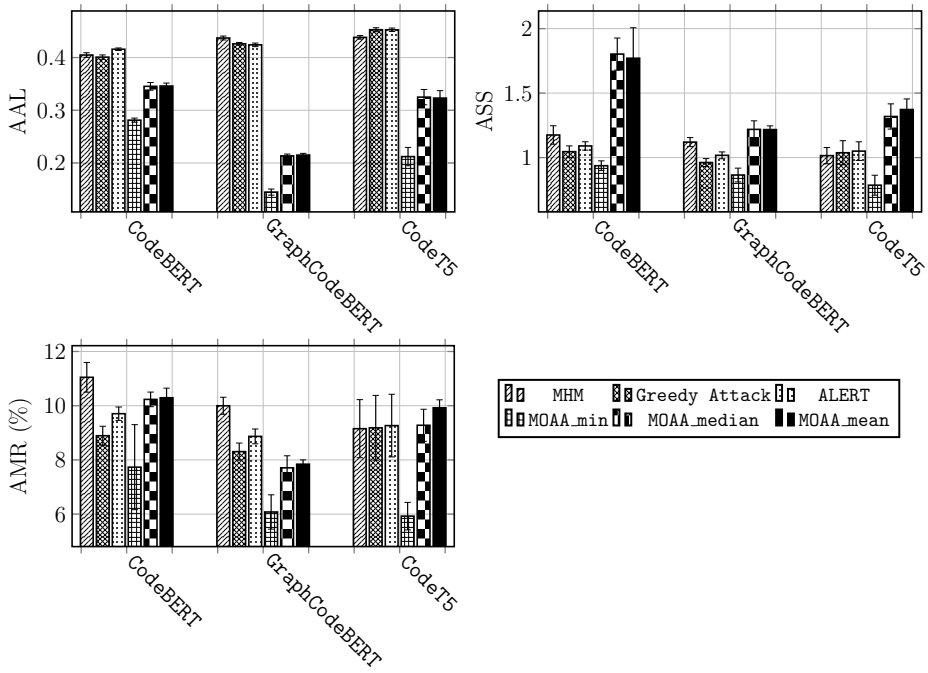


Fig. 19. The AAL, ASS, and AMR metrics when selecting the minimum, median, and average values from the population generated by MOAA for the `defect detection` task.

B.3 Ablation Study of Importance Score and Identifier Name Prediction

B.3.1 Methods. To investigate the usefulness of importance score and identifier name prediction, we develop three variations:

- **MOAA-v1**: This variant considers a vanilla NSGA-II without importance score and identifier name prediction. Instead, MOAA-v1 use randomly select an identifier name for mutation and replaced by a token provided by ALERT.
- **MOAA-v2**: This variant is similar to MOAA-v1 except using the importance score the same as MOAA to guide the mutation process.
- **MOAA-v3**: This variant is similar to MOAA except using random selection.

Note that the other parameter settings are kept the same as Section 5.2.

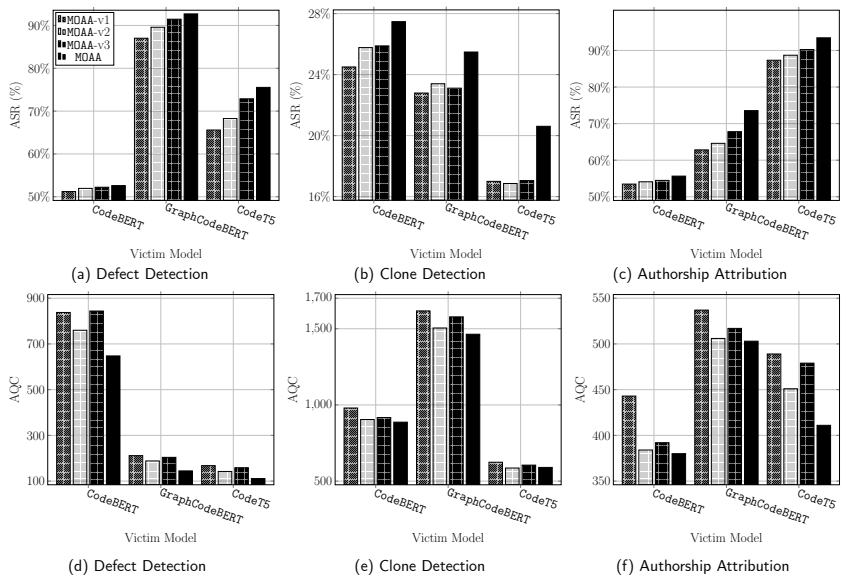


Fig. 20. Bar charts of the attack success rate and average query count of MOAA-v1, MOAA-v2, MOAA-v3 and MOAA on three victim models for the defect detection, clone detection and authorship attribution tasks.

B.3.2 Results and Analysis. Fig. 20 exhibits the result of the ablation study. We observe that MOAA consistently outperforms the other three variations in terms of ASR and AQC. This means that both the importence score and identifier name prediction are crucial for the success of MOAA.

B.4 Examples of Generated AEs

Table 5 presents examples of AEs generated by MHM, Greedy Attack, ALERTand MOAAfor CodeBERT on the Defect Detection dataset. In the first example, only MOAA manages to generate a successful AE, despite MHM, Greedy Attackattempt to substutute all identifiers in the original code. This shows the superior effeiciency of the search space utilized by MOAA. Regarding the second example, while all attackers achieve success, MOAA distinguishes itself by opting to replace 'mr' with 'memory_map', a choice informed by the function name 'set_system_memory_map'. This strategic replacement demonstrates MOAA's ability to grasp the broader context implied by the code, as opposed to the

other methods which rely solely on the superficial information provided by the identifier name 'mr', thus struggling to comprehend the code's context.

Table 5. Examples of AEs generated by MHM, Greedy Attack, ALERT and MOAA for CodeBERT on Defect Detection dataset.

Attacker	Input	Output	Success

Original	int ff_schro_queue_push_back(FFSchroQueue *queue, void *p_data)
	{
	FFSchroQueueElement *p_new = av_mallocz(sizeof(FFSchroQueueElement));
	if (!p_new)
	return -1;
	p_new→data = p_data;
	if (!queue→p_head)
	queue→p_head = p_new;
	else
	queue→p_tail→next = p_new;
	queue→p_tail = p_new;
	++queue→size;
	return 0;
	}
		1	

MHM	queue→port; p_data→pfdat; p_new→p_fresh; size→name	1	×
Greedy Attack	queue→port; size→address; p_data→pfdat; p_new→pockNEW	1	×
ALERT	N/A	1	×
MOAA	queue→p_list	0	✓

Original	void set_system_memory_map(MemoryRegion *mr)
	{
	memory_region_transaction_begin();
	address_space_memory.root = mr;
	memory_region_transaction_commit();
	}
		1	

MHM	mr→dr	0	✓
Greedy Attack	mr→mm	0	✓
ALERT	mr→fr	0	✓
MOAA	mr→memory_map	0	✓

B.5 Subjective Study

Fig. 21 shows the questionnaire used in the subjective study. We also evaluate the naturalness of the AEs generated by each algorithm by conducting a user study. Following previous works [67, 78], we invite 10 non-author participants who possess a Bachelor/Master degree in Computer Secience. For this study, to accommodate the preferences of the participants, we select tasks of different programming languages for each of them. To alleviate recognition burden, we then filter the dataset to exclude code snippets longer than 200 tokens. To make comparison fair, we only select examples that can be successfully attacked by all four algorithms. We randomly pick up 100 examples from the remaining code snippets. For each of these examples, we take the AEs generated by each

algorithm[11]. Therefore, we ask the participants to grade 400 $\langle \mathbf{x}, \mathbf{x}' \rangle$ pairs blindly in a 5-point rating scale. They are instructed to focus on whether the replaced identifiers are natural in the context of the code snippet.
In this subjective study, we follow the grading criteria as follows:

- 5 for highly natura and contextually appropriate identifier replacement;
- 4 for natural identifier replacements with minor contextual discrepancies;
- 3 for correct but less natural identifier replacements;
- 2 for awkward or contextually inappropriate identifier replacements;
- 1 for incorrect, nonsensical, or misleading identifier replacements.

Table 2 presents the population of AEs generated by `MOAA` when attacking `CodeBERT` model for the `Defect Detection` dataset. The diversity observed in these exampls underscores a significant challenge for decision-makers, i.e. determining superiority among the AEs is not straightforward. This highlights the critical role of diversity when generating AEs.

[11]For `MOAA` we randomly select one AE from the generated population.

The Naturalness of Identifier

Welcome to our survey! Your feedback is invaluable in helping us understand how natural the replacement of certain identifiers in code snippets appears to experienced developers like you. Below, you will find some code excerpt where a specific identifier has been replaced. Your task is to assess the naturalness of this substitution based on your expertise and intuition about coding practices.

Thank you for participating in our study. Let's get started!

1. Read the following code:

```
AVFrame *avcodec_alloc_frame(void)
{
    AVFrame *frame = av_mallocz(sizeof(AVFrame));
    if (frame == NULL)
        return NULL;
    FF_DISABLE_DEPRECATION_WARNINGS
    avcodec_get_frame_defaults(frame);
    FF_ENABLE_DEPRECATION_WARNINGS
    return frame;
}
```

Is '**buffer**' replace '**frame**' natural?

Very Unnatural | Unnatural | Neutral | Natural | Very Natural

2. Read the following code:

```
static void cpu_set_irq(void *opaque, int irq, int level)
{
    CPUState *env = opaque;
    if (level) {
        CPUIRQ_DPRINTF("Raise CPU IRQ %d\n", irq);
        env->halted = 0;
        env->pil_in |= 1 << irq;
        cpu_check_irqs(env);
    } else {
        CPUIRQ_DPRINTF("Lower CPU IRQ %d\n", irq);
        env->pil_in &= ~(1 << irq);
        cpu_check_irqs(env);
    }
}
```

Is '**que**' replace '**env**' natural?

Very Unnatural | Unnatural | Neutral | Natural | Very Natural

Fig. 21. The questionnaire used in the subjective study.