CodeBC: A More Secure Large Language Model for Smart Contract Code Generation in Blockchain

Anonymous ACL submission

Abstract

Blockchain, a decentralized distributed ledger database, records transactions across multiple computers in a secure, transparent and tamperresistant manner. To ensure this, smart contract code is introduced to predefine transaction rules, and stipulate that the code should automatically execute without intermediaries when someone calls it. That is, if malicious actors call the code with vulnerabilities, these automatic execution codes may cause significant economic losses to users. Therefore, the 011 security of smart contract code is crucial in Blockchain domain. Currently, smart contracts 014 are primarily manually written by developers, facing challenges such as experienced developer shortage, low development efficiency, and substantial security risks. There is an urgent 018 need for code generation technology to assist 019 both developers and non-professional programmers in creating secure and efficient smart contract codes. In this paper, we propose CodeBC, a more secure smart contract Code generation model for Blockchain, which employs a twostage fine-tuning approach based on CodeLlama: the first stage uses a multi-task learning framework for code infilling and vulnerability detection, enhancing the model's understanding 027 of smart contract code and its ability to identify security vulnerabilities; in the second stage, tags-guided instruction fine-tuning is employed 031 to improve the model's comprehension of human instructions, thereby generating highersecurity code. We construct an Blockchain-HumanEval dataset to assess whether the generated code meets human requirements. Experimental results demonstrate that CodeBC achieves higher BLEU, CodeBLEU, compilation pass rates and lower vulnerability rates compared to baselines, validating the effectiveness of our two-stage fine-tuning strategy.

1 Introduction

043

Blockchain is a decentralized and distributed ledger database that records transactions across multiple



Figure 1: An example of smart contract reentrancy attack. The process of reentrancy attack by contract(b) on contract(a):① The attacker initiates a withdrawal request; ② EtherStore responds and automatically calls the *fallback()* function. ③ The attacker re-initiates the withdrawal request. Since the account balance has not been cleared, the loop can continue.

computers in a secure, transparent, and tamperresistant manner. It is considered as a core technology supporting the development of digital economy and web3.0 by both academic and industrial communities (Bitcoin, 2008; Wood, 2022), providing a secure foundation for various scenarios, such as digital currencies, data asset exchange, and renewable energy trading (Zheng, 2017).

To ensure the security and transparency of transactions, blockchain requires developers to predefine transaction rules through smart contract code, and stipulate that the code should automatically execute without intermediaries when someone calls it. Therefore, the security of smart contract code is crucial, because if the automatic code contains security vulnerabilities, malicious actors could exploit them, leading to significant financial losses for users. Figure 1 illustrates an example of a reentry attack. A typical withdrawal operation should clear the account balance before processing the withdrawal. However, the withdrawal function defined in con-

065tract(a) does the opposite. So, when an attacker's066contract(b) continuously calls contract(a), it results067in erroneous behavior of continuously sending vir-068tual currency. Statistics ¹ reveal a significant rise in069economic losses due to smart contract vulnerabili-070ties year by year, reaching around \$75.82 billion in0712023 compared to approximately \$3.77 billion in0722022. Therefore, urgent measures are required to073enhance the security of blockchain smart contract074code and improve vulnerability detection accuracy.

Currently, smart contracts are manually crafted by developers, facing challenges such as a shortage of developers, low development efficiency, and high-security risks. Firstly, code development of smart contract requires professional knowledge, leading to a severe mismatch between the numbers of developers and users, with over a hundred million blockchain users and just over twenty thousand active developers. Secondly, the time-consuming nature of manual code writing greatly affects development efficiency, often involving repetitive code and requiring weeks to months of development time. Finally, some developers lack professionalism, leading to frequent security incidents and escalating economic losses. If high-quality code can be generated by LLMs, it can not only enhance developer efficiency but also reduce security risk and prevent economic losses.

086

087

096

098

101

102

103

104

106

107

108

109

110

111

112

113

In this paper, we propose CodeBC, a more secure smart contract Code generation model for Blockchain, employing a two-stage fine-tuning approach to enhance the model's understanding of smart contract code and its ability to follow human instructions. Specifically, in the first stage, we employ a multi-task learning fine-tuning strategy, involving code infilling and vulnerability detection tasks, to strengthen the model's comprehension of smart contract code and its awareness of security vulnerabilities. In code infilling task, we randomly mask some lines of code as input and make the masked code as output, following CodeLlama. In vulnerability detection, we input the correct or flawed code into the model, with the correct or bug as output. In the second stage, we employ tags-guided instruction fine-tuning to improve the model's execution of human instructions. We consider the code annotation as human instruction and concatenate them with correct or vulnerable tags as input. Then utilizes its corresponding code as

output, facilitating its understanding of tags' meanings and enabling the generation of higher-security, correct smart contract code.

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

We construct an Blockchain-HumanEval dataset to assess whether the generated code meets human requirements. Experimental results demonstrate that our model achieves higher BLEU, CodeBLEU scores, compilation pass rates, and vulnerabilityfree rates compared to the baseline CodeLlama.

The innovations of this paper are as follows:

- We are the first to introduce an LLM-based code generation model into blockchain field, fostering interdisciplinary research.
- We propose a customized two-stage finetuning strategy: multitasking learning to enhance adaptability to blockchain domain and tags-guided approach to improve understanding of human instructions.
- We construct the first human evaluation dataset for blockchain contract code generation, and experimental results demonstrate the effectiveness of our model.

2 Related Work

2.1 Code Generation

With the development of LLMs, the usability of code generation models has increased significantly, especially after the successful commercial application of CodeX (Chen et al., 2021). More and more researchers (Xu et al., 2022; Zan et al., 2023; Lu et al., 2022) focus on code generation tasks. Alpha-Code (Li et al., 2022) significantly improved the code generation model's ability to understand complex instructions and implement complex codes by generating a large number of code samples and effectively filtering them. However, the expensive execution computation of AlphaCode poses challenges for its real-world application. Nijkamp et al. (2022) propose CodeGen, which understands complex instructions by splitting them while controlling model computation costs. Incode (Fried et al., 2022) and CodeGeex (Zheng et al., 2023) attempt to enhance the alignment between model-generated code and instructions in different ways. To improve the usability of generated code, Wang et al. (2022) utilize reinforcement learning to increase the compilation rate of generated code. Zhang et al. (2023) patch generated code through error reports and runtime execution results to enhance the correct execution rate of generated code. Yin et al.

¹https://www.slowmist.com/report/2023-Blockchain-Security-and-AML-Annual-Report(EN).pdf

(2023) propose vertical-domain code generation
models for interactive data processing tasks. However, these works only focus on code compilability
and execution results, neglecting code vulnerabilities' impact on application stability. In the highsecurity blockchain domain, we develop a tailored
and security-enhanced code generation model for
smart contract to enhance its usability.

2.2 Vulnerability Detection

The task of vulnerability detection is highly crucial 172 in blockchain smart contracts (Brent et al., 2018; 173 Mehar et al., 2019). Oyeente (Luu et al., 2016) 174 is the first smart contract vulnerability detection 175 technique, which detects vulnerabilities by con-176 structing a contract control flow graph, executing 177 symbolic states, and comparing them with proper-178 ties defined based on vulnerabilities. Subsequently, 179 researchers (Jiang et al., 2018; Grishchenko et al., 180 2018; Rodler et al., 2018) proposed smart contract vulnerability detection techniques based on fuzz testing, taint analysis, and formal verification. 183 However, these traditional methods have the prob-184 lems of low coverage and low efficiency. Recently, Tann et al. (2018) used smart contract data to train 186 a binary classifier based on LSTM. Liu et al. (2018) propose S-gram to identify irregular subsequences as candidate vulnerabilities by training a language model and scanning the token sequences of the tar-190 get contract for auditing. He et al. (2019) propose 191 ILF to use the symbolic execution results as the 192 training dataset for neural networks, and generate input sequences for the test program utilizing neu-194 195 ral networks. Gao et al. (2020) propose SmartEmbed to characterize the stream of contract symbols 196 extracted from AST based on word embedding and 197 vector space techniques, and detect vulnerabilities by comparing the similarity with the vulnerable 199 contracts. However, current detection methods rely solely on simple classification models and may not fully comprehend the complex logic of code. In this paper, we design various strategies to enhance vulnerability detection capabilities, thereby preventing the model from generating vulnerable code.

3 CodeBC Model

3.1 Task Definition

207

208

210

211

The dataset of smart contract code generation task is $\mathbb{D} = \{(S_1, T_1, C_1), \dots, (S_N, T_N, C_N)\}$, where S_i is a natural language instruction, T_i is a security tag and C_i is a smart contract code. Given a natural language instruction $S_i = \{s_1, \ldots, s_{|S_i|}\}$, where s_i is a instruction token of S_i , and a security tag $T_i \in \{correct, bug\}$, the goal of CodeBC model is to generate code $C_i = \{c_1, \ldots, c_{|C_i|}\}$ based on the generative probability $P(C_i|S_i, T_i)$ by restricting tag T_i and instruction S_i , where c_i is the code token. The $P(C_i|S_i, T_i)$ can be represented as:

212

213

214

215

216

217

218

220

221

222

223

224

225

226

227

229

230

231

232

233

234

236

237

238

239

240

241

242

243

244

245

246

247

248

249

253

254

255

$$P(C_i|S_i, T_i) = \frac{P(C_i)P(T_i|C_i)P(S_i|T_i, C_i)}{P(S_i, T_i)}.$$

The overall structure of the CodeBC model consists of two stages, as shown in Figure 2. The first stage multi-task learning fine-tuning includes two tasks: code infilling task to optimize $P(C_i)$ and vulnerability detection task to optimize $P(T_i|C_i)$. From a probabilistic decomposition perspective, this can theoretically improve the generative probability $P(C_i|S_i, T_i)$. The second stage tags-guided instruction fine-tuning directly optimizes $P(C_i|S_i, T_i)$ based on different T_i , namely "correct" and "bug", which can help the model generate more secure code in the inference process when $T_i = correct$.

3.2 Multi-task Learning Fine-tuning

The multi-task learning fine-tuning consists of two tasks: code infilling and vulnerability detection, as shown in the left and middle of Figure 2.

3.2.1 Code Infilling

For the code infilling task, we follow the efficient training method in CodeLlama (Bavarian et al., 2022), which splits the text into three parts: prefix, midfix, and suffix. We adjust the position of each part so that the model can make full use of contextual information when predicting the midfix code. PSM denotes the combination of prefix, suffix, and midfix. SPM denotes the combination of suffix, prefix, and midfix. The three tokens <PRE>, <MID>, and <SUF> are used to denote the beginning of each part. Given the i^{th} example $C_i = (Pre_i, Mid_i, Suf_i) \in \mathbb{D}$, we minimize the negative log-likelihood of the target Mid_i :

$$\mathcal{L}_{I} = -\sum_{i=1}^{|\mathbb{D}|} \sum_{t=1}^{|Mid_{i}|} \log P(m_{t}|m_{1}...m_{t-1}, Pre_{i}, Suf_{i}; \theta),$$
251

where m_t denotes the token of Mid_i , and θ denotes the parameters of CodeLlama.

Due to the short length of some smart contracts and the fact that the contract usually starts with the



Figure 2: An illustration of the CodeBC model. (1) We first performed multi-task learning: (1 - 1) First fine-tuned the base pre-trained language model on smart contract code. (1 - 2) Continued to train the model with the goal of vulnerability detection task on the model with the first smart contract programming capability. (2) Finally, Tags-guided Instruction Fine-tuning was performed on the models trained in the first stage, aligning the vulnerability detection tags in stages (1 - 2).

version number of the used solidity, completely randomly dividing the contract code into three parts may make the model unable to effectively learn the contextual information. Therefore, when splitting the code, the contract code is first evenly divided into five parts according to the number of characters, and randomly selecting cut-off points in the second and fourth parts from left to right ensures that the data participating in the training has valid contextual information, as shown in the left of Figure 2. Then we randomly select the combination method among PSM and SPM with probability 0.5.

3.2.2 Vulnerability Detection

256

257

261

265

267

272

273

276

277

281

283

The main purpose of the vulnerability detection task is to strengthen the model's ability to sense security vulnerabilities in smart contracts and provide alignment guidance for subsequent Tags-guided Instruction Fine-tuning. Given the code C_i , if C_i has no vulnerabilities, the ground-truth output is $T_{g_i} = \{[Tag] < correct>[/Tag]\}$, otherwise $T_{g_i} = \{[Tag] < bug>[/Tag]\}$, where [Tag], [/Tag] respectively denote the beginning and the end of the code security detection result.

The model performs vulnerability detection task under a simple prompt "whether this smart contract Code is a correct solution:" and the negative loglikelihood of the target sentence T_{q_i} is:

$$\mathcal{L}_{ ext{D}} = -\sum_{i=1}^{|\mathbb{D}|} \sum_{j=1}^{|T_{g_i}|} ext{logP}(t_j | t_1 ... t_{j-1}, C_i; heta_I),$$

where t_j represents the token of T_{g_i} , and θ_I denotes the parameters of CodeLlama after code infilling. 284

285

286

287

288

290

291

292

293

294

295

296

297

298

300

301

302

303

305

306

307

309

3.3 Tags-guided Instruction Fine-tuning

The goal of tags-guided instructions fine-tuning is to use tags to assist the model in better distinguishing between correct code and vulnerable code. In the training process, as shown in the right-bottom of Figure 2, given a human instruction S_i and the tag information T_i , the objective of code generation model is to produce the ground-truth code C_i , where $T_i \in \{correct, bug\}$ indicates whether the code C_i is correct or vulnerable. We minimize the negative log-likelihood of the target code:

$$\mathcal{L}_{\mathrm{G}} = -\sum_{i=1}^{|\mathbb{D}|} \sum_{t=1}^{|C_i|} \log \mathsf{P}(c_t | c_1 ... c_{t-1}, S_i, T_i; \theta_D),$$

where c_t represents the t^{th} token of C_i , and θ_D denotes the parameters after vulnerability detection in section 3.2.2.

In the inference process, as shown in the righttop of Figure 2, given a human instruction S_i and tag information $T_i = correct$, the model will generate safer code C_p with the help of $T_i = correct$.

4 **Experiments**

4.1 Experimental Settings

We first introduce some empirical settings, including datasets, evaluation metrics, baselines and parameter settings for CodeBC.

Dataset	Total	Num_Security	Num_Insecurity
SASCsmall	10571	2693	7878
SASCsmall-F	7897	2112	5785
SASCsmall-F-CI	2112	2112	0
SASCsmall-F-VD	7567	1782	5785

Table 1:The statistics of datasets for smartcontract code generation task.Num_Security andNum_Insecurity are the number of security and insecurity smart contracts, respectively.

4.1.1 Dataset

311

312

313

314

315

316

317 318

319

321

322

323

324

325

329

330

334

335

336

337

338

341

344

345

346

347

348

Smart Contract Code Dataset

The dataset for smart contract code generation task is primarily based on the "Slither Audited Smart Contracts small-multilabel"(SASCsmall) dataset (Rossini, 2022), which is built on verified smart contracts provided by the Blockchain Ethereum community. They utilize Slither(Feist et al., 2019) to analyze each smart contract and map the detected vulnerability types to 9 classes based on the most threatening smart contract vulnerabilities provided by the Decentralized Application Security Project (DASP).² As shown in Table 1, the dataset contains a total of 10571 smart contract codes, with secure contracts accounting for 25% and insecure contracts for 75% of the total. To ensure that all smart contracts involved in model training can be deployed in practical application scenarios, we conducted a selection process for smart contracts. We excluded code that encapsulated multiple smart contracts to form complete and complex functionalities from SASCsmall, and finally obtain our dataset SASCsmall-F.

> Based on SASCsmall-F dataset, we constructed different datasets for three training stages, and randomly split them into training, validation, and test sets in an 8:1:1 ratio. The statistics of these datasets are shown in Table 1:

 Code Infilling dataset(SASCsmall-F-CI): To maximize the model's performance in generating secure code, the code infilling task selected 2112 secure smart contract codes from SASCsmall-F that are free of security vulnerabilities. Additionally, each smart contract involved in training is randomly split five times and completely shuffled.

• Vulnerability Detection dataset(SASCsmall-F-VD): The training data for the vulnerability detection task utilizes all smart contract codes with security vulnerabilities from the SASCsmall-F dataset. To ensure a roughly equal sample length for both secure and vulnerable types of samples involved in the training, excessively long smart contract codes from the secure contract codes are filtered out.

350

351

352

353

354

355

356

357

358

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

377

378

379

380

381

382

383

384

385

386

388

389

390

391

392

393

• Tags-guided Instruction dataset: the entire dataset from SASCsmall-F.

Blockchain-HumanEval Dataset

The Blockchain-HumanEval dataset proposed in this paper is constructed based on the open-source smart contract code repository Openzeppelin.³ The specific reasons are as follows:

- As the most popular and verified contract code repository, the code it provides can be considered as correct smart contract code without security vulnerabilities.
- As a contract code that provides security guarantees for smart contracts, it imposes higher requirements on its security which will place high demands on the security of the modelgenerated code.
- Although it is also smart contract code, it does not have complete practical application functionality, and there is no complete code information on Blockchain Ethereum, so data leakage will not occur in the training data.

The Blockchain-HumanEval dataset, proposed in this paper, selects smart contract code from the Openzeppelin code repository that does not have internal references as the target code. Additionally, natural language instructions are manually added to these contract codes, and detailed specifications of the Solidity version number, the names of functions, events, and errors, as well as internal parameters and their functions, are provided. A total of 41 missions. The SASCsmall-F and Blockchain-HumanEval dataset will be released when this paper is accepted.

4.1.2 Baselines and Parameter Settings

We compare our CodeBC model with three baselines, including CodeLlama-7b-Instruct (Roziere et al., 2023), CodeGen25-7b-Instruct (Nijkamp et al., 2023), and DeepSeek-Coder-6.7b-Instruct (Daya Guo, 2024). In the model training

²https://dasp.co/

³https://github.com/OpenZeppelin/openzeppelincontracts

phase, all three stages of training are performed using LoRA (Hu et al., 2021) on the A100 GPU, which has been shown to reduce training time significantly while ensuring good results. The rank of the update matrix (lora-r) is set to 4, the LoRA scaling factor (lora-alpha) is set to 32, and all parts of the attention block are used as target modules. The learning rate is set to 1e-4, and ten rounds are trained on the Code Infilling and Vulnerability Detection tasks, respectively, and one round is trained on the Tags-guided Instruction task. In the model inference stage, we ran all models using a temperature value of 0.2 and a nucleus sampling method with a parameter value of 0.95.

4.1.3 Metrics

395

396

400

401

402

403

404

405

406

407

408

409

410

411

412

413

414

415

416

417

418

419

420

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

436

437

438

439

440

Each model generates five samples for each natural language instruction from the Blockchain HumanEval dataset. We evaluate the models based on three aspects: code generation quality, security, and vulnerability detection accuracy. The specific evaluation metrics are as follows:

BLEU(Papineni et al., 2002) and Code-BLEU(Ren et al., 2020) are employed to evaluate the quality of smart contract code in meeting the requirements of natural language instructions. We calculate the average BLEU(AvgBLEU), the best BLEU(BestBLEU), the average Code-BLEU(AvgCB) and the best CodeBLEU(BestCB) for evaluation. We do not use the metric pass@k because of the nature of smart contracts that can only be called passively. To better assess the quality of the generated code, we removed all comments and code description sections from the target code and the generated samples, and performed the metrics calculations only on the code.

The performance of security is evaluated using the review results of the Slither inspection tool (Feist et al., 2019), which is the most popular review tool in the Blockchain community and is officially recommended by Ethernet. ⁴ The tool returns the compilation results of the code and returns security vulnerability analysis results for the compilable contract code. We use the compilation pass rate(ComPass) and the vulnerability rate(VulRate) of the generated code to visualize the performance of the generated results in terms of security, and the safe-availability rate(SafeAval)⁵ to further reflect the model's ability in terms of security.

In order to test the performance of vulnerability detection for our model, we utilize handwritten smart contract code provided by the open source project SmartBugs (Durieux et al., 2020), providing a total of 141 smart contract code for different types of security vulnerabilities, each type of vulnerability is briefly described in Appendix A. We calculate the accuracy of the binary classification vulnerability detection task. 441

442

443

444

445

446

447

448

449

450

451

452

453

454

455

456

457

458

459

460

461

462

463

464

465

466

467

468

469

470

471

472

473

474

475

476

477

478

479

480

481

482

483

484

485

486

487

488

489

490

4.2 Experimental Results

In this section, we demonstrate our experiment results on BlockChian-HumanEval datasets and the ablation experimental results.

4.2.1 Metric-based Evaluation

The metric-based evaluation results are shown in Table 2. From the results, we can see that in terms of meeting the demands of natural language instructions, CodeGen25 and DeepSeek-Coder perform basically the same, slightly worse than CodeLlama, and the behavior on the metrics AvgBLEU, Best-BLEU, AvgCB and BestCB are lower than CodeLlama by about 8.25%, 3.53%, 9.26%, 7.18% respectively. In terms of security, CodeGen25 and DeepSeek-Coder also perform roughly the same, and are significantly worse than the CodeLlama model, with about 19.27% lower compilation pass rate, and about 16.58% more vulnerability rate, and about 18.29% fewer contract code that can be used safely. This shows that CodeLlama outperforms the other two baseline models both in terms of fit with natural language descriptions and security.

Our CodeBC model has excellent performance in all the metrics, compared with all baselines. In terms of satisfying natural language instruction requirements, it achieves scores above 0.64 in AvgBLEU, BestBLEU, AvgCB, and BestCB metrics. Compared to the best-performing CodeLlama model in the baseline models, it shows improvements of approximately 16.54%, 13.03%, 11.35%, and 8.57% respectively in the above metrics. In terms of security, compared to the CodeLlama model, the compilation pass rate has increased from 43.90% to 86.82%, an increase of 42.92%, the vulnerability rate has decreased from 61.95% to 26.34%, a decrease of 35.61%, and the code of the contract that can be used safely has increased from 40.48% to 78.56%, an increase of 38.08%.

From the above experimental results, we can see that our CodeBC proposal outperforms the baseline

⁴https://ethereum.org/zh/developers/docs/smartcontracts/testing

⁵The proportion of smart contract code that is compilable and free of security vulnerabilities

Models	AvgBLEU	BestBLEU	AvgCB	BestCB	ComPass(%)	VulRate(%)	SafeAval(%)
CodeGen25	0.4298	0.5614	0.4411	0.5111	24.87	79.02	22.43
DeepSeek-Coder	0.4251	0.5650	0.4371	0.5084	24.39	78.04	21.95
CodeLlama	0.5099	0.5985	0.5317	0.5816	43.90	61.95	40.48
CodeBC(our)	0.6753	0.7288	0.6452	0.6674	86.82	26.34	78.56

Table 2: The metric-based evaluation results.

Models	AvgBLEU	BestBLEU	AvgCB	BestCB	ComPass(%)	VulRate(%)	SafeAval(%)
CodeBC	0.6753	0.7288	0.6452	0.6674	86.82	26.34	78.56
CodeBC-CI	0.5279	0.6271	0.5739	0.6247	51.70	57.07	47.80
CodeBC-w.o.CI&TI	0.5759	0.6735	0.5254	0.5633	52.68	53.66	65.33
CodeBC-w.o.CI&VD	0.5698	0.6298	0.6001	0.6345	77.06	44.87	68.78

Table 3: Evaluation results of each stage of the model on the Blockchain-HumanEval dataset.

Models	AvgBLEU	BestBLEU	AvgCB	BestCB
CodeLlama	0.1842	0.1843	0.5225	0.5231
our CodeBC-CI	0.4771	0.4829	0.8032	0.8042

Table 4: The test results of CodeLlama and ourCodeBC-CI model for code infilling task on Blockchain-HumanEval dataset.

model in both the satisfaction of natural language instruction requirements and security. The efficient code-infilling fine-tuning task allows the model to learn smart contract programming well, additionally, multiple tasks involving the construction of smart contract code provide the model with repeated exposure to smart contract code data, which significantly improves CodeBC's ability to generate smart contracts. As for the vulnerability detection task and the tags-guided instruction finetuning focuses on improving the security of the model-generated code by improving the model's ability to sense vulnerabilities and directing the model to generate more secure smart contract code through tags when generating smart contracts. The improvement of CodeBC over the baseline model in various metrics demonstrates the effectiveness of our two-phase fine-tuning strategy.

491

492

493

494

495

496

497

498

499

501

502

504

507

508

509

4.2.2 Ablation Experiments

To verify the actual effect of the training of the 510 models in each stage, we conducted ablation ex-511 periments, and the test results of the models in 512 513 each stage on Blockchain-HumanEval are shown in Table 3. CodeBC-CI denotes the model fine-514 tuned only through the code completion task. 515 CodeBC-w.o.CI&TI denotes the model fine-tuned 516 only for the vulnerability detection task. CodeBC-517

w.o.CI&VD denotes the model fine-tuned only after label-guided instruction fine-tuning. CodeBC-VD denotes the model fine-tuned for the first-phase multi-task learning. 518

519

520

521

522

523

524

525

526

527

528

529

530

531

532

533

534

535

536

537

538

539

540

541

542

543

544

545

546

547

548

549

550

551

552

From the experimental results, it can be observed that the training methods in each stage can improve the score of AvgBLEU, BestBLEU, AvgCB, and BestBLEU, which represent the code generation and natural language requirements. By comparing the changes in these four metrics for CodeBC-CI, CodeBC-VD, and CodeBC, it can be seen that the training in all three stages contributes to the performance of the final model. This indicates that as long as the model is exposed to more code data during the training process, its code generation capability can be improved.

In terms of security, the training methods used in each stage also improve the model's performance in security aspects. However, comparing CodeBC-CI with other stages reveals that although the training dataset in this stage only includes contract code without vulnerabilities, the lack of security information in the training data does not significantly improve the security of the generated code by the model. On the other hand, comparing CodeBC-VD with CodeBC-w.o.CI&TI shows that the smart contract programming knowledge learned through code infilling tasks enhances the model's vulnerability awareness, thereby improving the security of the generated code.

4.3 Analysis

In this section, we tested the performance of the CodeBC model on the code completion task and the vulnerability detection task, respectively, to analyze the reasons for the performance improvement of

Models	Accuracy(%)										
	RE	AC	AR	ULLC	DoS	BR	FR	TM	other	all	
CodeBC-VD	90	33	0	40	50	100	25	80	67	52	
CodeBC	90	17	0	42	67	100	50	100	67	52	

Table 5: The test results of CodeBC-VD and CodeBC model for vulnerability detection task on SmartBugs dataset.

the CodeBC model due to the multi-task learning fine-tuning strategy.

4.3.1 Code Infilling

553

554

557

558

559

561

564

565

567

571

573

574

575

576

577

578

582

583

584

586

590

591

594

We tested the smart contract code-filling capabilities of the CodeLlama and the CodeBC-CI, respectively. The results of the assessment are shown in table 4. From the results, we can see that the CodeBC-CI model performs better than the CodeLlama model in completing contract code infilling tasks, with approximately a 0.3 improvement in both BLEU and CodeBLEU metrics. This indicates that fine-tuning the model through code infilling tasks enables it to learn the structural information of smart contract code and better understand the contextual information of the code, thereby enhancing the model's ability to program smart contracts.

As demonstrated in the ablation experiments in section 4.2.2, the structural information of smart contracts learned through code infilling task can improve the model's vulnerability awareness in vulnerability detection task, which enhances the security of the generative smart contract code.

4.3.2 Vulnerability Detection

The results of the vulnerability detection part are shown in Table 5. From the results, we can see that the model can perceive 52% of the vulnerabilities, and there is no significant decrease in the ability to perceive the vulnerabilities after training in the Tags-guided Instruction phase. The model has a good ability to perceive vulnerabilities such as Reentrancy(RE), Bad Randomness(BR) and other vulnerabilities that can easily cause economic losses.

In order to further validate the impact of model vulnerability-awareness capability on improving the security of model-generated code, we analyzed the vulnerabilities of model-generated smart contracts at each stage, and the results are shown in Table 6. From the results, we can see that the smart contract code generated by the model after fine-tuning the vulnerability detection task can well avoid contract vulnerabilities such as re-entry at-

Models	RE	AC	AR	ULLC	other	ignore	
CodeBC-CI	5	4	6	0	2	0	
CodeBC-VD	0	0	0	0	7	13	
CodeBC	0	0	0	1	11	15	

Table 6: Number of vulnerabilities of each type of model-generated contract in each phase.

tacks, access control, and so on, which appear in the contract code generated by CodeBC-CI. Besides, these models primarily generate vulnerabilities that are ignored because they do not pose an actual security threat. Smart contracts with these types of vulnerabilities are labeled as secure in the training data. 595

596

597

598

600

601

602

603

604

605

606

607

608

609

610

611

612

613

614

615

616

617

618

619

620

621

622

623

624

625

626

627

The above results show that the model's improved vulnerability sensing capability through the vulnerability detection task well enhances the security of the model-generated smart contract code.

5 Conclusion

In this paper, we propose CodeBC, a code generation model for blockchain smart contracts, and design a two-stage fine-tuning strategy for the security requirements of smart contracts. To validate the performance of CodeBC on the smart contract generation task, we propose the Blockchain-HumanEval dataset. Through experiments, we show that our model improves by nearly 17% in BLEU, 13% in CodeBLEU, 42% in compilability rate, 36% in vulnerability rate, and the percentage of contract code that is safe to use increased from 40% to 78%. In the future, we will leverage the control flow graph of smart contracts to design more stringent security policies to ensure that the model can generate secure and usable code.

Limitations

Our work has several limitations, which we aim to address in our future work:

Firstly, although our model outperforms other baseline models in terms of evaluation metrics, its performance is still poor for certain tasks, espe-

732

733

cially when it comes to longer code completion. Therefore, further optimization is required.

Secondly, our model was trained only on snippets of smart contract code, without fully utilizing information such as control flow graphs. In the future, we will design further improvements based on the specific characteristics of smart contract data to enhance the quality of generated smart contracts.

Finally, our approach has effectively improved the security of smart contract code generated by the model. However, we have not conducted rigorous experimental validation on other programming languages. In the future, we will consider developing solutions to enhance the security of code generation tasks for a wider range of programming languages.

Ethical Considerations

628

634

637

641

647

648

649

651

670

671

672

673

674

675

677

CodeBC's main goal is to generate smart contract code based on user instructions. Due to the highsecurity requirements of smart contracts, the safety of the generated code was a key consideration during the model design. Our data is collected from open-source projects, respecting the respective license restrictions or publicly available benchmarks. Additionally, since the data used mostly consists of publicly available smart contract code information on the blockchain, it does not involve any privacy data. We also discuss the limitations and future work of this paper.

References

- Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*.
- Nakamoto S Bitcoin. 2008. Bitcoin: A peer-to-peer electronic cash system.
- Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Dejian Yang Zhenda Xie Kai Dong Wentao Zhang Guanting Chen Xiao Bi Y. Wu Y.K. Li Fuli Luo Yingfei Xiong Wenfeng Liang Daya Guo, Qihao Zhu.

2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

- Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, pages 530–541.
- Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pages 8–15. IEEE.
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.
- Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2020. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering*, 47(12):2874–2891.
- Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *Principles* of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7, pages 243–269. Springer.
- Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 531–548.
- Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.
- Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.
- Han Liu, Chao Liu, Wenqi Zhao, Yu Jiang, and Jiaguang Sun. 2018. S-gram: towards semantic-aware security auditing for ethereum smart contracts. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 814–819.

- 734 735

740 741

- 742 743
- 744 745

747

748

- 753
- 756

757 758

- 762 763
- 764

770

772 773

774 775

- 776
- 777

778 779

780 781

787

783

- Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A retrieval-augmented code completion framework. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 6227–6240, Dublin, Ireland. Association for Computational Linguistics.
- Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pages 254-269.
- Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. 2019. Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack. Journal of Cases on Information Technology (JCIT), 21(1):19-32.
- Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. arXiv preprint arXiv:2305.02309.
- Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474.
 - Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting of the Association for Computational Linguistics, pages 311–318.
- Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297.
- Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. arXiv preprint arXiv:1812.05934.
- Martina Rossini. 2022. Slither audited smart contracts dataset.
- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950.
- Wesley Joon-Wie Tann, Xing Jie Han, Sourav Sen Gupta, and Yew-Soon Ong. 2018. Towards safer smart contracts: A sequence learning approach to detecting security threats. arXiv preprint arXiv:1811.06632.

Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable neural code generation with compiler feedback. In Findings of the Association for Computational Linguistics: ACL 2022, pages 9-19, Dublin, Ireland. Association for Computational Linguistics.

788

789

791

792

795

796

797

798

799

801

802

803

804

805

806

807

808

809

810

811

812

813

814

815

816

817

818

819

820

821

822

823

824

825

826

827

828

829

830

831

832

833

834

835

836

837

- G Wood. 2022. What is web 3? here's how future polkadot founder gavin wood explained it in 2014. Coindesk. CoinDesk.
- Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, pages 1–10.
- Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Oleksandr Polozov, and Charles Sutton. 2023. Natural language to code generation in interactive data science notebooks. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 126–173, Toronto, Canada. Association for Computational Linguistics.
- Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet nl2code: A survey. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 7443-7464.
- Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 769–787, Toronto, Canada. Association for Computational Linguistics.
- Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. arXiv preprint arXiv:2303.17568.
- Xie-S. Dai H. Chen X. Wang H. Zheng, Z. 2017. An overview of blockchain technology: Architecture, consensus, and future trends. IEEE international congress on big data (BigData congress), pages 557-564.

Appendix

A Description of Vulnerability Types

Reentrancy(RE): Smart contracts allow external 838 contracts or accounts to be called under certain 839 rules. However, if the calling rules or contract exe-840 cution logic are not properly designed, reentrancy 841 vulnerabilities can arise. A reentrancy vulnerability allows attackers to repeatedly call the same function during the execution of a contract. By exploiting this vulnerability, attackers can reenter a function within the contract during its execution and carry out malicious operations. One notable example of such a vulnerability is the infamous "The DAO" attack, which resulted in nearly \$50 million in losses at the time.

843

844

852

853

870

871

872

873

874

877

878

885

Access Control(AC): In smart contracts, access to functions and data also needs to be restricted through permission controls and condition checks. On one hand, attackers may bypass or tamper with access control mechanisms to gain unauthorized privileges or perform unauthorized operations, leading to sensitive data leaks, financial losses, or disruption of contract logic. On the other hand, overly strict access control can result in virtual currencies and other assets being locked within the contract, causing economic losses. A typical example is the 2017 Parity wallet contract attack. By exploiting a special transaction to bypass access control, attackers gained access to funds within the contract, resulting in approximately \$30 million in losses at the time.

Arithmetic(AR): Also known as integer overflow and integer underflow. Integer overflow vulnerabilities are not a new type of vulnerability, but they pose a particularly significant risk in smart contracts. This is because unsigned integers are commonly used, and many developers are accustomed to using simple types. If an overflow occurs, many seemingly benign smart contracts can become vehicles for theft or denial-of-service attacks. One notable example is the 2018 BEC attack, where attackers caused the price of BEC to plummet to zero by flooding the exchange with a large number of tokens, resulting in immeasurable economic losses.

Unchecked Low Level Calls(ULLC): Solidity allows the use of low-level functions such as *call()*, *delegatecall()*, and *staticcall()*. These functions behave differently from regular functions when it comes to error handling. Instead of propagating or reverting the current execution, they simply return a boolean value of false. If not properly checked and validated, these functions can be exploited by attackers to execute unauthorized operations, manipulate contract state, or steal funds.

Denial of Service(DoS): DoS attacks are prevalent in the field of security, and they also pose a significant risk in blockchain systems, including Ethereum. In Ethereum, a DoS attack can be described as "irreversible malicious operations or infinite resource consumption." Due to the possibility of loops in on-chain calls, Ethereum requires each request to consume a certain amount of gas. Once the gas limit is exhausted, the program cannot continue executing, regardless of the circumstances. Therefore, a DoS attack on the blockchain not only disrupts the normal execution of contract code logic but also results in significant consumption of tokens and gas. 894

895

896

897

898

899

900

901

902

903

904

905

906

907

908

909

910

911

912

913

914

915

916

917

918

919

920

921

922

923

924

925

926

927

928

929

930

931

932

933

934

935

936

937

938

939

940

941

942

943

Bad Randomness(BR): BR has always been a challenge in modern computer systems, and it becomes even more complex in open blockchain networks. In Ethereum smart contracts, for example, implementing logic based on random numbers can be impractical. This is because on-chain information is publicly accessible, and anyone can query and analyze the data stored on the blockchain. If the contract code does not carefully consider the public nature of on-chain data when using random numbers, it can be maliciously exploited for cheating purposes.

Front Running(FR): In Ethereum, all transactions need to be confirmed before being fully recorded on the blockchain. Each transaction rewards miners with a certain amount of transaction fees as an incentive. The amount of fees determines the priority of a transaction being confirmed by miners. Therefore, if an attacker can obtain the specific transaction information or related information of a user in advance and complete the operation before the user does, they can cause losses to the original user by using various means (usually by increasing the bid) to complete the transaction ahead of the user.

Time Manipulation(TM): In Solidity, vulnerabilities related to time-dependent events can arise. The *block.timestamp* in Solidity is controlled by miners, so if the contract logic relies on *block.timestamp*, there is a risk of being exploited. When an attacker (usually a miner) confirms a transaction, they can manipulate *block.timestamp* to their advantage. This allows them to have foreknowledge of the outcome of contract code that depends on the timestamp. The attacker can choose a suitable value to achieve their desired outcome.

B Case Study

The performance of the stage models on the task "Address" will be listed and analyzed here. The

task requires the completion of a library called Address, which provides several functions related to 945 address manipulation. The representation in the 946 Blockchain-HumanEval dataset is shown in Figure 3. The manually written instructions specify the Solidity version number used in the contract, provide 949 an overall description of the contract's functionality, 950 and provide detailed descriptions of the function-951 ality of each function in the contract. In the code section, annotations are removed, and only the code 953 information is retained as the target for generation. 954 The performance of the model CodeLlama on the 955 "Address" task is shown in figure 4. From the results, although the generated code seems to encom-957 pass the implementation of all the functionalities mentioned in the instructions, there are still some details that may pose security risks: Firstly, all functions that should be declared as internal have 961 been declared as public, which could potentially 962 lead to issues with access control and data integrity. 963 Furthermore, the generated smart contract code 964 adopts a strategy of direct and complete rollback 965 when handling execution errors, which is different 966 from the target code that executes different rollback 967 968 strategies based on different return results. The direct and complete rollback strategy may result in irreversible operations and the inability to restore 970 to the previous state, causing serious impacts on the data integrity and consistency of the system. 972 Additionally, the direct and complete rollback can-973 not avoid unnecessary rollbacks, which could be 974 exploited by malicious users to launch denial-of-975 service attacks. Lastly, the generated smart contract 976 declares the function functionCall() as public view, 977 978 which means it can only be used for reading and not for modifying the state. However, the statement 979 *target.call(data)* within the function may modify the state. This error directly leads to the contract 981 failing to compile. 982 983

The performance of the CodeBC-CI model on the "Address" task is shown in the figure5. From the results, there are slight differences in the implementations of different functions, but there is not much difference in terms of functionality. The type of function declarations has been improved by declaring all functions as internal types. However, there has been no improvement in the rollback strategy. At the same time, the function *functionCall()* declares a return value of type bytes memory, but it actually returns the result of type bool from *target.call(data)*, which causes the generated code to still fail to compile.

984

985

987

991

992

995

The performance of the CodeBC-VD model on 996 the "Address" task is shown in the figure6. From 997 the results, it can be observed that the model has 998 further strengthened the security review of various 999 functionalities and correctly declared the functions 1000 that should be declared as internal types. Optimiza-1001 tion has also been done for the rollback strategy. 1002 Additionally, the generated contract code is error-1003 free and can be compiled and executed success-1004 fully. 1005

The performance of the CodeBC model on the 1006 "Address" task is shown in the figure7. Compared 1007 to CodeBC-VD, which has already demonstrated 1008 excellent functionality and security, CodeBC does 1009 not show significant improvements. However, it 1010 has removed some redundant code, indicating an 1011 improvement in the model's understanding of in-1012 structions. 1013

Instrction:

Generate smart contract code that meets the following requirements: Programming with solidity ^ version number 0.8.20 Implement a library named Address for handling address related operations, throwing exceptions when the account balance is insufficient, code at the target address, or internal calls fail The following functions need to be implemented, with the following names and functions: sendValue: Sends Ether to the specified address functionCall: Calls a function at the target address without passing Ether. functionCallWithValue: Calls a function at the target address and passes the specified amount of Ether. functionStaticCall: Call a function at the target address statically. functionDelegateCall: Delegate a call to the target address. verifyCallResultFromTarget: Verifies that the call result was successful and handles exceptions. verifyCallResult: Verifies that the call result was successful and handles exceptions. revert: Rolls back the transaction and throws an exception. Please give the contract code Code: pragma solidity ^0.8.20; library Address { error AddressInsufficientBalance(address account); error AddressEmptyCode(address target); error FailedInnerCall(); function sendValue(address payable recipient, uint256 amount) internal { if (address(this).balance < amount) {</pre> revert AddressInsufficientBalance(address(this)); (bool success,) = recipient.call{value: amount}(""); if (!success) { revert FailedInnerCall(); } function functionCall(address target, bytes memory data) internal returns (bytes memory) { return functionCallWithValue(target, data, 0); function functionCallWithValue(address target, bytes memory data, uint256 value) internal returns (bytes memory) { if (address(this).balance < value) {
 revert AddressInsufficientBalance(address(this));</pre> (bool success, bytes memory returndata) = target.call{value: value}(data); return verifyCallResultFromTarget(target, success, returndata); function functionStaticCall(address target, bytes memory data) internal view returns (bytes memory) { (bool success, bytes memory returndata) = target.staticcall(data); return verifyCallResultFromTarget(target, success, returndata); function functionDelegateCall(address target, bytes memory data) internal returns (bytes memory) { (bool success, bytes memory returndata) = target.delegatecall(data); return verifyCallResultFromTarget(target, success, returndata); function verifyCallResultFromTarget(address target, bool success, bytes memory returndata) internal view returns (bytes memory) { if (!success) { _revert(returndata); } else { if (returndata.length == 0 && target.code.length == 0) { revert AddressEmptyCode(target); } return returndata; } function verifyCallResult(bool success, bytes memory returndata) internal pure returns (bytes memory) { if (!success) { _revert(returndata); } else { return returndata; } function _revert(bytes memory returndata) private pure { if (returndata.length > 0) { assembly { let returndata_size := mload(returndata) revert(add(32, returndata), returndata_size) } } else { revert FailedInnerCall(); } } }



Instruction: [INST]Generate smart contract code that meets the following requirements: Programming with solidity ^ version number 0.8.20 Implement a library named Address for handling address related operations, throwing exceptions when the account balance is insufficient, code at the target address, or internal calls fail The following functions need to be implemented, with the following names and functions: sendValue: Sends Ether to the specified address functionCall: Calls a function at the target address without passing Ether. functionCallWithValue: Calls a function at the target address and passes the specified amount of Ether. functionStaticCall: Call a function at the target address statically. functionDelegateCall: Delegate a call to the target address. verifyCallResultFromTarget: Verifies that the call result was successful and handles exceptions. verifyCallResult: Verifies that the call result was successful and handles exceptions. revert: Rolls back the transaction and throws an exception. Please give the contract code[/INST] Output code: pragma solidity ^0.8.20; library Address { function sendValue(address payable target, uint256 amount) public { require(target.balance >= amount, "Insufficient balance"); (bool success,) = target.call{value: amount}(""); require(success, "Failed to send value"); function functionCall(address target, bytes memory data)[public] { require(target.code.length > 0, "Target address has no code") (bool success, bytes memory result) = target.call(data); require(success, "Failed to call function");); function functionCallWithValue(address target, uint256 amount, bytes memory data) public { require(target.balance >= amount, "Insufficient balance"); (bool success, bytes memory result) = target.call{value: amount}(data); require(success, "Failed to call function with value"); function functionStaticCall(address target, bytes memory data) public view { require(target.code.length > 0, "Target address has no code"); (bool success, bytes memory result) = target.staticcall(data); require(success, "Failed to call function statically"); function functionDelegateCall(address target, bytes memory data) public { require(target.code.length > 0, "Target (bool success, bytes memory result) = target.delegatecall(data); require(success, "Failed to delegate call"); function verifyCallResultFromTarget(address target, bytes memory data) public view { require(target.code.length > 0, "Target address has no c (bool success, bytes memory result) = target.call(data); code"); require(success, "Failed to call function"); require(result.length > 0, "No result returned"); function verifyCallResult(bytes memory result) public pure {

Figure 4: CodeLlama generation results on the "Address" task.

ed");

require(result.length > 0, "No result retur

function _revert() public pure {
 revert("Transaction reverted");

}

```
Instrction:
[INST]Generate smart contract code that meets the following requirements:
Programming with solidity ^ version number 0.8.20
Implement a library named Address for handling address related operations, throwing exceptions when the account balance
is insufficient, code at the target address, or internal calls fail
The following functions need to be implemented, with the following names and functions:
sendValue: Sends Ether to the specified address.
functionCall: Calls a function at the target address without passing Ether.
functionCallWithValue: Calls a function at the target address and passes the specified amount of Ether.
functionStaticCall: Call a function at the target address statically.
functionDelegateCall: Delegate a call to the target address.
verifyCallResultFromTarget: Verifies that the call result was successful and handles exceptions. verifyCallResult: Verifies that the call result was successful and handles exceptions.
 revert: Rolls back the transaction and throws an exception.
Please give the contract code[/INST]
Output Code:
pragma solidity ^0.8.20;
library Address {
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount,
        require(address(this).balance >= amount, "Address: insufficient balance");
(bool success, ) = recipient.call{value: amount}("");
require(success, "Address: unable to send value, recipient may have reverted");
    function functionCallWithValue(address target, bytes memory data, uint256 value) internal returns (bytes memory) {
        return target.call{value: value}(data);
    function functionStaticCall(address target, bytes memory data) internal view returns (bytes memory) {
        return target.staticcall(data);
    function functionDelegateCall(address target, bytes memory data) internal returns (bytes memory) {
        (bool success, bytes memory returndata) = target.delegatecall(data);
require(success, "Address: delegate call failed");
        return returndata;
    function _revert(bytes memory errorMessage) internal pure {
        revert(string(errorMessage));
    }
    function _revertWithErrorMessage(bytes memory errorMessage) internal pure {
        revert(string(errorMessage));
    function verifyCallResultFromTarget(bytes memory returndata) internal returns (bool) {
        if (returndata.length == 0) {
             return false;
        }
        require(returndata[0] == 0x09, "Address: incorrect return value");
        return true:
    function verifyCallResult(bytes memory returndata) internal returns (bool) {
        if (returndata.length == 0) {
             return false:
        }
        require(returndata[0] == 0x09, "Address: incorrect return value");
        return true:
```

Figure 5: CodeBC-CI generation results on the "Address" task.

}

```
Instrction:
 [INST]Generate smart contract code that meets the following requirements:
Programming with solidity ^ version number 0.8.20
Implement a library named Address for handling address related operations, throwing exceptions when the account balance is insufficient, code at the
target address, or internal calls fail
The following functions need to be implemented, with the following names and functions:
The following functions need to be implemented, with the following names and functions:
sendValue: Sends Ether to the specified address.
functionCall: Calls a function at the target address without passing Ether.
functionCallWithValue: Calls a function at the target address and passes the specified amount of Ether.
functionStaticCall: Call a function at the target address saticCally.
functionDelegateCall: Delegate a call to the target address.
verifyCallResult: Verifies that the call result was successful and handles exceptions.
verifyCallResult: Verifies that the call result was successful and handles exceptions.
Please give the contract code[/INST]
 Output Code:
 pragma solidity ^0.8.20;
pragma solidity "w.o.co,
library Address {
  function sendValue(address payable recipient, uint256 amount) internal {
    require(address(this).balance >= amount, "Address: insufficient balance");
    (bool success, ) = recipient.call{value: amount}("");
    require(success, "Address: unable to send value, recipient may have reverted");
            function functionCall(address target, bytes memory data) internal returns (bytes memory) {
                    return functionCall(target, data, "Address: low-level
                                                                                                                                                           call failed");
            function functionCall(address target, bytes memory data, string memory errorMessage) internal returns (bytes memory) {
                    return functionCallWithValue(target, data, 0, errorMessage);
           function functionCallWithValue(address target, bytes memory data, uint256 value) internal returns (bytes memory) {
                    return functionCallWithValue(target, data, value, "Address: low-level call wit
            ,
function functionCallWithValue(address target, bytes memory data, uint256 value, string memory errorMessage) internal returns (bytes memory) {
                                                                                                                                                           fficient
ract");
                    require(address(this).balance >= value, "Address: insufficient balance for
require(isContract(target), "Address: call to non-contract");
(bool success, bytes memory returndata) = target.call{value: value}(data);
                                                                                                                                                                             t balance for call");
                     return _verifyCallResult(success, returndata, errorMessage);
           function functionStaticCall(address target, bytes memory data) internal view returns (bytes memory) {
                    return functionStaticCall(target, data, "Address: low-level
                                                                                                                                                                                              all failed"):
            function functionStaticCall(address target, bytes memory data, string memory errorMessage) internal view returns (bytes memory) {
                     require(isContract(target), "Address: static call to non-contract"
(bool success, bytes memory returndata) = target.staticcall(data);
                      return _verifyCallResult(success, returndata, errorMessage);
           function functionDelegateCall(address target, bytes memory data) internal returns (bytes memory) {
                     return functionDelegateCall(target, data,
            function functionDelegateCall(address target, bytes memory data, string memory errorMessage) internal returns (bytes memory) {
                     require(isContract(target), "Address: delegate call to non-contract"
(bool success, bytes memory returndata) = target.delegatecall(data);
return _verifyCallResult(success, returndata, errorMessage);
                                                                                                                                                                                              );
           function verifyCallResultFromTarget(address target, bytes memory returndata, string memory errorMessage) internal view returns (bytes memory) {
    (bool success, bytes memory returndataCopy) = target.staticcall(returndata);
    return _verifyCallResult(success, returndataCopy, errorMessage);
          function _verifyCallResult(bool success, bytes memory returndata, string memory errorMessage) private pure returns(bytes memory) {
    if (success) {
        return returndata;
    }
}
                    } else {
    if (returndata.length > 0) {
        returndata.length > 0) {
        returndata.
                                        assembly {
    let returndata_size := mload(returndata)
    revert(add(32, returndata), returndata_size)
                                        3
                              } else {
    revert(errorMessage);
                             }
                   }
            function isContract(address account) internal view returns (bool) {
                    uint256 size;
assembly { size := extcodesize(account) }
return size > 0;
           ,
function _revert() internal {
    assembly {
        revert(0, 0)
    }
                   }
         }
}
```

Figure 6: CodeBC-VD generation results on the "Address" task.

Instrction: [INST]Generate smart contract code that meets the following requirements:

```
Programming with solidity ^ version number 0.8.20
Implement a library named Address for handling address related operations, throwing exceptions when the account balance is
insufficient, code at the target address, or internal calls fail
The following functions need to be implemented, with the following names and functions:
sendValue: Sends Ether to the specified address.
functionCall: Calls a function at the target address without passing Ether.
functionCallWithValue: Calls a function at the target address and passes the specified amount of Ether.
functionStaticCall: Call a function at the target address statically.
functionDelegateCall: Delegate a call to the target address.
verifyCallResultFromTarget: Verifies that the call result was successful and handles exceptions.
verifyCallResult: Verifies that the call result was successful and handles exceptions.
_revert: Rolls back the transaction and throws an exception.
Please give the contract code[/INST]
[Tag]correct[/Tag]
Output Code:
pragma solidity ^0.8.20;
library Address {
      function isContract(address account) internal view returns (bool) {
          uint256 size:
          assembly { size := extcodesize(account) }
          return size > 0;
      function sendValue(address payable recipient, uint256 amount) internal {
          require(address(this).balance >= amount, "Address: insufficient balance");
(bool success, ) = recipient.call{value: amount}("");
require(success, "Address: unable to send value, recipient may have reverted");
      ,
function functionCall(address target, bytes memory data) internal returns (bytes memory) {
          return functionCall(target, data, "Address: low-level call failed");
      function functionCall(address target, bytes memory data, string memory errorMessage) internal returns (bytes memory) {
          return functionCallWithValue(target, data, 0, errorMessage);
      function functionCallWithValue(address target, bytes memory data, uint256 value) internal returns (bytes memory) \{
          return functionCallWithValue(target, data, value, "Address: low-level call with value failed");
     function functionCallWithValue(address target, bytes memory data, uint256 value, string memory errorMessage) internal returns
(bytes memory) {
          require(address(this).balance >= value, "Address: insufficient balance for call");
          require(isContract(target), "Address: call to non-contract");
(bool success, bytes memory returndata) = target.call{value: value}(data);
           return _verifyCallResult(success, returndata, errorMessage);
     function functionStaticCall(address target, bytes memory data) internal view returns (bytes memory) {
    return functionStaticCall(target, data, "Address: low-level static call failed");
     function functionStaticCall(address target, bytes memory data, string memory errorMessage) internal view returns (bytes memory) {
          require(isContract(target), "Address: static call to non-contract"
(bool success, bytes memory returndata) = target.staticcall(data);
                                                                                                );
           return verifyCallResult(success, returndata, errorMessage);
      function functionDelegateCall(address target, bytes memory data) internal returns (bytes memory) \{
          return functionDelegateCall(target, data, "Address: low-level delegate call failed");
     function functionDelegateCall(address target, bytes memory data, string memory errorMessage) internal returns (bytes memory) {
          require(isContract(target), "Address: delegate call to non-contract"
(bool success, bytes memory returndata) = target.delegatecall(data);
return _verifyCallResult(success, returndata, errorMessage);
                                                                                                   );
     function verifyCallResultFromTarget(address target, bytes memory returndata, string memory errorMessage) internal view returns
(bytes memory) {
          (bool success, bytes memory returndataCopy) = target.staticcall(returndata);
           return _verifyCallResult(success, returndataCopy, errorMessage);
     function _verifyCallResult(bool success, bytes memory returndata, string memory errorMessage) private pure returns(bytes memory)
{
          if (success) {
                return returndata;
          } else {
                if (returndata.length > 0) {
                    assembly {
                         let returndata_size := mload(returndata)
revert(add(32, returndata), returndata_size)
             revert(errorMessage);
}
               } else {
         }
    }
}
```

Figure 7: CodeBC generation results on the "Address" task.