Supplementary Material

Thanks for reviewers' suggestions, we provided the following to support our research. Due to the limitation of response period, we will add following contents to our paper before final submission.

1. Supplemental Experiment

We have conducted a comparative analysis between SolaSim and current text-based multilingual code clone tools, namely CCFinderSW[1] and MSCCD[2].

Experiment setup:

This experiment is proposed to compare the performance of SolaSim with existing tools. All evaluations are based on the ground truth test dataset, which contains 374 smart contract pairs (184 True, 190 False). The criteria for ground truth labeling are presented in the next section.

Since SolaSim is the first code clone tool for Solana smart contracts, we compared these tools on contract-level similarity. The accuracy is computed by (TP+TN)/(TP+TN+FP+FN) [3], which denotes the ratio of correct prediction in total contract pairs. Meanwhile, we tried various thresholds to find the best performances of these tools. If the contract-level similarity is larger than the threshold, then the tool recognizes the sample as true(related). Otherwise, it is recognized as false(not related).

Experiment results and analysis:

As depicted in Figure 1, SolaSim demonstrated superior performance compared with two baseline tools.

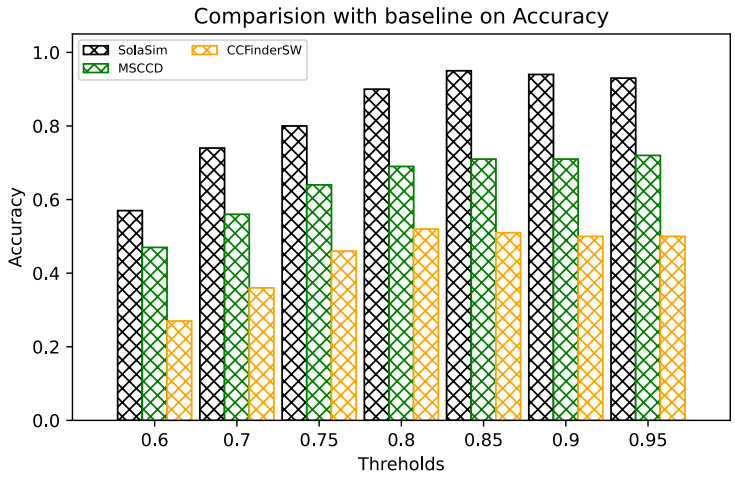

Figure 1

Regarding the performance of CCFinderSW:

First and foremost, CCFinderSW is limited to detecting Type 1 and Type 2 code clones, which hinders its effectiveness on Type 3 clones. Nevertheless, it still presents some effectiveness, given the prevalence of copy-and-paste operations within the Solana smart contract ecosystem.

Second, CCFinderSW is mainly a diff tool rather than a contract-level similarity comparison tool. Consequently, our statistical similarity computation of Solana smart contracts is based on simple code fragments directly. However, assessing the similarity between smart contracts necessitates considering dependencies among multiple files/functions. Such textual code clone tools are incapable of capturing control dependencies existing in the CFG within the MIR, resulting in inferior performance.

In addition, CCFinderSW requires specific parsers as input to parse the program. Although we utilized the provided Rust parser, there is a lack of well-defined domain-specific parsers tailored for Solana smart contracts, contributing to its suboptimal performance.

Regarding the performance of MSCCD:
MSCCD showed relatively better performance compared to CCFinderSW, owing to its capability for file-level similarity comparison and support for Type-3 level comparisons.
Nevertheless, it still performs badly behind SolaSim. We analyzed such performance, and it may attributed to the following reasons:
1. Smart contract programs typically include multiple files, and a single-file comparison approach cannot effectively capture the dependencies between program files.
2. MSCCD is a multi-language code clone detection tool that requires a Rust tokenizer to compare token bags. However, since we are the first to focus on code cloning for Solana smart contracts, there is currently no Rust tokenizer specifically tailored for Solana smart contracts.
3. The current similarity calculation in MSCCD is not geared towards contract-level similarity. Thus, such a comparison has resulted in information gaps.

2. Ground Truth Test Dataset

To better elucidate the rationality and consistency underlying our dataset construction. We employ the following criteria to assess the test dataset and mitigate biases. In the meantime, we provided a set of detailed rules to asses the similarity of the smart contract pairs.

(Sampling Bias)

To mitigate potential sampling bias, we carefully choose the sampling size and ratio for the dataset. Spe cifically, by random sampling 374 contract pairs from 3740 x 3740 contract pairs, we achieved a confidence level of 95% and a confidence interval of 5.07, which is considered sufficient in previous studies [4].

(Subjective Bias)

Furthermore, to mitigate potential subjective biases in the labeling process, we implemented a double-check procedure. Specifically, four authors independently labeled the contract pairs, with the help of the fifth author to resolve any disagreements. The agreement between authors, measured by Kappa value, is 0.78, demonstrating a substantial agreement. These authors possess over three years of experience in smart contract research, which could reduce potential labeling errors.

(Rules for Labeling)

We established the ground truth through random sampling of pairs of smart contracts and instruction-level functions from the entire dataset. The dataset was labeled according to the following criteria:

1. For instruction-level function assessment:

Rule 1: If the function code pairs are strictly identical, they are categorized as true (Type 1 clones); otherwise, they are labeled as false.

Rule 2: If the function code pairs are strictly identical except for variable names, they are categorized as true (Type 2 clones); otherwise, they are labeled as false.

Rule 3:

a) Whether the two function code pairs implement the same functionality, such as "deposit" and "withdraw."

b) Whether the two function code pairs merely reorder logic sequences without altering the logical structure.

c) Whether the two function code pairs introduce additional code statements that do not affect the implementation of other code.

d) Whether the two function code pairs delete code statements that do not affect the implementation of other code.

If the function code pairs fulfill the above sub-rules of Type-3 clones, then they are labeled as true; otherwise, they are labeled as false. In addition, we provided an example of Type 3 code clones in Figure 6 of the original paper.

2. For smart contract-level assessment:

We first analyze the business purpose of pairs of smart contracts. If two smart contracts served different business purposes, they were directly labeled as false (not related). Otherwise, we scrutinized the functions within each smart contract. We allowed for some differences in functions between pairs of smart contracts, as long as approximately 70% of the functions were similar. The criteria for assessing function pairs remained consistent with the aforementioned rules. Therefore, if the above requirements were met, we labeled the pair of smart contracts as true (related); otherwise, they were considered unrelated.

(Motivation Example)

We presented a code snippet pair, which is false negative example of textual code clone tools, to illustrate the importance of MIR and the criteria of labeling rules.

```
1 pub fn withdraw (
2 ... ) -> Result<Instruction, ProgramError> {
4 let data = Instruction::Withdraw(instruction).pack();

5 let mut accounts = vec![
6  AccountMeta::new_readonly(*user_pubkey, false),
7  AccountMeta::new_readonly(*authority_pubkey, false),
8  AccountMeta::new_readonly(*user_transfer_authority_pubkey
  true),
9 ...];

10 if let Some(host_fee_pubkey) = host_fee_pubkey {
    accounts.push(AccountMeta::new(*host_fee_pubkey, false));}

11 Ok(Instruction{program_id:*program_id, accounts, data})}
```

Figure 2

```
1 pub fn deposit (
2 ... ) -> Result<Instruction, ProgramError> {
4 let data = Instruction::Deposit(instruction).pack();

5 let mut accounts = vec![
6   AccountMeta::new_readonly(*user_pubkey, false),
7   AccountMeta::new_readonly(*authority_pubkey, false),
8   AccountMeta::new_readonly(*user_transfer_authority_pubkey
    true),
9  ...];

10 //different parts

11 Ok(Instruction{program_id:*program_id, accounts, data})}
```

Figure 3

As we showed above, these two code snippets are instruction-level functions that implement different business logic. Figure 2 is a withdrawal instruction and Figure 3 is a deposit. Both of them are only different in line 10, so it has been recognized as a Type 3 code clone in textual code clone tools. However, as we listed above, if two functions implement different functionalities, it would be labeled as false. Thus, this example is a false negative. Nevertheless, SolaSim can detect these two instruction functions as different, because of the MIR-based CFG, which captures the dependency of functions. As shown in Line 11 of Figure 2 and Figure 3, the main business logic is contained in Instructions called functions based on input program id and data. And such essential information would be included in MIR based CFG.

References:

[1] Semura, Y., Yoshida, N., Choi, E., & Inoue, K. (2017, December). CCFinderSW: Clone detection tool with flexible multilingual tokenization. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)* (pp. 654-659). IEEE.

[2] Zhu, W., Yoshida, N., Kamiya, T., Choi, E., & Takada, H. (2022, May). MSCCD: grammar pluggable clone detection based on ANTLR parser generation. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (pp. 460-470).

[3] Accuracy https://developers.google.com/machine-learning/crash-course/classification/accuracy

[4] Confidence based Sample Size Computation https://www.surveysystem.com/sscalc.htm