Clone Detection for Smart Contracts: How Far Are We?

ZUOBIN WANG, Zhejiang University, China
ZHIYUAN WAN*†, Zhejiang University, China
YUJING CHEN, Zhejiang University, China
YUN ZHANG, Hangzhou City University, China
DAVID LO, Singapore Management University, Singapore
DIFAN XIE, Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China
XIAOHU YANG, Zhejiang University, China

In smart contract development, practitioners frequently reuse code to reduce development effort and avoid reinventing the wheel. This reused code, whether identical or similar to its original source, is referred to as a code clone. Unintentional code cloning can propagate flaws and vulnerabilities, potentially undermining the reliability and maintainability of software systems. Previous studies have identified a significant prevalence of code clones in Solidity smart contracts on the Ethereum blockchain. To mitigate the risks posed by code clones, clone detection has emerged as an active field of research and practice in software engineering. Recent studies have extended existing techniques or proposed novel techniques tailored to the unique syntactic and semantic features of Solidity. Nonetheless, the evaluations of existing techniques, whether conducted by their original authors or independent researchers, involve codebases in various programming languages and utilize different versions of the corresponding tools. The resulting inconsistency makes direct comparisons of the evaluation results impractical, and hinders the ability to derive meaningful conclusions across the evaluations. There remains a lack of clarity regarding the effectiveness of these techniques in detecting smart contract clones, and whether it is feasible to combine different techniques to achieve scalable yet accurate detection of code clones in smart contracts. To address this gap, we conduct a comprehensive empirical study that evaluates the effectiveness and scalability of five representative clone detection techniques on 33,073 verified Solidity smart contracts, along with a benchmark we curate, in which we manually label 72,010 pairs of Solidity smart contracts with clone tags. Moreover, we explore the potential of combining different techniques to achieve optimal performance of code clone detection for smart contracts, and propose SourceRECLone, a framework designed for the refined integration of different techniques, which achieves a 36.9% improvement in F1 score compared to a straightforward combination of the state of the art. Based on our findings, we discuss implications, provide recommendations for practitioners, and outline directions for future research.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Software evolution**; **Reusability**; • **General and reference → Empirical studies**.

Additional Key Words and Phrases: Smart Contract, Code Clone, Clone Detection, Ethereum, Blockchain

*Zhiyuan Wan is the corresponding author.
†Also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

Authors' Contact Information: Zuobin Wang, wangzuobin@zju.edu.cn, Zhejiang University, The State Key Laboratory of Blockchain and Data Security, Hangzhou, China; Zhiyuan Wan, wanzhiyuan@zju.edu.cn, Zhejiang University, The State Key Laboratory of Blockchain and Data Security, Hangzhou, China; Yujing Chen, chenyujing@zju.edu.cn, Zhejiang University, The State Key Laboratory of Blockchain and Data Security, Hangzhou, China; Yun Zhang, yunzhang@hzcu.edu.cn, Hangzhou City University, Hangzhou, China; David Lo, davidlo@smu.edu.sg, Singapore Management University, Singapore, Singapore; Difan Xie, xiedifan@bcds.org.cn, Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, China; Xiaohu Yang, yangxh@zju.edu.cn, Zhejiang University, The State Key Laboratory of Blockchain and Data Security, Hangzhou, China.

ACM 2994-970X/2025/7-ARTFSE057
https://doi.org/10.1145/3715776

ACM Reference Format:
Zuobin Wang, Zhiyuan Wan, Yujing Chen, Yun Zhang, David Lo, Difan Xie, and Xiaohu Yang. 2025. Clone Detection for Smart Contracts: How Far Are We?. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE057 (July 2025), 21 pages. https://doi.org/10.1145/3715776

1 Introduction

Blockchain is a distributed ledger that offers an open, decentralized, and fault-tolerant transaction mechanism. Blockchain technology has evolved to support general-purpose computations and a wide range of decentralized applications. The *smart contract* technology is one notable decentralized application, which allows computations to be executed on top of a blockchain. A smart contract is an executable code that runs on a blockchain to enforce the terms of an agreement between untrusted parties. Blockchain technology ensures that a smart contract is immutable, and transactions initiated by the contract are autonomously and truthfully executed. Several blockchain platforms support smart contracts [42], such as Ethereum [9], BNB Smart Chain [6], and Polygon [7], with Ethereum being the most prominent platform [3].

Practitioners tend to frequently reuse code in smart contract development to avoid reinventing the wheel and reduce development efforts [43], typically by copying and pasting existing code fragments into new projects. The reused code, which is identical or similar to the original source, is known as *code clone* [36]. While code cloning can have benefits such as accelerating the learning curve for APIs and providing rapid bug workarounds [25], unintentional cloning can propagate flaws and vulnerabilities, compromising the reliability and maintainability of codebases [18]. For instance, as reported in February 2022, two critical vulnerabilities in the smart contracts provided by Multichain (previously AnySwap) [33] affected nearly 8,000 user addresses, resulting in the exploitation of over 3 million dollars [5]. Previous studies have reported high code clone ratios in smart contracts on Ethereum, with 79.2% at the contract level [28] and 30.13% at the function level [27], indicating the potential risks associated with code clones in the blockchain ecosystem.

To mitigate the risks posed by code clones, clone detection has been an active field of research and practice in software engineering [26]. Over the past two decades, numerous techniques and tools have been developed for detecting code clones in traditional programming languages like C/C++ and Java [27], including NiCad [14], SourcererCC [37] and Deckard [24]. To support clone detection in the programming languages used for developing smart contracts, such as Solidity [15], recent studies either extend existing tools [27, 28] or propose novel techniques [19, 31] to accommodate the unique syntactic (e.g., the tradeoff between storage and execution) and semantic features (e.g., the ERC20 token[1]) of Solidity. Nevertheless, the evaluations of existing techniques, whether performed by their original authors or independent researchers [17, 35, 40, 41, 46], involve codebases written in diverse programming languages and employ different versions of the corresponding tools. The inconsistency renders direct comparisons of their evaluation results impractical and limits the ability to draw meaningful conclusions across these evaluations. There remains a lack of clarity regarding the effectiveness of these clone detection techniques for Solidity, and whether it is feasible to combine different techniques to achieve effective yet scalable code clone detection for Solidity smart contracts.

To address this gap, we conduct a comprehensive empirical study to evaluate the capability of clone detection techniques for Solidity smart contracts, and explore the combination of different techniques to achieve optimal code clone detection for smart contracts in the blockchain ecosystem. Specifically, we identify clone detection techniques through a literature search, applying inclusion criteria such as tool availability, open-source code accessibility, and customizability. The process

[1] https://eips.ethereum.org/EIPS/eip-20

leads to the selection of five representative code clone techniques for Solidity smart contracts, namely, SourcererCC [37], EClone [30], SmartEmbed [19], Deckard [28], and NiCad [27]. First, we compare the clone ratios reported by the selected techniques at various levels of granularity levels, utilizing a widely used dataset [28] that includes 33,073 verified Solidity smart contracts on Ethereum. Additionally, we measure the recall and precision of these techniques using a benchmark we curate, and evaluate their scalability across different input sizes. The benchmark comprises 72,010 manually labeled pairs of smart contracts randomly sampled from the verified contracts. Second, we examine the level of agreement among the selected techniques concerning clone labels and similarity scores for smart contract pairs. Third, we explore the synergistic combination of these techniques to achieve optimal performance in clone detection for smart contracts. We investigated the following research questions:

RQ1. How effective are existing techniques on clone detection for smart contracts?

The techniques report comparable clone ratios at the contract level, and demonstrate a consistent increase in clone ratios as the granularity level increases, suggesting that function-level code reuse is more prevalent on Ethereum compared to code reuse at the contract or subcontract level. All techniques achieve almost 100% recall for Type-1 and Type-2 clones. SourcererCC and EClone both achieve recall exceeding 99% for Type-3 clones, with EClone leading at an 88.2% recall for Type-4 clones. Precision across techniques is generally above 90%, except for 29.6% of EClone. In terms of scalability, SourcererCC demonstrates the shortest execution times among these techniques, and scales efficiently with inputs up to 1 million lines of code. In contrast, EClone performs well only with inputs of up to 1,000 lines of code, showing limited scalability for large datasets.

RQ2. Do different techniques complement each other?

EClone detects 89.5% of clone pairs in our dataset, yet generates numerous false positives, demonstrating poor agreement with the four other techniques. In contrast, SourcererCC exhibits substantial overlap with SmartEmbed, Deckard, and NiCad in their detected clones, thus complementing EClone by increasing true positives in clone detection. Furthermore, the poor to moderate agreement of EClone with other techniques on similarity scores of code pairs highlights its unique ability to detect code clones with semantic similarities.

RQ3. Can we go beyond the state of the art by combining existing techniques?

The combination of SourcererCC and EClone outperforms any individual technique in F1 score, and also outperforms other combinations of techniques with respect to execution overhead. We further propose SourceREClone, a novel framework that refines the integration of **SourceR**erCC and **EClone**, which consists of two phases: (1) parameter optimization, where EClone thresholds are refined with respect to SourcererCC similarity scores, and (2) clone detection, where both techniques are employed to achieve effective code clone detection in smart contracts. The evaluation results indicate that our framework surpasses the straightforward combination of SourcererCC and EClone, achieving a significant increase in precision (from 29.5% to 46.4%) while experiencing only a slight reduction in recall (from 89.1% to 87.7%), resulting in a 36.9% improvement in F1 score.

Based on the findings, we discuss implications and provide practical recommendations for leveraging existing clone detection techniques to advance beyond the current state of the art. In addition, we outline several research avenues such as improving the precision and scalability of Type-4 clone detection, as well as expanding support for bytecode-level clone detection in smart contracts.

This paper makes the following contributions:

- We conduct an empirical study investigating the effectiveness of five clone detection techniques in detecting smart contract clones across consistent datasets, aiming at providing a comprehensive understanding of the capabilities of existing techniques.

- We propose SourceREClone, a novel framework that integrates different techniques for clone detection in smart contracts, resulting in a 36.9% improvement in F1 score over the straightforward combination of the state of the art.
- We construct a benchmark consisting of 72,010 manually labeled pairs of Solidity smart contracts, with 48 as Type-1, 664 as Type-2, 1,167 as Type-3, 13,841 as Type-4 clones, and the remaining as non-clones. The benchmark can facilitate the evaluation of clone detection techniques for smart contracts in the future by others.
- We provide practical recommendations for practitioners and outline future avenues of research.

2 Background

2.1 Smart Contracts

A smart contract is a software program that can be deployed and run on blockchains. Ethereum is one of the most popular blockchains, with objects called *accounts* to represent its state [9]. Each account has a 20-byte address, and can be categorized into two types: (1) externally owned accounts, controlled by private keys, and (2) contract accounts, controlled by their contract code deployed.

Solidity [2] is designed for developing smart contracts on blockchains compatible with the Ethereum Virtual Machine (EVM), which serves as the execution environment of smart contracts and works in a stack-based architecture [47]. Specifically, Solidity smart contracts are compiled into bytecode, from which EVM iteratively fetches an instruction and operates on data and resources.

As code blocks in Solidity smart contracts, subcontracts can be categorized into four types [1, 38]: (1) *interface*, which defines protocols with no implementations for functions; (2) *abstract contract*, which defines fundamental structures of protocols with some functions implemented; (3) *contract*, which is complete and executable code; and (4) *library*, which provides reusable implementations of common operations without storage. Note that functions in smart contracts can be invoked by other smart contracts or automatically run.

2.2 Classification of Code Clones

Code clones refer to identical or "very similar" code fragments, which can be classified into four types [36]:

- **Type-1 clones.** Exact code copies differ only in whitespace or comments.
- **Type-2 clones.** Syntactically identical codes with different identifiers or literals.
- **Type-3 clones.** Include Type-2 clones, but allow code fragments to differ in complete lines of code, thereby capturing clones with entire lines added or removed.
- **Type-4 clones.** Semantically identical codes, but may have different syntactic implementations.

2.3 Literature Search and Scope

Given the widespread use of Solidity for developing smart contracts on Ethereum and the support of Solidity across multiple blockchain platforms, our study primarily concentrates on clone detection techniques for Solidity smart contracts. To identify relevant literature for our evaluation on clone detection techniques, we employed the keywords "smart contract" and "clone", and searched for related work published between 2014 and 2024 across seven academic databases, i.e., IEEE Explore, ACM Digital Library, Google Scholar, Springer Link, Web of Science, DBLP Bibliography, and EI Compendex. Our initial search yielded a total of 191 papers (see the replication package for details). We then reviewed the abstracts of these papers to exclude those that do not propose any clone detection techniques for smart contracts, including (1) 145 papers that only discuss smart contracts

or code clones in the background (e.g., [16, 20, 23]), (2) 24 papers that conduct studies relevant to code clones (e.g., [11, 21, 38]), and (3) 8 papers that do not support the clone detection for Solidity smart contracts (e.g., [8, 22, 44]). Ultimately, we identified 14 papers that propose clone detection techniques for smart contracts (see Appendix A in the replication package for details).

3 Study Design

3.1 Tool Selection

Given the considerable engineering efforts required to evaluate all the identified techniques, we selected five clone detection techniques for smart contracts from our literature search results, based on the inclusion criteria of (i) tool availability, (ii) open-source code accessibility, and (iii) customizability, as summarized in Table 1. The selected techniques vary in how they represent code (i.e., as text, tokens, trees, or graphs), and in their application of deep learning models. The following briefly introduces each of the selected techniques.

Table 1. Summary of Selected Clone Detection Techniques for Solidity Smart Contracts.

Tool Name	Year	Code Representation				Learning Based	Clone Granularity			Bytecode as Input	Open-Sourced URL
		Text	*Token*	*Tree*	*Graph*		*Contract*	*Subcontract*	*Function*		
SourcererCC [37]	2016		✓				●	◐	◐		https://github.com/Mondego/SourcererCC
EClone [30]	2019				✓		●	○	○	✓	https://github.com/njaliu/ethereum-clone
SmartEmbed [19]	2020			✓		✓	●	◐	◐		https://github.com/beyondacm/SmartEmbed
Deckard [28]	2020			✓			●	●	●		https://github.com/skyhover/Deckard
NiCad [27]	2022	✓					●	●	●		https://github.com/eff-kay/solidity-nicad

● Inherent Support ◐ Support after Customization ○ No Support

NiCad [27] involves three main steps: (i) source code parsing, (ii) normalization, and (iii) comparison and clone clustering. In Stage (i), NiCad uses the TXL source transformation system [13] to extract code fragments of a given granularity, such as functions and whole source files, from the input source code base. Extracted fragments are converted to a standardized form by removing noise such as spacing, formatting and comments, and reformatting using pretty-printing, to expose Type-1 clones. In Stage (ii), the extracted code fragments are further normalized by renaming, standard parenthesization, or removal of declarations, to expose Type-2 clones. In Stage (iii), clones are identified through line-wise comparison of normalized code fragments, parameterized by a difference threshold to allow for the detection of Type-3 clones. Clone pairs are clustered using a threshold-sensitive transitive closure to expose clone classes.

SourcererCC [37] operates in two phases: (i) partial index creation, and (ii) clone detection. In Phase (i), it tokenizes code blocks parsed from source code files, and builds a partial inverted index that maps tokens to code blocks. In Phase (ii), it iterates through all code blocks, and retrieves candidate clone blocks by querying the index with sub-block tokens; it then applies a filtering heuristic to eliminate candidates based on the similarity of code blocks to reduce the number of code block comparisons. Since SourcererCC matches tokens and not sequences or structures in source code, it exhibits a high tolerance to minor modifications, making it capable of detecting Type-3 clones, including those where statements are swapped, added, and/or deleted.

Deckard [24] includes three main steps: (i) parsing, (ii) vector generation, and (iii) vector clustering and clone detection. In Step (i), Deckard translates source files into *parse trees* by using a parser that is automatically generated from a formal syntax grammar. In Step (ii), Deckard processes the parse trees to produce fixed-dimension vectors that capture structural information within parse trees, In Step (iii), Deckard clusters the produced vectors w.r.t. their *Euclidean distances*, and generates clone reports using heuristics.

EClone [31] analyzes the semantic similarity between pairs of smart contracts by processing their EVM bytecode through a three-step workflow: (i) control flow graph generation, (ii) *birthmark*

vector creation, and (iii) comparison and clone detection. In Step (i), EClone generates a static control flow graph (CFG) based on the input bytecode of each contract. In Step (ii), EClone symbolically executes the CFG in the Ethereum runtime, and creates *birthmark* vectors for the basic blocks in each contract, which capture both syntactic and semantic features of the contract. In Step (iii), EClone estimates the similarity between the pair of contracts based on basic block-level semantic clones, and considers a similarity threshold ϕ for clone detection.

SmartEmbed [19] leverages deep learning techniques to automatically encode lexical, syntactical, and even semantic information of smart contracts into numerical vectors, which could be used to support clone detection by measuring the similarity between these vectors. Specifically, SmartEmbed first parses the source code to generate ASTs for smart contracts, and converts ASTs into token streams. The token streams are then normalized by removing semantically irrelevant information, such as stop words, punctuation, and the values of constants or literals. Using the normalized token streams, the code representation learning models in SmartEmbed convert code fragments of a given granularity (e.g., functions or contracts) into fixed-length dimension vectors that capture both the syntax and semantics of the code. To support clone detection, SmartEmbed measures the similarity between the vectors of two code fragments, and identifies them as a clone pair if the similarity score exceeds the threshold δ.

3.2 Dataset

We selected a dataset widely used in previous studies on code reuse of smart contracts [27, 28] as our evaluation benchmark (**Dataset I**), which consists of 33,073 smart contracts, amounting to 8,671,545 lines of code. The 33,073 smart contracts were collected in July 2018 from Etherscan[2] with their source code in Solidity available on and verified by Etherscan. Specifically, each verified smart contract in the dataset publishes its flattened version on Etherscan, which is referred to as its *code file*. Given the inclusion of verified smart contracts deployed to Ethereum in the dataset, we consider the dataset to be representative of smart contract code in production. To mitigate the potential cost of computation-intensive evaluation, we further curated **Dataset II** by randomly sampling 380 smart contracts from **Dataset I**, at 95% confidence level and with 5% error margin. In addition, to facilitate the evaluation of code clones of different granularity levels, we used Solidity ANTLR4 grammar parser[3] to extract subcontracts from each contract, as well as functions from each subcontract. The details of our datasets are summarized in Table 2.

Table 2. Dataset Statistics.

	Dataset I	**Dataset II**
Number of Contracts	33,073	380
Number of Subcontracts	168,359	2,050
Number of Functions	847,551	11,625
Number of Lines of Code (LOC)	8,671,545	123,614

3.3 Tool Execution

3.3.1 Tool Configuration and Execution. We used the default settings of the selected tools and set the thresholds as suggested by previous studies [19, 27, 28, 30, 37], in particular, 0.70 for SourcererCC, 0.84 for EClone, 0.95 for SmartEmbed, 0.79 for Deckard, and 0.70 for NiCad. All experiments were

[2] https://etherscan.io/
[3] https://github.com/OpenZeppelin/sgp

executed on a server equipped with two Intel(R) Xeon(R) Platinum 8358P CPUs (64 cores and 128 threads) running at 2.60 GHz , 755 GB memory, and operating on Ubuntu 20.04 LTS.

3.3.2 Output Processsing. Due to the overlapping definitions of clone types, some clone instances may conform to multiple types. For example, if two code fragments are identical, they will also remain identical after blind renaming, making Type-2 clone instances obtained by blind renaming potentially contain fragments that belong to Type-1 clones. Therefore, we excluded every Type-1 clone instance from the Type-2 category. Type-3 and Type-4 clone instances differ in their levels of syntactic similarity [46]. To differentiate Type-3 and Type-4 clone instances, we measured the syntactic similarity between two code fragments by calculating the minimum ratio of common lines between them after eliminating Type-1 formatting differences and applying Type-2 normalization. Common lines are obtained by a *diff* algorithm that takes into account the order of the lines. We used a syntactic similarity cutoff of 0.5 to differentiate between Type-3 and Type-4 clones by referring to BigCloneBench [39].

3.4 Research Questions

We seek to answer the following research questions:

RQ1. How effective are existing techniques on clone detection for smart contracts?

We first systematically investigated the effectiveness of each selected technique for code clone detection of smart contracts, in terms of detected clone ratios, precision and recall of detected code clones, and scalability.

In terms of clone ratios, we ran the clone detection techniques on Dataset I. Based on the output of the clone detection techniques, we first identified the *duplicate* code fragments flagged as clones of some other code fragments in our datasets. We calculated the clone ratio as the percentage of the total number of *duplicate* code fragments over the total number of code fragments of a given granularity level, i.e., entire contracts, sub-contracts, and functions.

As for the measurement of precision and recall, we first constructed the ground truth for Dataset II. Two of the authors independently labeled each smart contract in Dataset II with an initial classification of clone types. Specifically, the labeling process began with identifying Type-1 clones by performing a normalization process on the smart contracts. The normalization involves removing comments and applying strict pretty-printing to standardize code formatting. Contracts that were identical after the normalization process were labeled as Type-1 clones. For Type-2 clone labeling, further transformations were applied: all identifiers were renamed to a generic placeholder (e.g., “X”), and literal values were replaced with standardized defaults (e.g., numerical values set to 0, strings to “default”). Contracts that remained identical following these transformations were designated as Type-2 clones. Each group of Type-1 and Type-2 clones was verified by two labelers, with only one contract from each group retained for subsequent steps. For the labeling of Type-3 and Type-4 clones, two labelers independently examined the semantics of the remaining smart contracts and grouped them based on semantic similarity Type-3 and Type-4 clones differ in their levels of syntactic similarity [46]. To differentiate between the two, we applied a syntactic similarity threshold of 0.5, by referring to BigCloneBench [39]. Specifically, for each pair of contracts demonstrating semantic similarity, we evaluated syntactic similarity by calculating the ratio of common lines between them. We further used Cohen’s Kappa [12] to examine the agreement between the two labelers, resulting in a Kappa value of 0.73, which indicates substantial agreement between the two labelers. For any disagreements, the two labelers, along with a moderator, reviewed and discussed the cases to reach a consensus. Finally, following the initial labeling of clone types for individual smart contracts, we labeled each pair of smart contracts in Dataset II.

Subsequently, we ran the clone detection techniques on Dataset II, and classified their outputs into true positive (TP), true negative (TN), false positive (FP), and false negative (FN), based on the ground truth of Dataset II. The precision, recall, and F1 score are calculated as shown in Equation 1, 2, and 3:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (1) \qquad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (2) \qquad \text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (3)$$

To evaluate the scalability of the selected techniques, we measured the execution time of the techniques on the inputs of different orders of magnitude. Specifically, we randomly selected smart contracts as inputs from Dataset I of different orders of magnitude in the number of *lines of code* (LOC), from 1K LOC to 1M LOC (1K, 10K, 100K and 1M). In addition, we limited the techniques to 32GB to account for the memory usage of the machine.

RQ2. Do different techniques complement each other?

To investigate the complementary of existing clone detection techniques, we first computed Cohen's Kappa to evaluate the inter-technique agreement of code clones in Dataset II at the contract level. Cohen's Kappa [12] is a widely used statistical measure that calculates the degree of agreement among two labelers in categorical classifications over that expected by chance. Kappa values can vary between -1 and 1, where values below 0 indicate poor agreement, 0 no agreement beyond chance, between 0.01 and 0.20 slight agreement, 0.21 and 0.40 fair agreement, 0.41 and 0.60 moderate agreement, 0.61 and 0.80 substantial agreement, and any value above 0.81 perfect agreement.

We further used the intraclass correlation coefficient (ICC) [4] to measure the agreement on the similarity of clone pairs in Dataset II between the techniques. ICC [32] is widely used to measure the extent of agreement and consistency among two or more observers for numerical or quantitative observations, which represents the between-pair variance expressed as a proportion of the total variance of the observations. Specifically, ICC values were calculated using two-way mixed effects model in the `Pingouin` statistical package version 0.5.4[4]. ICC values can vary between 0 and 1.0, where values below 0.5 indicate poor agreement, between 0.5 and 0.75 moderate agreement, between 0.75 and 0.9 strong agreement, and any value above 0.9 indicates perfect agreement, based on the 95% confidence intervals [29].

RQ3. Can we go beyond the state of the art by combining existing techniques?

We first combined existing techniques to find the most reasonable combination to achieve optimal performance in clone detection for smart contracts. Specifically, we evaluated the recall, precision, and F1 score of multiple combinations of the existing techniques for clone detection in smart contracts of Dataset II. Subsequently, we developed a framework for a refined integration of existing techniques, aiming at further advancing the effectiveness of clone detection for smart contracts as compared to a straightforward combination of existing techniques, and surpassing the state of the art.

4 Results

4.1 Effectiveness of Existing Techniques (RQ1)

4.1.1 Clone Ratios. Table 3 provides a detailed summary of the clone ratios at the contract, sub-contract, and function levels in our datasets. We made the following observations:

- **Contract Level:** The clone ratios of Dataset I and II at the contract level range from 81.31% to 89.92%, and from 69.21% to 98.68% as reported by the techniques, respectively. In Dataset I, SourcererCC reports the highest clone ratio of 89.92% among the tools, while Deckard reports the lowest clone ratio of 81.31%, which is comparable with the clone ratio of 79.2% as reported in a prior study that used Deckard to measure clone ratio [28]. In Dataset II, EClone

[4]https://pingouin-stats.org/build/html/generated/pingouin.intraclass_corr.html

reports the highest clone ratio of 98.68%, which is over 20% higher than those reported by other techniques.
- **Subcontract Level:** The clone detection ratios of Dataset II at the subcontract level vary from 80.78% to 87.17%, as reported by the techniques. SourcererCC leads by reporting the highest clone ratio of 87.17%, closely followed by NiCad at 84.34%.
- **Function Level:** the clone ratios in Dataset II demonstrate considerable variation. SourcererCC and Deckard both report function-level clone ratios exceeding 90%, whereas NiCad records the lowest ratio at 73.10%.

As for the clone ratios of Dataset II, we observed a consistent increasing trend as the levels of granularity increase across the techniques except NiCad, indicating that function-level code reuse practice tends to be the most prevalent in Ethereum, as compared to contract- and subcontract-level code reuse.

Finding 1: The clone detection techniques for smart contracts report comparable clone ratios at the contract level, with the exception of EClone, which reports a 99% clone ratio in Dataset II. Among these techniques, SourcererCC reports the highest clone ratios across all granularity levels. Additionally, except for NiCad, all techniques exhibit a steady increase in clone ratios as the granularity level rises, indicating that function-level code reuse is more prevalent in Ethereum than contract- and subcontract-level reuse.

Table 3. Clone Ratios across Levels of Granularity.

	Contract Level (Dataset I / Dataset II)	Subcontract Level (Dataset II)	Function Level (Dataset II)
SourcererCC	89.92% / 78.16%	87.17%	90.76%
EClone	- / 98.68%	-	-
SmartEmbed	85.41% / 73.95%	83.71%	86.65%
Deckard	81.31% / 69.47%	80.78%	90.18%
NiCad	83.06% / 69.21%	84.34%	73.10%

4.1.2 Recall and Precision. Table 4 presents the recall and precision values of the selected techniques for contract-level clone detection on Dataset II. We made the following observations:

- **Recall:** SourcererCC demonstrates the highest recall in detecting Type-1 (100%), Type-2 (100%), and Type-3 (99.8%) code clones, highlighting its robust capability in detecting clones with minimal or moderate code modifications. Conversely, EClone exhibits the highest recall in detecting Type-4 code clones (88.2%), indicating its strength in identifying more complex, semantic-level clones.
- **Precision:** With the exception of EClone (29.6%), all techniques under investigation achieve a precision exceeding 90%, with Deckard achieving the pinnacle of precision at 99.7%, suggesting its reliability in minimizing false positives during clone detection.

Finding 2: All techniques exhibit perfect recall for Type-1 and Type-2 clones. SourcererCC and EClone both achieve a recall above 99% for Type-3 clones, while EClone achieves a recall of 88.2% for Type-4 clones, outperforming other techniques. All techniques demonstrate a precision above 90%, with the exception of EClone, which has a precision of 29.6%.

Table 4. Recall and Precision Values of Clone Detection Techniques (in %).

	Recall				Precision
	Type-1 (48 Pairs)	*Type-2 (664 Pairs)*	*Type-3 (1,167 Pairs)*	*Type-4 (13,841 Pairs)*	
SourcererCC	100.0	100.0	99.8	9.7	91.8
EClone	100.0	98.9	99.1	88.2	29.6
SmartEmbed	100.0	100.0	53.3	3.0	94.9
Deckard	100.0	100.0	47.0	2.3	99.7
NiCad	100.0	100.0	97.3	2.7	96.3

Table 5. Execution Time for Varying Input Sizes of Clone Detection Techniques.

	1K LOC	**10K LOC**	**100K LOC**	**1M LOC**
SourcererCC	12s	13s	13s	32s
EClone	20s	34m 31s	2d 15hr 24m 38s	-
SmartEmbed	9s	17s	1m 40s	44m 49s
Deckard	7s	11s	24s	2m 47s
NiCad	1s	2s	10s	3m 9s

4.1.3 Scalability. Table 5 presents the execution times of the selected techniques with inputs under different orders of magnitudes. We made the following observations:

- **SourcererCC** demonstrates comparable execution times within 1 minute across input sizes ranging from 1K LOC to 1M LOC, indicating its robust scalability to the largest input with reasonable execution time.
- **EClone** efficiently handles 1K LOC input, with execution times under 1 minute. However, it encounters scalability limits before 100K LOC, and fails to execute clone detection for the 1M LOC input.
- **SmartEmbed** demonstrates comparable execution times within 1 minute across input sizes ranging from 1K LOC to 10K LOC. Nonetheless, it fails to scale to 1M LOC, with execution times exceeding 30 minutes.
- **Deckard** exhibits comparable execution times within 1 minute across input sizes ranging from 1K LOC to 100K LOC, but its execution time increases to over 2 minutes for 1M LOC input.
- **NiCad** maintains execution times within 1 minute for input sizes ranging from 1K LOC to 100K LOC, but its execution time surpasses 3 minutes for 1M LOC input.

In comparison, SourcererCC demonstrates the shortest execution times and scales efficiently to inputs of at least 1M LOC. Deckard and NiCad exhibit execution times comparable to SourcererCC for inputs up to 100K LOC, but are over twice as slow for 1M LOC input. SmartEmbed performs comparably to SourcererCC for 1K and 10K LOC inputs, yet becomes significantly slower for 100K LOC and 1M LOC inputs. In contrast, EClone is only efficient for 1K LOC inputs, suggesting inadequate scalability for larger inputs.

> **Finding 3:** SourcererCC exhibits the shortest execution times among the clone detection techniques and scales efficiently with inputs of up to 1 million lines of code. In contrast, EClone performs efficiently only with inputs of 1,000 lines of code, indicating limited scalability for large inputs.

4.2 Agreement of Existing Techniques (RQ2)

4.2.1 Agreement on Code Clones. Figure 1 presents the agreement on the labels of smart contract pairs between the selected techniques. As suggested by the Kappa values, the selected techniques can generally be classified into two clusters: (i) graph-based techniques (i.e., EClone), and (ii) non-graph-based techniques (i.e., SourcererCC, SmartEmbed, Deckard and NiCad). The Kappa values between cross-cluster techniques (EClone vs. other techniques) are under 0.1, suggesting poor agreement on the clone labels of smart contract pairs between them. In contrast, the Kappa values between non-graph-based techniques range from 0.61 to 0.88, suggesting moderate to strong agreement on the clone labels of smart contract pairs between them.

We further conducted an in-depth analysis of the overlaps in code clone pairs detected by EClone, SourcererCC, and other selected clone detection techniques, taking into account both true positives and false positives, as depicted in Figure 3. We made the following observations:

- **EClone and SourcercerCC:** The combined use of EClone and SourcererCC covers the entirety of the detected clone pairs, as indicated by the "0" code pair at the bottom of the Venn diagrams in Figure 3a and Figure 3b.
- **EClone vs. non-graph-based techniques:** As illustrated in Figure 3a, EClone identifies the highest number of code clone pairs, detecting 14,066 pairs and accounting for 89.5% of the total clone pairs in Dataset II. Of these 14,066 code clone pairs, as detected by EClone, non-graph-based techniques report only 3,110 (2,365+743+2) overlapping pairs, but also identify 109 (84+25) distinct code clones not captured by EClone. Nonetheless, EClone generates a substantial number of false positives (33,389), as shown in Figure 3b.
- **Across non-graph-based techniques:** SmartEmbed, Deckard, and NiCad collectively detect 2,392 code clone pairs, as depicted in Figure 3a. SourcererCC detects the largest number of clone pairs (3,217) among the non-graph-based techniques, with 2,390 of these overlapping with those identified by SmartEmbed, Deckard, and NiCad. This significant overlap suggests that SourcererCC achieves a comparable level of true positives in clone detection as the combined output of SmartEmbed, Deckard, and NiCad.

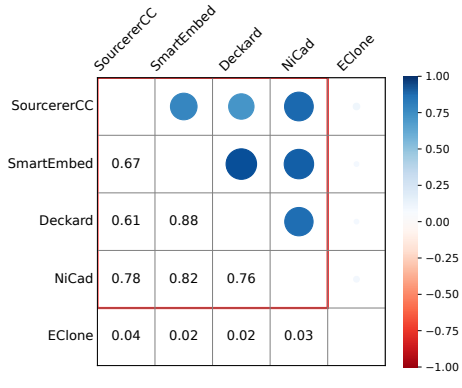


Fig. 1. Pairwise Agreement on Clone Labels of Smart Contract Pairs between Clone Detection Techniques.

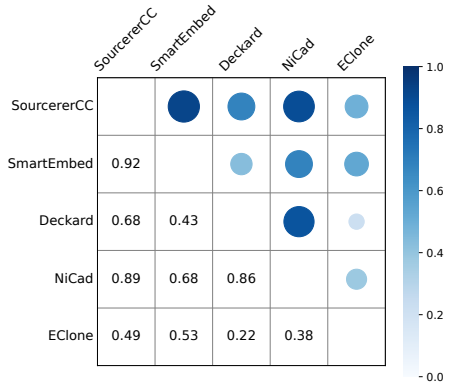
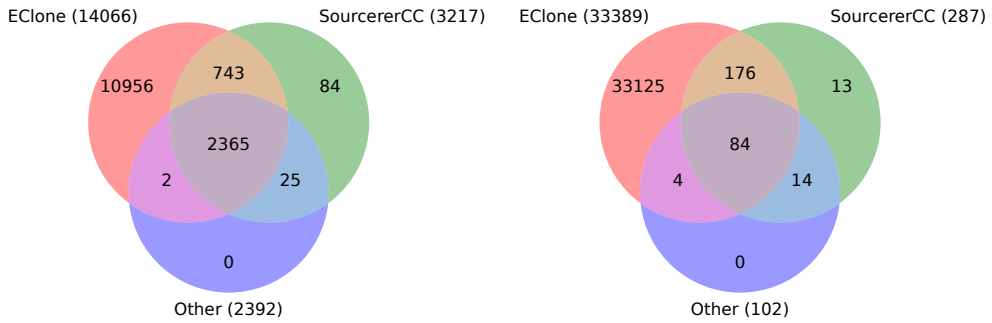


Fig. 2. Intraclass Correlation Coefficients of Similarity Scores Produced by Clone Detection Techniques.



(a) True Positives (b) False Positives

Fig. 3. Smart Contract Clones Detected by Clone Detection Techniques.

Finding 4: EClone detects 88.2% of the total clone pairs in Dataset II but generates a significant number of false positives, demonstrating poor agreement with non-graph-based techniques on clone labeling. SourcererCC, by contrast, exhibits substantial overlap with other non-graph-based techniques, complementing EClone by improving true positive rates in clone detection.

4.2.2 Agreement in Similarity Scores. Figure 2 illustrates the agreement on similarity scores for smart contract pairs across the selected techniques. Specifically, the entries below the diagonal in Figure 2 detail the ICC values for each pair of techniques. As indicated by the ICC values, we observed that:

- SourcererCC and SmartEmbed exhibit perfect agreement in their similarity scores for smart contract pairs (ICC = 0.92), suggesting that they may be reliably used interchangeably when evaluating the similarity of smart contracts.
- NiCad demonstrates strong agreement with both SourcererCC (ICC = 0.89) and Deckard (ICC = 0.86) in the similarity scores for smart contract pairs. Meanwhile, SourcererCC and Deckard demonstrate moderate agreement in the similarity scores (ICC = 0.68), suggesting that NiCad may serve as a complementary technique for validating similarity across these three techniques.
- EClone shows moderate agreement with SmartEmbed (ICC = 0.53), but exhibits poor agreement with the other three techniques (ICC = 0.49, 0.22 and 0.38). This disparity may be attributed to the capability of SmartEmbed, as a learning-based technique for clone detection, to capture some of the semantic features in smart contracts similar to EClone, which might be overlooked by the other techniques.

Finding 5: SourcererCC and SmartEmbed exhibit perfect agreement in their similarity scores for code pairs, making them reliably interchangeable. The strong agreement of NiCad with SourcererCC and Deckard suggests its suitability for validating similarity scores. In addition, the moderate to poor agreement of EClone with other techniques on similarity scores emphasizes its distinct capability in detecting semantic similarities.

4.2.3 Case Study. To explore the potential for the complementarity of existing techniques, we further analyze the cases of smart contract pairs with disagreement in labels and similarity scores across the selected techniques.

Table 6. A True Positive Case Detected by EClone and Missed by Other Clone Detection Techniques.

Contract A:

```
1  pragma solidity ^0.4.18;
2
3  contract ERC20Interface {
4      function allowance(address tokenOwner
       , address spender) public constant
       returns (uint remaining);
5      ...
6  }
7  ...
8  contract Cryptshopper is ERC20Interface,
   Owned, SafeMath {
9      function allowance(address tokenOwner
       , address spender) public constant
       returns (uint remaining) {
10         return allowed[tokenOwner][
           spender];
11     }
12     ...
13 }
```

Contract B:

```
1  pragma solidity ^0.4.4;
2
3  ...
4  contract StandardToken is Token {
5      function allowance(address _owner,
       address _spender) constant returns (
       uint256 remaining) {
6          return allowed[_owner][_spender];
7      }
8      ...
9  }
10 contract ERC20Token is StandardToken {
11   ...
12 }
```

The first case is shown in Table 6, representing a pair of smart contracts (Contract A[5] and B[6]) detected as a Type-4 code clone by EClone, but missed by other techniques.

The pair of smart contracts are semantically identical considering they both implement a basic ERC20 token. Consequently, EClone considers Contracts A and B as code clones with a similarity score of 0.92.

On the other hand, the pair of smart contracts differ in their ASTs, leading to their low similarity score in tree-based clone detection techniques (SmartEmbed = 0.77, Deckard = 0). As shown in Table 6, Contract A, as an ERC20 token, implements relevant functions in the self-defined `Cryptshopper` contract, which implements the `ERC20Interface` abstract contract that aligns with ERC20 standard. In contrast, Contract B implements relevant functions in the `StandardToken` contract, and further inherits the `StandardToken` contract to implement the `ERC20Token` contract.

Furthermore, the pair of smart contracts use different names for parameters in semantically identical functions, causing their low similarity in token-based clone detection techniques (SourcererCC = 0.60). For instance, as shown in Table 6, for the semantically similar function `allowance()` in both contracts, Contract A uses `tokenOwner` and `spender` as the parameter names (line 9), while Contract B uses `_owner` and `_spender` (line 5).

The second case, illustrated in Table 7, involves a pair of smart contracts, Contract C[7] and Contract D[8], which is detected as code clones by EClone with a similarity score of 0.89, despite exhibiting limited syntactic or semantic resemblance. From a syntactic perspective, SourcererCC, SmartEmbed, Deckard and NiCad report similarity scores of 0, 0.33, 0, and 0.03, respectively. In terms of semantic similarity, Contract C enables users to send ETH, with the highest contributor being marked as the "Richest". In contrast, Contract D facilitates the locking of tokens for a duration of 52 weeks, after which the tokens are released as payments to a designated beneficiary.

Note that EClone measures the similarity of code pairs by analyzing the corresponding basic blocks. As such, EClone outputs high similarity scores for Contracts C and D due to their substantial resemblance at the basic block level. For instance, while the variable `theAddress` at line 5 of Contract C and the variable `owner` at line 4 of Contract D differ considerably in their semantics, the difference is diminished post-compilation. The Solidity compiler automatically generates a `getter()` function

[5] https://etherscan.io/address/0x79acf4a12ebadcfe57833260b5863b34314da737
[6] https://etherscan.io/address/0x8bf65fcd0837e64c799228c7e3671217185c9b02
[7] https://etherscan.io/address/0x198889d1b8948a297048d85eb99d0b692c2a699d
[8] https://etherscan.io/address/0xbcc219fe1d2453d9ce1c14067e90a17f41a46caa

Table 7. A False Positive Case Detected by EClone.

Contract C:

```
pragma solidity ^0.4.19;

contract TheRichest {
    ...
    address public theAddress;
    ...
    function TheRichest() public {
        ...
        theAddress = msg.sender;
    }
    function () public payable {
        if (msg.value > theBid) {
            theAddress = msg.sender;
          ...
        }
    }
    ...
}
```

Contract D:

```
pragma solidity ^0.4.18;
...
contract Ownable {
    address public owner;
    ...
}
contract lockEtherPay is Ownable {
    ...
    function lock() public onlyOwner
    returns (bool){
        require(!isLocked);
        require(tokenBalance() > 0);
        start_time = now;
        end_time = start_time.add(
        fifty_two_weeks);
        isLocked = true;
    }
    ...
}
```

for each public variable in a smart contract. Consequently, after compilation, both the `theAddress` variable in Contract C and the `owner` variable in Contract D are transformed into identical basic blocks, specifically: `JUMPDEST PUSH1 PUSH1 SWAP1 SLOAD SWAP1 PUSH2 EXP SWAP1 DIV PUSH20 AND DUP2 JUMP`.

4.3 Combination of Existing Techniques (RQ3)

4.3.1 Performance Achieved by Combining Different Techniques. In RQ2, we observed the complementary strengths of EClone and the four other techniques regarding code clone labels, leading us to set eight different combinations of these techniques as shown in Table 8. Specifically, Table 8 outlines four sections of recall and precision values on Dataset II achieved by: (i) individual technique as a baseline, (ii) EClone combined with a non-graph-based technique, (iii) EClone and SourcererCC combined with another non-graph-based technique, and (iv) all techniques combined for clone detection in smart contracts. Note that each clone detection tool is run independently. A pair of smart contracts is classified as a clone if identified as such by one of the tools. We highlight the best recall, precision, and F1 score in each section. Overall, the combination of EClone and SourcererCC outperforms any individual in terms of F1 score. Additionally, with regard to execution overhead, the combination of EClone and SourcererCC demonstrates superior performance compared to other combinations.

> **Finding 6:** The combination of EClone and SourcererCC outperforms individual techniques in terms of F1 score. Considering execution overhead, the combination of EClone and SourcererCC exhibits superior performance compared to other combinations.

4.3.2 Integration for Optimal Performance. Although the combination of EClone and SourcererCC excels in recall and runtime efficiency compared to other combinations of the selected techniques, it suffers from relatively low precision. The low precision is largely attributed to the significant false positive rate associated with EClone. Thus, we consider exploring avenues to improve overall performance by mitigating the occurrence of false positives. Specifically, we analyzed the distributions of similarity scores for true and false positives in EClone, as measured by EClone and SourcererCC, respectively (see Appendix B in the replication package for details). EClone demonstrates no substantial difference in similarity scores between true and false positives. In contrast, SourcererCC yields notably higher similarity scores for true positives (median = 0.48) compared to

Table 8. Recall, Precision and F1 Scores Achieved by Combining Different Techniques.

	Recall (in %)	Precision (in %)	F1 Score
SourcererCC	20.5	91.8	0.3347
EClone	**89.5**	29.6	**0.4453**
SmartEmbed	11.2	94.9	0.1997
Deckard	10.0	**99.8**	0.1814
NiCad	14.1	96.3	0.2460
Combinations			
EClone + SourcererCC	**90.2**	**29.8**	**0.4478**
EClone + SmartEmbed	89.6	29.7	0.4457
EClone + Deckard	89.6	29.7	0.4456
EClone + NiCad	89.6	29.7	0.4457
EClone + SourcererCC + SmartEmbed	**90.2**	**29.8**	**0.4478**
EClone + SourcererCC + Deckard	**90.2**	**29.8**	**0.4478**
EClone + SourcererCC + NiCad	**90.2**	**29.8**	**0.4478**
All (Five Techniques)	**90.2**	**29.8**	**0.4478**

false positives (median = 0.22), suggesting the potential to use similarity scores of SourcererCC to reduce false positives in EClone.

Consequently, we develop a framework for a refined integration of SourcererCC and EClone, named SourceRECLone, as shown in Figure 4, which consists of two phases: (1) parameter optimization, and (2) clone detection. SourceRECLone takes a dataset of smart contracts D as input, dividing it into a training set $D_{Training}$ and a test set D_{Test}. During the *parameter optimization* phase, SourceRECLone utilizes both SourcererCC and EClone for selecting EClone thresholds to determine optimal thresholds for EClone, aiming to enhance clone detection performance on $D_{Training}$. Specifically, SourceRECLone leverages SourcererCC to compute the similarity score of each pair of contracts in $D_{Training}$ (1.a in Figure 4), and grouping contract pairs based on their similarity scores (1.b in Figure 4). For each group, SourceRECLone identifies an optimal threshold t_i for each group of contract pairs G_i with similarity scores below 0.7 (1.c in Figure 4), aiming to maximize the F1 score for clone detection using EClone. As a result, SourceRECLone derives an *EClone Threshold Mapping* that assigns a specific threshold to each group, facilitating the selection

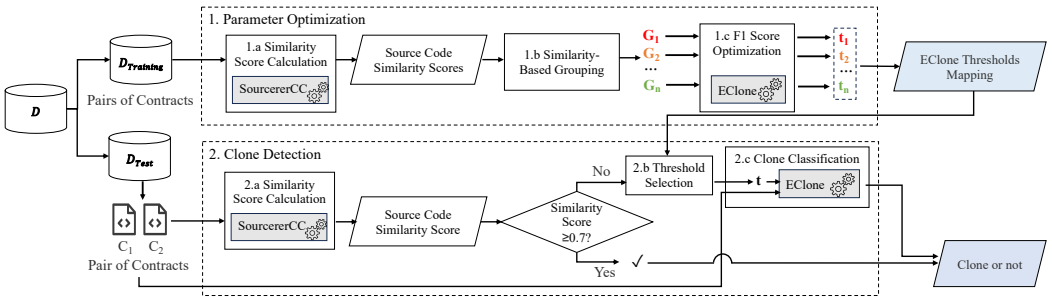


Fig. 4. Overview of SourceRECLone Framework.

of EClone thresholds for code pairs that lie in different similarity ranges. In the *clone detection* phase, SourceRECLone employs both SourcererCC and EClone to achieve effective clone detection on D_{Test}. Specifically, SourceRECLone leverages SourcererCC to calculate the similarity score for a given pair of contracts in D_{Test} (2.a in Figure 4). If the similarity score meets or exceeds 0.7, the pair is classified as a clone. If the score is lower than 0.7, a threshold t is selected based on the corresponding similarity score from the *EClone Threshold Mapping* (2.b in Figure 4). Finally, SourceRECLone employs EClone to compute the similarity score of the contract pair (2.c in Figure 4). The pair is classified as a clone if the calculated similarity score is greater than or equal to the selected threshold t.

Evaluation. We take Dataset II as D in the evaluation of SourceRECLone. We first randomly split the 72,010 code pairs in Dataset II into $D_{Training}$ with 50,407 pairs and D_{Test} with 21,603 pairs, following a 7:3 ratio. During the parameter optimization phase, we set 0.1 as the interval to segment $D_{Training}$, creating ten groups (G_1 to G_{10}) of code pairs across various similarity ranges, as well as corresponding thresholds t_i ($i = 1, 2, ..., 7$) to optimize F1 scores in EClone.

We first compare the F1 score of SourceRECLone for clone detection on D_{Test} against the five baseline techniques. As shown in Table 9, SourceRECLone achieves an overall F1 score of 0.6070, corresponding to a 36.9% improvement over EClone, which is the best-performing baseline that achieved an F1 score of 0.4433. We then evaluate the precision, recall, and F1 score of clone detection on D_{Test} of SourceRECLone, and compare them with a straightforward combination of SourcererCC (threshold = 0.7) and EClone (threshold = 0.84). As shown in Table 10, SourceRECLone achieves an overall precision of 46.4%, which is significantly higher than that of the straightforward combination, and a recall of 87.7%, with only a marginal decrease of 1.4% as compared to the straightforward combination, resulting in a 36.9% increase of F1 score (from 0.4433 to 0.6070).

Table 9. Performance Comparison of SourceRECLone and Baseline Techniques in Clone Detection.

Group (G_i)	**Similarity Range**	**F1 Score**					
		SourcererCC	EClone	SmartEmbed	Deckard	NiCad	SourceRECLone
G_1	[0,0.1)	-	0.0012	-	-	-	-
G_2	[0.1,0.2)	-	0.0806	-	-	-	0.1429
G_3	[0.2,0.3)	-	0.2501	-	-	-	0.2629
G_4	[0.3,0.4)	-	0.4625	-	-	-	0.4625
G_5	[0.4,0.5)	-	0.6739	-	-	-	0.6753
G_6	[0.5,0.6)	-	0.7849	-	-	-	0.8187
G_7	[0.6,0.7)	-	0.6998	-	-	-	0.7463
G_8	[0.7,0.8)	0.8707	0.8059	0.0800	0.1714	0.0606	0.8707
G_9	[0.8,0.9)	0.9776	0.9641	0.1718	0.0806	0.7512	0.9776
G_{10}	[0.9,1.0]	0.9825	0.9796	0.9657	0.9558	0.9776	0.9825
	Overall [0,1]	0.3346	0.4433	0.2053	0.1903	0.2487	0.6070

Table 10. Evaluation Results of SourceRECLone.

Parameter Optimization Phase				**Clone Detection Phase**					
Group (G_i)	**Similarity Range**	**# Code Pairs**	**Threshold (t_i)**	**Precision (in %)**		**Recall (in %)**		**F1 Score**	
				E + S → SourceRECLone		E + S → SourceRECLone		E + S → SourceRECLone	
G_1	[0,0.1)	9,938	0.99	0.1	0.0	50.0	0.0	0.0012	-
G_2	[0.1,0.2)	10,616	1.00	4.2	9.0	87.9	34.0	0.0806	0.1429
G_3	[0.2,0.3)	10,461	0.90	14.8	18.5	80.3	45.2	0.2501	0.2629
G_4	[0.3,0.4)	7,863	0.84	31.5	31.5	87.2	87.2	0.4625	0.4625
G_5	[0.4,0.5)	4,777	0.83	53.8	53.2	90.1	92.6	0.6739	0.6753
G_6	[0.5,0.6)	3,035	0.01	70.8	69.3	88.1	100.0	0.7849	0.8187
G_7	[0.6,0.7)	1,261	0.01	58.0	59.5	88.1	100.0	0.6998	0.7463
	Overall [0,1]			29.5	46.4	89.1	87.7	0.4433	0.6070

"E + S" refers to the straightforward combination of EClone and SourcererCC.

Finding 7: We propose a novel framework SourceRECLone, which consists of two phases: (1) parameter optimization, where EClone thresholds are refined with respect to SourcererCC similarity scores, and (2) clone detection, where both techniques are employed to achieve effective code clone detection in smart contracts. The evaluation results indicate that SourceRECLone surpasses the straightforward combination of SourcererCC and EClone, achieving a substantial increase in precision while experiencing only a slight reduction in recall, resulting in a 36.9% improvement in the F1 score.

5 Discussion

5.1 Implications

We reflect on our findings from the research questions, offering recommendations to assist practitioners in effectively leveraging existing techniques and tools for detecting smart contract clones. We also highlight the avenues of future research.

For Practitioners: In RQ1, our findings indicate that the existing techniques perform effectively in detecting Type-1 and Type-2 clones, with SourcererCC scaling well for inputs up to 1 million lines of code. Additionally, SourcererCC achieves the highest recall in detecting Type-3 clones, making it recommended for detecting Type-1, Type-2, and Type-3 clones, particularly in large-scale datasets. Conversely, existing techniques fall short in effectively detecting Type-4 clones. Although EClone achieves a recall of 88.2% in detecting Type-4 clones, it produces a significant number of false positives. The framework proposed in RQ3 provides valuable insights for practitioners on how to combine existing techniques, such as EClone and SourcererCC, to surpass the current state of the art.

For Researchers: ***Precise and Scalable Detection of Type-4 Clones.*** Among the evaluated techniques, EClone stands out as the only technique capable of detecting Type-4 smart contract clones with a notable recall rate of 88.2%, albeit accompanied by a significant number of false positives and high execution overhead (RQ1). Our observations in RQ3 reveal that a straightforward combination of SourcererCC and EClone can outperform existing state-of-the-art techniques, with further improvements in precision and F1 score achievable through refined integration in our proposed framework. Additionally, in RQ2, we observe that EClone exhibits poor agreement in the similarity scores of code pairs when compared with text-based (NiCad), token-based (SourcererCC), and tree-based (Deckard) techniques. Future work could explore the synergistic potential of leveraging the complementarity between techniques with diverse code representations, aiming to improve both the precision and scalability of Type-4 clone detection for smart contracts. ***Bytecode-Level Clone Detection Support.*** Smart contracts on the Ethereum platform are deployed in the form of bytecode, with fewer than 1% disclosing their source code [10]. Nonetheless, we observe limited contributions in bytecode-level clone detection for smart contracts through our literature search, highlighting a critical gap in the existing literature. As demonstrated in our case study in RQ2, the bytecode representation of smart contracts often suffers from a significant loss of high-level semantic information due to the compilation process. This deficit not only obfuscates meaningful patterns but also contributes to an increased likelihood of false positives in clone detection techniques like EClone. Future work could explore scalable byte-code level clone detection for smart contracts, with a particular focus on mitigating the challenges posed by semantic degradation during compilation. Moreover, we observe that numerous smart contracts in our dataset explicitly denote the version of the Solidity compiler used. The extent to which variations in compiler versions influence the performance and reliability of byte-code level clone detection techniques remains unclear and presents an important avenue for future research.

5.2 Threats to Validity

The primary threat to external validity stems from the curation of Dataset II, which serves as the benchmark for evaluating the techniques. The process of labeling code pairs in Dataset II involves manual effort, which may introduce potential subjectivity and bias. To mitigate such a threat, two experts independently labeled the code pairs, with a moderator resolving any disagreements to minimize individual bias.

The main threat to construct validity arises from the configurations of the selected techniques, as these configurations can significantly impact the performance of the techniques. To mitigate such a threat, we used the configurations that achieved optimal performance for the techniques as reported in previous studies, and carefully experimented with these configurations.

6 Related Work

6.1 Evaluation of Clone Detection Techniques and Tools

We start by discussing the evaluations of clone detection techniques conducted by their respective authors. Jiang et al. [24] evaluated Deckard on large codebases written in C and Java, focusing on the evaluation of clone quantity and quality, as well as scalability. Additionally, they compared Deckard against CloneDR and CP-Miner. Sajnani et al. [37] evaluated SourcererCC on a large inter-project repository, and compared it with CCFinderX, Deckard, iClones and NiCad using two benchmarks. Wang et al. [45] compared CCAligner with NiCad and SourcererCC across eight projects written in C and Java. Liu et al. [31] evaluated EClone on two curated datasets of smart contracts, demonstrating the potential of EClone in detecting semantic clones.

Several researchers have conducted comparative studies across multiple clone detection techniques. For instance, Tsantalis et al. [41] performed a large-scale comparison of four clone detection techniques, i.e., CCFinder, Deckard, CloneDR, and NiCad, showing that tree-based techniques excel in detecting Type-2 and Type-3 clones in production code. Ragkhitwetsagul et al. [35] evaluated 30 clone detection techniques and tools on Java codebases under various scenarios. Farmahinifarahani et al. [17] evaluated the precision of eight clone detection techniques, suggesting the use of type-based precision when comparing clone detection techniques. Svajlenko et al. [40] proposed a mutation and injection framework to evaluate the recall of ten clone detection techniques on six benchmarks in Java, C and C#. Most recently, Wang et al. [46] compared 12 clone detection techniques with diverse code representations on a Java dataset composing over 8 million tagged clone pairs.

In summary, internal evaluations by authors involve codebases of different programming languages, with various versions of techniques and tools, which makes their results difficult to compare directly. Our work seeks to provide an independent evaluation of clone detection techniques, focusing specifically on Solidity for smart contracts, an area that has not been explored in prior studies.

6.2 Empirical Studies on Code Reuse in Smart Contracts

He et al. [21] conducted an empirical study to examine code reuse practices in the Ethereum ecosystem, revealing that over 96% of the contracts have duplicates, with 9.7% of the duplicate contracts exhibiting identical vulnerabilities. Kondo et al. [28] used Deckard to detect a clone ratio of 79.2% at the contract level on Ethereum, identifying 16.7% as Type-1 clones and 43.4% as Type-2 clones. Khan et al. [27] further extended Kondo et al.'s work, applying NiCad to detect a clone ratio of 30.13% at the function level, incorporating Type-3 clones into the analysis. Chen et al. [11] investigated the impacts of code reuse in smart contracts, identifying common revision patterns associated with code reuse. Pierro et al. [34] classified code clones in smart contracts into local

and global clones, observing an upward trend in global clones and a decline in local clones over time. Most recently, Sun et al. [38] conducted an empirical study with over 350,000 Solidity smart contracts, analyzing their compositions and the reuse of subcontracts to reveal prevalent patterns in smart contract development.

In this work, we consider multiple levels of granularity to evaluate the effectiveness of clone detection techniques for smart contracts, drawing on insights from the aforementioned studies. Unlike prior studies, our study also incorporates Type-4 clones, adding a new dimension to the investigation of code reuse in smart contracts.

7 Conclusion and Future Work

In this paper, we conducted a comprehensive empirical study to evaluate the capability of five representative clone detection techniques, SourcererCC, EClone, SmartEmbed, Deckard and NiCad, and explore their integration for optimal code clone detection of smart contracts in the blockchain ecosystem. We observed that these techniques report comparable clone ratios across various levels of granularity of code, achieving nearly 100% recall for Type-1 and Type-2 clones, with SourcererCC and EClone exceeding 99% recall for Type-3 clones. SourcererCC efficiently scales to 1 million lines of code, whereas EClone manages up to 1,000 lines of code. While EClone detects 89.5% of the clone pairs in our dataset, it produces numerous false positives and demonstrates poor agreement with other techniques on clone labels and similarity scores of code pairs. Conversely, SourcererCC aligns well with other techniques in the detected clones, complementing EClone by increasing true positive rates in clone detection. Based on our observations, we propose a two-phase framework that integrates SourcererCC and EClone, which (1) optimizes EClone thresholds based on the similarity scores from SourcererCC, and (2) uses both techniques to improve the precision of clone detection. Our framework achieves a notable increase in F1 score over a straightforward combination of SourcererCC and EClone.

Future work could put efforts into developing precise and scalable techniques for detecting Type-4 clones in smart contracts, bolstering the capabilities of bytecode-level clone detection, and investigating the impact of varying compiler versions on the effectiveness of clone detection techniques.

8 Data Availability

Our replication package is available online: https://doi.org/10.5281/zenodo.13756064.

Acknowledgments

This work was supported by the National Science Foundation of China (No. 62472383 and No. 62102358), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), and the Open Research Fund of the State Key Laboratory of Blockchain and Data Security, Zhejiang University.

References

[1] 2024. Contracts - Solidity v0.8.27 Documentation. [Online]. https://docs.soliditylang.org/en/v0.8.27/contracts.html.
[2] 2024. Solidity, the Smart Contract Programming Language. [Online]. https://github.com/ethereum/solidity.
[3] Maher Alharby and Aad Van Moorsel. 2017. Blockchain-based smart contracts: A systematic mapping study. *arXiv preprint arXiv:1710.06372* (2017). https://doi.org/10.48550/arXiv.1710.06372
[4] John J Bartko. 1966. The intraclass correlation coefficient as a measure of reliability. *Psychological reports* 19, 1 (1966), 3–11. https://doi.org/10.2466/pr0.1966.19.1.3
[5] Rob Behnke. 2022. Explained: The Multichain Hack. https://halborn.com/explained-the-multichain-hack-january-2022/ Accessed: 2024-07-22.
[6] Binance. 2024. BNB Smart Chain. https://docs.bnbchain.org/bnb-smart-chain/ Accessed: 2024-07-22.

[7] Mihailo Bjelic, Sandeep Nailwal, Amit Chaudhary, and Wenxuan Deng. 2024. POL: one token for all polygon chains. https://polygon.technology/papers/pol-whitepaper Accessed: 2024-07-22.
[8] Benjamin Bowman and H Howie Huang. 2020. VGRAPH: A robust vulnerable code clone detection system using code property triplets. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 53–69. https://doi.org/10.1109/EuroSP48549.2020.00012
[9] Vitalik Buterin. 2014. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/ Accessed: 2024-07-22.
[10] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 1503–1520. https://doi.org/10.1145/3319535.3345664
[11] Xiangping Chen, Peiyong Liao, Yixin Zhang, Yuan Huang, and Zibin Zheng. 2021. Understanding code reuse in smart contracts. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 470–479. https://doi.org/10.1109/SANER50967.2021.00050
[12] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. https://doi.org/10.1177/001316446002000104
[13] James R Cordy. 2006. The TXL source transformation language. *Science of Computer Programming* 61, 3 (2006), 190–210. https://doi.org/10.1016/j.scico.2006.04.002
[14] James R Cordy and Chanchal K Roy. 2011. The NiCad clone detector. In *2011 IEEE 19th international conference on program comprehension*. IEEE, 219–220. https://doi.org/10.1109/ICPC.2011.26
[15] Chris Dannen. 2017. *Introducing Ethereum and solidity*. Vol. 1. Springer. https://doi.org/10.1007/978-1-4842-2535-6
[16] Amir M Ebrahimi, Bram Adams, Gustavo A Oliva, and Ahmed E Hassan. 2024. A large-scale exploratory study on the proxy pattern in Ethereum. *Empirical Software Engineering* 29, 4 (2024), 1–51. https://doi.org/10.1007/s10664-024-10485-1
[17] Farima Farmahinifarahani, Vaibhav Saini, Di Yang, Hitesh Sajnani, and Cristina V Lopes. 2019. On precision of code clone detection tools. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 84–94. https://doi.org/10.1109/SANER.2019.8668015
[18] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.
[19] 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 (2020), 2874–2891. https://doi.org/10.1109/TSE.2020.2971482
[20] Sicheng Hao, Yuhong Nan, Zibin Zheng, and Xiaohui Liu. 2023. SmartCoCo: Checking Comment-Code Inconsistency in Smart Contracts via Constraint Propagation and Binding. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 294–306. https://doi.org/10.1109/ASE56229.2023.00142
[21] Ningyu He, Lei Wu, Haoyu Wang, Yao Guo, and Xuxian Jiang. 2020. Characterizing code clones in the ethereum smart contract ecosystem. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*. Springer, 654–675. https://doi.org/10.1007/978-3-030-51280-4_35
[22] Yangyu Hu, Guosheng Xu, Bowen Zhang, Kun Lai, Guoai Xu, and Miao Zhang. 2020. Robust app clone detection based on similarity of ui structure. *IEEE Access* 8 (2020), 77142–77155. https://doi.org/10.1109/ACCESS.2020.2988400
[23] Teng Huang, Jiahui Huang, Yan Pang, and Hongyang Yan. 2023. Smart contract watermarking based on code obfuscation. *Information Sciences* 628 (2023), 439–448. https://doi.org/10.1016/j.ins.2023.01.126
[24] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105. https://doi.org/10.1109/ICSE.2007.30
[25] C. J. Kapser and M. W. Godfrey. 2008. "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software. *Empirical Software Engineering* 13, 6 (2008), 645–692. https://doi.org/10.1007/s10664-008-9076-6
[26] I. Keivanloo, F. Zhang, and Y. Zou. 2015. Threshold-Free Code Clone Detection for a Large-Scale Heterogeneous Java Repository. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*. 201–210. https://doi.org/10.1109/SANER.2015.7081830
[27] Faizan Khan, Istvan David, Daniel Varro, and Shane McIntosh. 2022. Code cloning in smart contracts on the ethereum platform: An extended replication study. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2006–2019. https://doi.org/10.1109/TSE.2022.3207428
[28] Masanari Kondo, Gustavo A Oliva, Zhen Ming Jiang, Ahmed E Hassan, and Osamu Mizuno. 2020. Code cloning in smart contracts: a case study on verified contracts from the ethereum blockchain platform. *Empirical Software Engineering* 25 (2020), 4617–4675. https://doi.org/10.1007/s10664-020-09852-5

[29] Terry K Koo and Mae Y Li. 2016. A guideline of selecting and reporting intraclass correlation coefficients for reliability research. *Journal of Chiropractic mMedicine* 15, 2 (2016), 155–163. https://doi.org/10.1016/j.jcm.2016.02.012

[30] Han Liu, Zhiqiang Yang, Yu Jiang, Wenqi Zhao, and Jiaguang Sun. 2019. Enabling clone detection for ethereum via smart contract birthmarks. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 105–115. https://doi.org/10.1109/ICPC.2019.00024

[31] Han Liu, Zhiqiang Yang, Chao Liu, Yu Jiang, Wenqi Zhao, and Jiaguang Sun. 2018. Eclone: Detect semantic clones in ethereum via symbolic transaction sketch. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 900–903. https://doi.org/10.1145/3236024.3264596

[32] Aaron K Massey, Richard L Rutledge, Annie I Antón, and Peter P Swire. 2014. Identifying and classifying ambiguity for regulatory requirements. In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*. IEEE, 83–92. https://doi.org/10.1109/RE.2014.6912250

[33] Multichain. 2024. Multichain Smart Contracts. https://github.com/anyswap/multichain-smart-contracts Accessed: 2024-07-22.

[34] Giuseppe Antonio Pierro and Roberto Tonelli. 2021. Analysis of source code duplication in ethreum smart contracts. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 701–707. https://doi.org/10.1109/SANER50967.2021.00089

[35] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. 2018. A comparison of code similarity analysers. *Empirical Software Engineering* 23 (2018), 2464–2519. https://doi.org/10.1007/s10664-017-9564-7

[36] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[37] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. 1157–1168. https://doi.org/10.1145/2884781.2884877

[38] Kairan Sun, Zhengzi Xu, Chengwei Liu, Kaixuan Li, and Yang Liu. 2023. Demystifying the Composition and Code Reuse in Solidity Smart Contracts. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 796–807. https://doi.org/10.1145/3611643.3616270

[39] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 476–480. https://doi.org/10.1109/ICSME.2014.77

[40] Jeffrey Svajlenko and Chanchal K Roy. 2019. The mutation and injection framework: Evaluating clone detection tools with mutation analysis. *IEEE Transactions on Software Engineering* 47, 5 (2019), 1060–1087. https://doi.org/10.1109/TSE.2019.2912962

[41] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. 2015. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1055–1090. https://doi.org/10.1109/TSE.2015.2448531

[42] Zhiyuan Wan, Xin Xia, and Ahmed E Hassan. 2019. What Do Programmers Discuss about Blockchain? A Case Study on the Use of Balanced LDA and the Reference Architecture of a Domain to Capture Online Discussions about Blockchain platforms across the Stack Exchange Communities. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2921343

[43] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. 2021. Smart contract security: A practitioners' perspective. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1410–1422. https://doi.org/10.1109/ICSE43902.2021.00127

[44] Che Wang, Yue Li, Jianbo Gao, Ke Wang, Jiashuo Zhang, Zhi Guan, and Zhong Chen. 2024. SolaSim: Clone Detection for Solana Smart Contracts via Program Representation. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 258–269. https://doi.org/10.1145/3643916.3644406

[45] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. 2018. CCAligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*. 1066–1077. https://doi.org/10.1145/3180155.3180179

[46] Yuekun Wang, Yuhang Ye, Yueming Wu, Weiwei Zhang, Yinxing Xue, and Yang Liu. 2023. Comparison and evaluation of clone detection techniques with different code representations. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 332–344. https://doi.org/10.1109/ICSE48619.2023.00039

[47] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

Received 2024-09-12; accepted 2025-01-14