Automated Test Generation For Smart Contracts via On-Chain Test Case Augmentation and Migration

Jiashuo Zhang
School of Computer Science
Peking University
Beijing, China
zhangjiashuo@pku.edu.cn

Jiachi Chen*
Sun Yat-sen University & The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Zhuhai, China
chenjch86@mail.sysu.edu.cn

John Grundy
Monash University
Melbourne, Australia
john.grundy@monash.edu

Jianbo Gao*
Beijing Key Laboratory of Security and Privacy in Intelligent Transportation
Beijing Jiaotong University
Beijing, China
gao@bjtu.edu.cn

Yanlin Wang
School of Software Engineering
Sun Yat-sen University
Zhuhai, China
yanlin-wang@outlook.com

Ting Chen
University of Electronic Science and Technology of China
Chengdu, China
brokendragon@uestc.edu.cn

Zhi Guan
National Engineering Research Center for Software Engineering, Peking University
Beijing, China
guan@pku.edu.cn

Zhong Chen*
School of Computer Science
Peking University
Beijing, China
zhongchen@pku.edu.cn

***Abstract*—Pre-deployment testing has become essential to ensure the functional correctness of smart contracts. However, since smart contracts are stateful programs integrating many different functionalities, manually writing test cases to cover all potential usages requires significant effort from developers, leading to insufficient testing and increasing risks in practice. Although several testing techniques for smart contracts have been proposed, they primarily focus on detecting common low-level vulnerabilities such as re-entrancy, rather than generating expressive and function-relevant test cases that can reduce manual testing efforts. To bridge the gap, we propose SOLMIGRATOR, an automated technique designed to generate expressive and representative test cases for smart contracts. To our knowledge, SOLMIGRATOR is the first migration-based test generation technique for smart contracts, which extracts test cases from real-world usages of on-chain contracts and migrates them to test newly developed smart contracts with similar functionalities. Given a target smart contract to be tested and an on-chain similar source smart contract, SOLMIGRATOR first transforms the on-chain usage of the source contract into off-chain executable test cases based on on-chain transaction replay and dependency analysis. It then employs fine-grained static analysis to migrate the augmented test cases from the source to the target smart contract. We built a prototype of SOLMIGRATOR and have evaluated it on real-world smart contracts within the two most popular categories, ERC20 and ERC721. Our evaluation results demonstrate that SOLMIGRATOR effectively extracts test cases from existing on-chain smart contracts and accurately migrates them across different smart contracts, achieving an average precision of 96.3% and accuracy of 93.6%. Furthermore, the results indicate that these migrated test cases effectively cover common key functionalities of the target smart contracts. This provides promising evidence that real-world usages of existing smart contracts can be transformed into effective test cases for other newly developed smart contracts.**

***Index Terms*—Ethereum, smart contracts, test generation, test migration**

*Corresponding authors

I. INTRODUCTION

Smart contracts have attracted rapid development and widespread applications since their introduction. Due to the finance-related nature of smart contracts and recurrent security incidents, pre-deployment testing has become essential in the contract-development lifecycle [1], [2]. To gain confidence in the functional correctness of smart contracts, developers in over 90% of projects conduct functional testing during their development process [3], using frameworks like Hardhat [4] and Truffle [5] to develop and execute test cases. However, since smart contracts are stateful programs integrating many different functionalities, manually developing test cases that thoroughly cover the potential usage of each functionality is often both challenging and costly, requiring significant effort from developers. Such effort may prevent smart contracts from being effectively tested and increase the risk of undetected functional bugs in practice.

Automated test case generation techniques have been shown to be promising in reducing the effort of writing test cases and improving testing effectiveness [6], [7]. Although many testing

techniques for smart contracts have been proposed [8]–[10], they are typically fuzzing-based techniques aimed at detecting common low-level vulnerabilities such as re-entrancy [11] and integer overflow [12]. Their primary goal is to generate numerous randomized test cases to detect as many vulnerabilities as possible, rather than generating a good set of function-relevant test cases that represent the canonical usages of smart contracts. Consequently, the task of developing expressive and representative test cases still falls to real-world developers.

In this work, we aim to reduce the cost of testing smart contracts by extracting and migrating test cases from existing on-chain smart contracts. We are motivated in this direction by two observations. First, there are billions of historical transactions on Ethereum, documenting real-world call data of millions of on-chain contracts [13]. This on-chain usage data, with comprehensive information about common usage patterns of smart contracts, can become valuable test cases for other smart contracts. Second, although smart contracts have enabled a wide range of different applications, there are many cases where smart contracts share quite similar functionalities [14]. Previous studies found that 80% of a typical smart contract's code could be reused from external code sources [15], leading to similar functionalities and implementations. Thus it may be possible to *migrate* test cases extracted from existing on-chain contracts to other newly developed contracts in order to more thoroughly test these new smart contracts.

Based on these two key observations, we propose SOLMIGRATOR, *the first migration-based technique* that generates test cases for smart contracts by extracting and migrating on-chain usages of smart contracts with similar functionalities. Given (i) a target new smart contract to test and (ii) an on-chain source smart contract sharing common functionalities, SOLMIGRATOR takes on-chain transaction data for the source contract as input and employs two phases – *test augmentation* and *test migration* – to generate test cases for the new target contract. During *test augmentation*, SOLMIGRATOR analyzes the historical transactions of the source contract and transforms them into off-chain executable test cases, including test transaction sequences and test assertions. To ensure testing efficiency, SOLMIGRATOR de-duplicates test cases based on execution trace analysis and reduces the test sequence length through transaction dependency analysis and reconstruction. During *test migration*, SOLMIGRATOR employs fine-grained static analysis techniques, including inter-function control/data flow analysis and taint analysis, to match and migrate the augmented test cases from the source contract to the target contract. These migrated test cases can be directly used to test the target contract, while preserving testing wisdom extracted from real-world usages of the source contract.

To validate SOLMIGRATOR's effectiveness, we evaluated it on real-world smart contracts in two of the most prevalent categories, *i.e.*, ERC20 and ERC721. To evaluate test augmentation, we sampled 93 ERC20 and 51 ERC721 on-chain smart contracts and used SOLMIGRATOR to augment test cases based on their historical transactions. SOLMIGRATOR successfully augmented a total of 967 test cases from these contracts, corresponding to an average of 5.1 test cases per ERC20 contract and 9.6 test cases per ERC721 smart contract. To evaluate test migration, we chose ten popular contracts for each category and permuted them into 90 source-target pairs per category. We then used SOLMIGRATOR to migrate test cases between these pairs, resulting in a total of 1,719 attempted test case migrations for evaluating SOLMIGRATOR. SOLMIGRATOR achieves an average precision of 96.3% and accuracy of 93.6%, demonstrating the effectiveness of the migration process. After comparing the functions tested by migrated test cases with those used in real transaction history of the target contract, we validated that these migrated test cases can cover common real-world usages of the target contract, demonstrating the usability of SOLMIGRATOR.

This research makes the following key contributions:

- We propose SOLMIGRATOR, the first migration-based automated test case generation technique for smart contracts. It can extract and migrate test cases from real-world usage of similar on-chain smart contracts to generate test cases for newly developed smart contracts.
- We incorporate a set of new approaches into SOLMIGRATOR, including test case augmentation based on on-chain transaction replay and dependency analysis, and test case migration based on fine-grained static analysis.
- Our empirical evaluation of SOLMIGRATOR on real-world smart contracts demonstrates its effectiveness. This provides initial yet promising evidence about the feasibility of extracting test cases from on-chain contracts and migrating them across contracts.
- We make the source code of SOLMIGRATOR and the experimental data available at https://github.com/Jiashuo-Zhang/SolMigrator, to support further studies in this field.

II. BACKGROUND AND MOTIVATION

Smart contracts are programs running on a blockchain. They differ from regular programs in that they are immutable after deployment, consume resources (*e.g.*, gas on Ethereum [16]) to run, generate public transaction records about their execution as they run, and may conform to various standards [17], such as ERC20 [18] and ERC721 [19], which relate to token representation. To be executed, smart contracts first need to be compiled into bytecode, *i.e.*, a sequence of opcodes that can be interpreted by the Ethereum Virtual Machine (EVM) [16]. Each opcode refers to a specific operation that transitions the EVM from the current state to the next, including stack, memory, and storage operations, arithmetic calculations, and many other operations [20]. When a smart contract is called, the EVM will sequentially execute these opcodes and update the contract's state based on the execution results.

A. Smart Contract Testing

Smart contract testing is an essential step to ensure the functional correctness of smart contracts [2]. A typical test case for smart contracts comprises a sequence of test transactions and a set of test oracles [4]. The test transactions simulate potential uses of the function being tested, while the

oracles define the expected behavior during test execution. For example, when testing a token transfer function, the transaction sequences would include token transfer operations, and the oracle would validate certain assertions, such as requiring the transfer operation to be reverted if the token sender's balance is less than the amount to be transferred. Since oracles in smart contracts are assertion-based, we use the terms "assertion" and "oracle" interchangeably in this paper.

Currently, more than 90% of smart contract projects conduct functional testing during their development process [3], highlighting the importance of real-world testing practices. To assist developers in managing and executing their smart contract test cases, testing frameworks like Hardhat [4] and Truffle [5] have been introduced. These frameworks provide a unified interface for writing test transactions and extend assertion libraries like Chai [21] for writing test assertions. They automate the execution of test cases, including deploying the contract within a testing environment, executing the test transactions, and checking each test assertion to determine the success of the test. However, developing a good set of effective functional test cases for smart contracts is still very time-consuming and error-prone.

B. Motivation

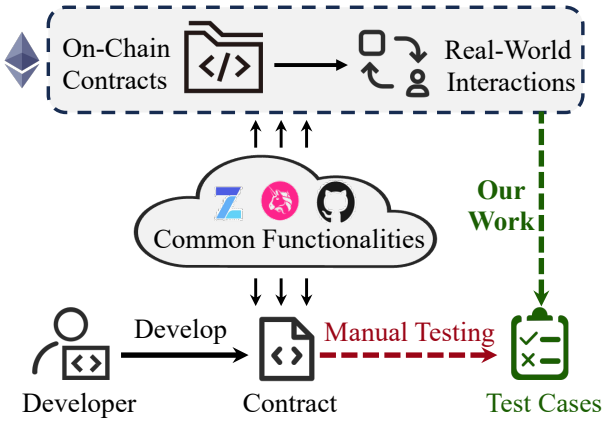

Fig. 1: A motivating scenario of SOLMIGRATOR

Consider the scenario in Fig. 1, where the developer Alice is implementing a smart contract. This contract involves several common functionalities, which have been widely used in existing smart contracts and implemented by many external code sources (*e.g.*, OpenZeppelin [22]). By reusing these external code sources, Alice efficiently completes the development process and begins testing the contract to verify the functional correctness of her contract. However, since the smart contract integrates many different functionalities, she finds that manually writing test cases to cover all potential usages of each functionality is both difficult and time-consuming.

At this point, SOLMIGRATOR enables Alice to generate test cases automatically, thereby reducing her manual effort. SOLMIGRATOR leverages a key insight: while Alice struggles with testing each potential usage of her contract, many of its functionalities, especially those with external code sources, have already been commonly implemented in existing smart contracts and extensively used in on-chain transactions. These real-world interactions cover various usages of each functionality, which are exactly what Alice aims to test. By using SOLMIGRATOR to transform these on-chain usages into off-chain executable test cases, Alice can avoid the repetitive and time-consuming task of manually writing test cases, enhancing the efficiency of her testing process.

III. OUR APPROACH

Fig. 2 provides an overview of SOLMIGRATOR. It takes a pair of user-specified source and target contracts as input and conducts a two-phase analysis, including *test augmentation* and *test migration*, to generate test cases for the target contract.

The *test augmentation* process is designed to extract off-chain executable test cases from the real-world interactions between users and existing on-chain smart contracts. It takes the source contract and its historical transactions as input. It then replays the transactions (Section IV-A) and analyzes dependency relations among them (Section IV-B). Based on these results, SOLMIGRATOR transforms the entire transaction history into a set of independent and self-contained test transaction sequences, covering different functionalities of the contract (Section IV-C). To enhance testing efficiency, SOLMIGRATOR conducts a transaction sequence folding process that reduces the number of transactions in each sequence while maintaining coverage of the same functionalities (Section IV-D). In the final step, SOLMIGRATOR executes these augmented test sequences and augments a set of test assertions that characterize the execution behaviors of the contract (Section IV-E).

After augmenting these test cases, including both test transaction sequences and assertions, SOLMIGRATOR *migrates* them from the source contract to the target contract. It takes the source code of both the source and target contracts as input and conducts fine-grained static analysis techniques on them. Based on these analyses, SOLMIGRATOR establishes matching relations between functions in the source and target contracts. It then migrates the test transactions (Section V-A) and assertions (Section V-B) for specific source functions to their counterparts in the target contract. Finally, it packs the migrated test cases into test scripts compatible with the Hardhat framework [4] (Section V-C), enabling developers to execute them directly on the target contracts.

IV. TEST AUGMENTATION

During test augmentation, SOLMIGRATOR takes the on-chain historical transactions of the source contract as inputs, and augments a set of off-chain test cases for the source contract. These augmented test cases, including both test transaction sequences and test assertions, will be further migrated from the source contract to the target contract in Section V.

A. Transaction Replay

Given an on-chain source contract, SOLMIGRATOR first replays historical on-chain transactions of the contract and

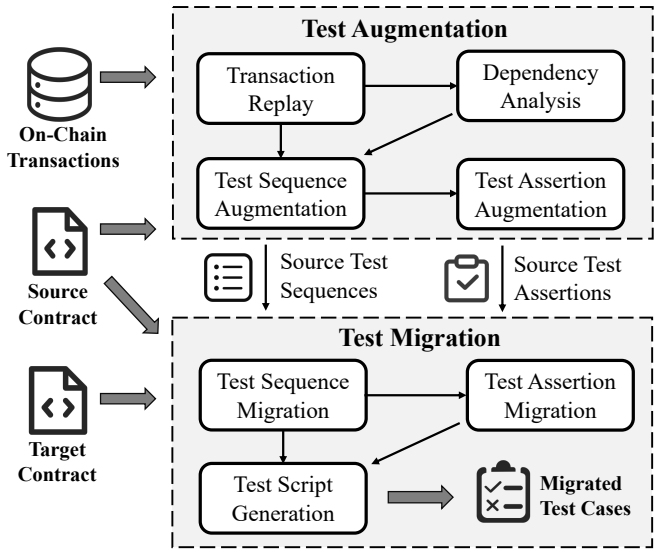

Fig. 2: The overview of SOLMIGRATOR

records the execution trace of each transaction. For each transaction, this execution trace records all executed opcodes and the intermediate states of EVM stack, memory, and storage during the transaction execution. To fetch these traces, SOLMIGRATOR uses the *debug_traceTransaction* API of Go-Ethereum [23] to interact with an Ethereum archive node [24], which maintains all on-chain state data since the genesis block.

B. Transaction Dependency Graph Construction

Based on the execution traces, SOLMIGRATOR extracts the read and write set of each transaction and then constructs a transaction dependency graph. For each transaction, SOLMIGRATOR analyzes the storage opcodes (*i.e.*, `SLOAD` and `SSTORE`) in its execution trace and records the EVM storage slots and values that are read or written as $\langle slot, value \rangle$ tuples. To preserve the order of the read and write operations on the same storage slot, SOLMIGRATOR indexes each tuple by the key $\langle block, tx \rangle$, where *block* refers to the block number that includes the transaction, and *tx* represents the transaction index within that block.

Using these extracted read and write sets, SOLMIGRATOR then constructs a transaction dependency graph of the replayed historical transactions. Nodes in this graph represent transactions, while the directed edges between nodes represent the dependency relations between transactions. These dependency relations are essentially *write-read* relations, *i.e.*, transaction A depends on transaction B if and only if B wrote a value into the storage, which was later read by A. If multiple transactions have written to the same slot that transaction A reads, A only depends on the transaction performing the latest write (ordered by the $\langle block, tx \rangle$ index), while having no dependency relation with the others. Additionally, transactions that only have *read-read* or *write-write* relations, *i.e.*, read or write the same storage slot, do not necessarily depend on each other.

Fig. 3 shows an example of the transaction dependency graph for an ERC20 contract [25]. The graph has nine nodes,

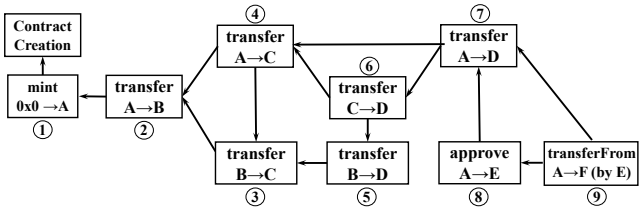

Fig. 3: An example of the transaction dependency graph

representing historical transactions 1-9, respectively. Transactions 1 mints tokens to *A*, and transactions 2-7 transfer tokens among several addresses using the ERC20 *transfer* function. In transaction 8, *A* approves *E* to spend his tokens, so that in transaction 9, *E* can successfully call the *transferFrom* function and transfer *A*'s token to *F*. The edges between nodes represent dependencies between transactions. For example, each *transfer* operation reads the balance states of the sender and receiver and updates these states accordingly. Therefore, for multiple *transfer* operations involving the same account (*e.g.*, transactions 2 and 3), the subsequent transaction reads the results of the previous one, forming a dependency relation.

C. Test Sequence Augmentation

Based on the constructed transaction dependency graph, SOLMIGRATOR transforms the entire on-chain transaction history into a set of off-chain executable test cases. Every historical transaction represents a real-world usage of a specific functionality. Therefore, by constructing off-chain test transaction sequences that faithfully replay each historical transaction, we can simulate the real-world usage performed by it and test the same functionality in an off-chain environment.

To construct test sequences that faithfully replay a target historical transaction, a trivial method would be to include every transaction from the contract creation up to the target transaction into the test sequence. However, this approach would be highly inefficient: the target transaction will often only depend on a small subset of previous transactions, rather than on all of the transactions that precede it. To achieve an efficient result, SOLMIGRATOR utilizes the transaction dependency graph to slice the entire transaction history and construct a succinct and self-contained transaction sequence. For each target transaction, SOLMIGRATOR starts with a test sequence containing only the target transaction itself. Then, it iteratively checks each transaction within the sequence to determine if all transactions it depends on are already included in the current sequence. If any transactions that a current transaction depends on, *i.e.*, preceding nodes in the transaction dependency graph, are not present in the sequence, SOLMIGRATOR adds them. This process continues until no new transactions are added to the sequence, *i.e.*, the constructed transaction sequence contains all necessary transactions to faithfully replay the target transaction off-chain.

After constructing these test transaction sequences, SOLMIGRATOR analyzes them further for de-duplication. Our intuition is that real-world transactions often repetitively call

the same function of the contract using exactly the same execution path. However, in testing smart contracts the focus is primarily targeted towards transactions that explore new execution paths. To achieve this, SOLMIGRATOR analyzes the execution of EVM opcodes and de-duplicates the test sequences based on whether the sequence leads to a new execution path of the called function. If the execution path of the transaction covers new execution branches not covered by existing test cases, SOLMIGRATOR will add it as a new test case. If multiple transaction sequences call the target function with exactly the same execution path, SOLMIGRATOR only includes the shortest one. Finally, SOLMIGRATOR generates a set of executable test transaction sequences, each covering different functionalities of the contract.

D. Test Sequence Folding

After generating a set of executable test cases, SOLMIGRATOR employs an additional process we call test sequence folding to further reduce the length of the test transaction sequences. This process is motivated by the findings of previous studies: longer test sequences are more difficult for developers to understand [26], and they may also have lower success rates of test migrations [7].

The rationale behind our test sequence folding process is that preserving all dependency relations between historical transactions may be too strict for the testing context and could lead to long test sequences. Therefore, by relaxing these strict dependency relations, it is possible to test the same functionality with a shorter transaction sequence. For example, transactions 1 to 9 in Fig. 3 form a test transaction sequence targeting the *transferFrom* function of an ERC20 contract. Within this sequence, transactions 2-7 are included to strictly maintain the dependency relations and set *A*'s balance to the latest value before transaction 9 reads it. However, these transactions, which only involve *A* transferring tokens to several irrelevant addresses, are actually not necessary for covering the target test path, *i.e.*, *E* successfully transfers tokens from *A* to *F* in transaction 9. Therefore, to generate meaningful test cases rather than strictly replay historical transactions, these transactions can be removed from the sequence.

During the folding process, SOLMIGRATOR iterates through the transaction sequence from back to front and tries to "relax" dependency relations of each transaction. Given a dependency relation between the current transaction and a previous transaction, SOLMIGRATOR analyzes the storage slots that cause this dependency and searches for earlier alternative transactions that have written to these same slots. If such alternative transactions exist, SOLMIGRATOR will try to reconstruct the dependency graph by updating the dependency relation from the current transaction to the originally dependent transaction to a new relation between the current transaction and that alternative transaction. If multiple alternatives are available, SOLMIGRATOR prioritizes the earliest transaction as the new dependent transaction. SOLMIGRATOR then reconstructs the transaction sequences based on the updated relations and executes the reconstructed sequence to check whether the relaxation changes the execution path of the target test transaction. If the execution path remains unchanged, the relaxations of dependencies will be preserved. Otherwise, SOLMIGRATOR reverts them and tries to relax other dependencies of the current transaction. The process of reconstructing dependency relations continues for each transaction, until SOLMIGRATOR reaches the start of the transaction sequence.

As an example, to fold the transaction sequence shown in Fig. 3, SOLMIGRATOR first analyzes the dependencies of transaction 9. This includes transaction 7, due to its write to *A*'s balance, and transaction 8, due to its write to the allowance state. Then, SOLMIGRATOR searches for alternative transactions that write the same slots (*i.e.*, *A*'s balance) as transaction 7 and identifies transactions 1, 2, 4 as the result. Since transaction 1 is the earliest alternative transaction, SOLMIGRATOR replaces the dependency of transaction 9 on transaction 7 with a new dependency on transaction 1, and reconstructs the transaction sequence. This reconstruction removes transaction 7, along with transactions 2-6, which are not depended on by any remaining transactions in the sequence. The validity of the reconstructed sequence is confirmed as the execution path of the target transaction, *i.e.*, transaction 9, remains unchanged. Since there are no alternative transactions to replace the writes made by transaction 8, SOLMIGRATOR maintains this dependency and completes the folding process. As a result, the original sequence of transactions 1-9 is folded into a new sequence of transactions 1, 8, and 9.

E. Assertion Augmentation

A test case for a smart contract includes both the sequence of test transactions to be executed and the test assertions to be checked. These test assertions specify the smart contract's intended behaviors during test case execution. Therefore, after generating the test sequences, SOLMIGRATOR further executes these sequences and extracts assertions from their execution traces. To do this, SOLMIGRATOR augments the following three categories of assertions, which have been commonly used to characterize the execution behaviors of smart contracts by existing testing frameworks [4], [27].

Status-Based Assertions: SOLMIGRATOR augments assertions that specify the expected execution status for each transaction. If a transaction executes successfully, SOLMIGRATOR augments a success assertion, *i.e.*, *expect(tx).to.not.revert*. Otherwise, SOLMIGRATOR augments a failure assertion, *i.e.*, *expect(tx).to.revert*.

Event-Based Assertions: SOLMIGRATOR augments event-based assertions for the transactions that emit events. It extracts the event's name and parameters from the transaction receipts and the contract ABI, and then encodes them into an event-based assertion in the form of *expect(tx).to.emit(...).withArgs(...)*. If the same transaction emits more than one event, SOLMIGRATOR will generate one assertion for each of them.

Return-Value-Based Assertions: For each transaction that has a return value, SOLMIGRATOR will generate a return-value-

based assertion, which will check the transaction return value using statements like *expect(txResult).to.equal()*.

After generating the test sequence assertions, SOLMIGRATOR finally transforms the on-chain historical transactions into off-chain executable test cases.

V. TEST MIGRATION

The test migration process aims to migrate the test cases augmented in Section IV from the source contract to the target contract. Specifically, it involves three steps, *i.e.*, test transaction sequence migration, test assertion migration and test script generation.

A. Test Transaction Sequence Migration

Each transaction in the test transaction sequence is a call to a specific function of the source contract. Therefore, to migrate the test transaction sequence from the source contract to the target contract, we first need to match the functions in the source contract with those in the target contract.

1) Function Matching: Our function matching process starts with matching functions with the same function selectors [28], *i.e.*, functions with the same function name and input parameter types. Previous studies have shown that due to prevalent code reuse practices, smart contracts with common functionalities often share a large portion of the same underlying functions [15]. For example, in ERC20 contracts [18], functions with signature *transfer(address, uint256)* typically manage the functionality for transferring tokens. These functions can be directly matched based on function selectors.

SOLMIGRATOR then uses these already matched functions as matching "anchors" and matches the remaining functions based on their relations to these anchors. To do this, SOLMIGRATOR focuses on two types of relations – call relations and data dependency relations – that characterize the control/data flow dependencies among functions. For call relations, SOLMIGRATOR traverses the AST (Abstract Syntax Tree) [29] of each function and analyzes the inter-function call statements in it. For data dependencies, SOLMIGRATOR extracts each function's read and write operations on contract storage variables and records data flow dependencies where a function reads a variable that another function has written.

Based on these analyses, SOLMIGRATOR attempts to match source contract functions that have not yet been matched to corresponding functions in target contracts. If a target function has the same input interface as a source function and they both have control/data dependencies with a pair of already matched functions, SOLMIGRATOR will establish a match between them. If a source function has multiple potential matches in the target contract, SOLMIGRATOR prioritizes the target function that shares the highest number of related functions with the source function.

2) Transaction Sequence Migration: Based on matching relations between functions in source and target smart contracts, SOLMIGRATOR migrates each transaction in the source test sequence to a transaction for the target contract. There are two types of transactions in a test sequence, *i.e.*, the contract deployment transaction and other transactions that call the deployed contract. They are migrated as follows:

Contract Deployment Transactions: The input data of these transactions includes the creation bytecode of the contract and the value of the contract constructor parameters. The contract creation bytecode is directly obtained by compiling the target contract, while the constructor parameters are instantiated by matching and migrating constructor arguments in the source contract's deployment transaction. SOLMIGRATOR first identifies functions influenced by each constructor parameter. It then conducts dynamic taint analysis on both the source and target contracts, setting constructor parameters as taint sources and the state read operations in each function as sinks. SOLMIGRATOR then matches the constructor parameters between the source and target contracts based on whether they can influence the same function. If a target parameter and a source parameter share the same type, and the functions they influence include one or more pairs of matched functions, SOLMIGRATOR considers them matched and uses the value of the source parameter to instantiate the target parameter. If no match is found, SOLMIGRATOR randomly generates the target parameter's value based on its type.

Other Transactions: For subsequent transactions that call the deployed contract, the input data includes a function selector that specifies the function to be called and the value of the function parameters. To migrate them, SOLMIGRATOR replaces the function selector in the source transaction with the selector of the matched target function and instantiates the parameters based on their values in the source transaction.

Note that test migration is different from test generation techniques such as fuzzing, primarily in that it focuses on migrating test transactions existing in the source contract to the target contract. Therefore, SOLMIGRATOR will not insert newly generated test transactions into the testing sequence. If a source test sequence tests a function that does not have any counterpart in the target contract, it will not be migrated to the target contract. Automatically generating additional test transactions that do not exist in the source contract is beyond the scope of this paper and is potential future work.

B. Test Assertion Migration

After migrating test transaction sequences, SOLMIGRATOR further migrates test assertions from the source to the target smart contract. To migrate *status-based* assertions, SOLMIGRATOR enumerates each source transaction, finds its counterpart in the migrated sequence, and generates the same assertion for the counterpart. To migrate *return-value-based* assertions, SOLMIGRATOR analyzes the ABIs (Application Binary Interfaces) [30] of the source and target contracts to extract the return value types of all functions. If a source function has the same return value types as its counterpart in the target contract, SOLMIGRATOR migrates its *return-value-based* assertions to the target transactions calling its counterpart.

To migrate *event-based* assertions, SOLMIGRATOR first matches the event in the source and target contracts. It tra-

verses the contract's AST to locate event-emitting statements and records the functions that emit each event. If two events have the same parameter types and the functions that emit them include one or more pairs of matched functions, these events are matched. SOLMIGRATOR then migrates event assertions based on these matching relations. If a source transaction emits an event that has a matched event in the target contract, its event assertion will be migrated from the source transaction to its counterpart in the target transaction sequence.

C. Test Script Generation

Finally, after generating assertions for the target test sequences, SOLMIGRATOR creates complete and executable test cases for the target smart contract. It then runs these test cases and checks their execution results on the target contract. If the migrated test cases can successfully execute on the target contract, SOLMIGRATOR packages them into test scripts compatible with the Hardhat testing framework [4]. We choose Hardhat because it is one of the major test frameworks for smart contracts [31]. Our SOLMIGRATOR approach can also be adapted to other testing frameworks, such as Truffle [5] and Foundry [27], with minor adjustments.

If a migrated test case fails on the target contract, it will not be output as a migrated test case by SOLMIGRATOR. This is because a test case passing on the source contract but failing on the target contract indicates that the functionality being tested differs between the source and target contracts. Without additional information from developers, SOLMIGRATOR cannot determine whether these functional differences are intended. These test cases will be marked as migrated but failed test cases. Developers can then choose to review these migrated but failed test cases and decide whether to incorporate them into the testing process.

VI. EMPIRICAL EVALUATION

We evaluate SOLMIGRATOR's effectiveness in augmenting and migrating test cases for real-world smart contracts.

A. Experimental Setup

Research Questions. Specifically, we focus on the following three research questions:

- **RQ1.** Can SOLMIGRATOR effectively augment test cases from existing on-chain smart contracts?
- **RQ2.** Can SOLMIGRATOR effectively migrate the augmented test cases from the source to the target contracts?
- **RQ3.** Can test cases migrated by SOLMIGRATOR effectively cover common functionalities of the target contract?

Dataset. To answer these research questions, we collected real-world smart contracts within two smart contract categories, ERC20 and ERC721, and evaluated SOLMIGRATOR using them. These two categories are currently the most popular and influential categories on Ethereum, comprising over 1,200,000 contracts with a market capitalization exceeding 500 billion dollars [32]. To collect contracts for these two categories, we queried the public *crypto_ethereum* dataset provided by Google BigQuery [33] and extracted smart contracts with ERC20 and ERC721 labels. We collected 165,665 ERC20 contracts and 3,481 ERC721 contracts in total. To facilitate our analysis, we refined these contracts by filtering for contracts with available source code and over 1,000 historical transactions, resulting in a final experimental dataset of 2,829 ERC20 contracts and 108 ERC721 contracts.

To retrieve contracts' historical transactions and states, we maintained an Ethereum archive node [24] to record full Ethereum on-chain states. Before our experiments, we fetched the historical transactions for each contract in the dataset and the execution traces of these transactions. Then, we conducted the experiments on a machine with Intel Core i9 CPU (3.0GHz), 64 GB RAM, and running Windows WSL.

The source code of SOLMIGRATOR, our datasets and experiment results are all available in the online artifacts [34].

B. RQ1: Effectiveness in Augmenting Test Cases

In RQ1, we evaluate the effectiveness of SOLMIGRATOR in extracting test cases from historical transactions of on-chain smart contracts. In line with previous studies [35], [36], we randomly sampled a set of smart contracts for each category to facilitate the analysis. A total of 93 contracts from the ERC20 category and 51 from the ERC721 category were sampled, achieving a confidence level of 95% with a confidence interval of 10%. Then, for each of these contracts, we collected the first 1,000 transactions since its deployment and used SOLMIGRATOR to augment test cases based on them. As a result, SOLMIGRATOR augmented **an average of 5.1 test cases for each ERC20 contract and 9.6 test cases for each ERC721 contract,** demonstrating its effectiveness in extracting off-chain test cases from on-chain transactions. These augmented test cases, with an average of 4.3 test transactions and 5.0 assertions for ERC20 contracts, and an average of 14.9 test transactions and 29.0 assertions for ERC721 contracts, can be efficiently integrated into the testing process.

Furthermore, we evaluated the impact of the folding process on the test transaction sequence length by comparing the lengths of the augmented test transaction sequences before and after folding. We found that for 18% of ERC20 test cases and 37% of ERC721 test cases, the lengths of the test transaction sequences were reduced by the test folding process, with the average test case length reduced to 35% of the original. This indicates that the folding process can help SOLMIGRATOR test the same functionality with fewer test transactions, demonstrating its effectiveness.

We also investigated the impact of the number of historical transactions on the number of augmented test cases. Specifically, we used SOLMIGRATOR to generate test cases based on different numbers of historical transactions and analyzed the number of augmented test cases. As shown in Fig. 4, there is a rapid increase in the number of augmented test cases from 1 to 400 transactions, followed by a much slower growth rate after 400 transactions. This result may be attributed to the observation that real-world historical transactions often focus on the repetitive use of particularly common contract functionalities, thus not introducing new test cases. This indicates that

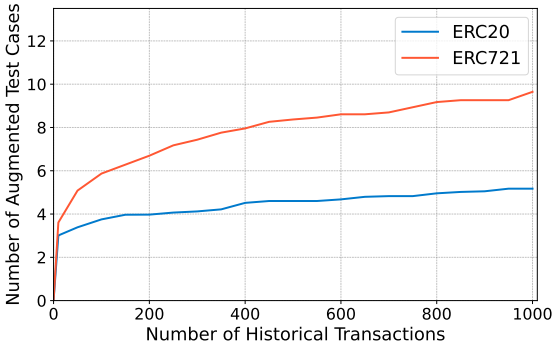

Fig. 4: The average number of augmented test cases with different number of historical transactions.

TABLE I: Statistics of the benchmark contracts

ERC20 Category			ERC721 Category		
ID	**Name**	**# Tests**	**ID**	**Name**	**# Tests**
A0	TetherToken	11	B0	BoredApeYachtClub	12
A1	BNB	6	B1	Miladys	17
A2	SHIBA	5	B2	Doodles	16
A3	TRX	9	B3	CoolCats	11
A4	IMXToken	6	B4	Azuki	6
A5	WBTC	12	B5	MutantApeYachtClub	6
A6	FetchToken	7	B6	Mfers	9
A7	PepeToken	14	B7	CoolmansUniverse	14
A8	CroToken	7	B8	IO	10
A9	GraphToken	7	B9	PudgyPenguins	6

in practice, using 1,000 transactions could efficiently cover commonly used functionalities of the contract and augment corresponding test cases.

C. RQ2: Effectiveness in Migrating Test Cases

In RQ2, we evaluated SOLMIGRATOR's effectiveness in migrating the augmented test cases. Specifically, we used SOLMIGRATOR to migrate test cases for a set of source-target contract pairs and manually analyzed the migration results. To ensure real-world significance, we chose ten contracts for each category based on the Etherscan Top Token list [37] and Top NFT list [38]. These contracts were chosen based on the number of transactions, excluding those whose primary function is not token management (*e.g.*, auctions) and those with external dependencies (*e.g.*, proxy contracts). Then, we migrated test cases among these contracts within the same category by setting each contract as the source and the remaining contracts as targets. This results in 180 source-target migration pairs (10 targets $\times$ 9 sources $\times$ 2 categories), with a total of 1,719 attempted test case migrations for evaluating SOLMIGRATOR. Table I shows the IDs, names, and number of augmented test cases for these benchmark contracts. Detailed information about the benchmarks and migration results can be found in our online supplementary material [34].

We manually analyzed the migration results for each test case to evaluate the effectiveness of SOLMIGRATOR's migration process. During the manual inspection, we classified test cases that are successfully migrated and executed on the target contract as positives, and those that failed to be migrated or executed as negatives. For the positive cases, we further classified them into True Positives (TPs) and False Positives (FPs) based on whether the migrated test cases correctly match their counterparts in the target smart contract. For the negative cases, we further classified them into False Negatives (FNs) and True Negatives (TNs) based on whether there is a counterpart in the target contract but SOLMIGRATOR failed to migrate the test cases.

Table II shows the breakdown results for each source-target migration pair. Specifically, the *From* and *To* columns show the ID of source and target smart contracts. The *Migrate* column shows the proportion of test cases successfully migrated and executed on the target contract, *i.e.*, positive rate. The remaining columns show the proportion of TPs, FPs, TNs, FNs of the migration results of each source-target pair, respectively. In summary, **SOLMIGRATOR achieved an average precision of 96.3%, recall of 92.8%, and accuracy of 93.6%,** demonstrating its effectiveness. It migrated an average of 58.1% and 56.3% test cases from source contracts to target contracts in ERC20 and ERC721 categories, respectively, which corresponds to 56.9% and 53.3% of test cases being correctly matched (TPs) and 1.2%, 2.9% being incorrectly matched (FPs). For the test cases not migrated, the true negative rates were 38.0% for ERC20 and 39.0% for ERC721, with 3.8% and 4.7% false negative rates, respectively.

The migration results show a relatively high proportion of true negatives. In line with existing test migration studies for other software [39], [40], such results are mainly influenced by the differences among the benchmark smart contracts, rather than the technique itself. After manually analyzing them, we found that in 68.1% of cases, there are no existing counterparts in the target contract, directly preventing SOLMIGRATOR from matching test cases from the source contract to the target contract. In the remaining 31.9%, the test cases from the source contract can be matched to the target contract, but they fail to execute in the target contract due to functional differences between the source and target contracts. For example, a contract might not allow transfers between users until the owner enables this function, causing migrated test cases that directly perform transfers between users to fail on this contract. Note that although these test cases failed to be directly migrated, they may still become useful after minor modifications by developers.

Our manual analysis revealed large variations in migration rates among different pairs of smart contracts. For example, the migration rate from contract A4 to A6 is 100%, whereas it is only 33% from A4 to A1. The main reason is that, although these contracts are within the same category, there can still be notable functional differences among them. For example, A4 and A6 allow mint operations, *i.e.*, the contract owner can mint tokens for users, while A1 does not. These differences could lead to different usage patterns of the contracts, such as how users initially obtain tokens, making test cases migrated from A4 fail on A1. These results support the intuition that migrating test cases between more similar smart contracts can result in a higher migration rate.

TABLE II: Migration results of test cases for ERC20 (A) and ERC721 (B) contracts by SOLMIGRATOR

From	To	Migrate	TP	FP	TN	FN	From	To	Migrate	TP	FP	TN	FN	From	To	Migrate	TP	FP	TN	FN
A0	A1	45%	45%	0%	55%	0%	A0	A2	55%	55%	0%	45%	0%	A0	A3	45%	45%	0%	55%	0%
A0	A4	45%	45%	0%	55%	0%	A0	A5	55%	55%	0%	45%	0%	A0	A6	55%	55%	0%	45%	0%
A0	A7	64%	64%	0%	36%	0%	A0	A8	55%	55%	0%	45%	0%	A0	A9	64%	64%	0%	36%	0%
A1	A0	83%	83%	0%	17%	0%	A1	A2	100%	83%	17%	0%	0%	A1	A3	50%	50%	0%	50%	0%
A1	A4	50%	50%	0%	50%	0%	A1	A5	50%	50%	0%	50%	0%	A1	A6	83%	83%	0%	17%	0%
A1	A7	83%	67%	17%	17%	0%	A1	A8	50%	50%	0%	50%	0%	A1	A9	100%	83%	17%	0%	0%
A2	A0	100%	100%	0%	0%	0%	A2	A1	80%	80%	0%	20%	0%	A2	A3	60%	60%	0%	40%	0%
A2	A4	60%	60%	0%	40%	0%	A2	A5	60%	60%	0%	40%	0%	A2	A6	100%	100%	0%	0%	0%
A2	A7	100%	100%	0%	0%	0%	A2	A8	60%	60%	0%	40%	0%	A2	A9	100%	100%	0%	0%	0%
A3	A0	56%	56%	0%	44%	0%	A3	A1	33%	33%	0%	44%	22%	A3	A2	78%	78%	0%	22%	0%
A3	A4	78%	78%	0%	0%	22%	A3	A5	56%	56%	0%	22%	22%	A3	A6	78%	78%	0%	0%	22%
A3	A7	67%	67%	0%	33%	0%	A3	A8	67%	67%	0%	22%	11%	A3	A9	78%	78%	0%	0%	22%
A4	A0	100%	83%	17%	0%	0%	A4	A1	33%	33%	0%	67%	0%	A4	A2	50%	50%	0%	50%	0%
A4	A3	50%	50%	0%	0%	50%	A4	A5	100%	100%	0%	0%	0%	A4	A6	100%	100%	0%	0%	0%
A4	A7	67%	50%	17%	33%	0%	A4	A8	67%	67%	0%	33%	0%	A4	A9	100%	100%	0%	0%	0%
A5	A0	42%	42%	0%	50%	8%	A5	A1	25%	25%	0%	75%	0%	A5	A2	33%	33%	0%	67%	0%
A5	A3	33%	33%	0%	67%	0%	A5	A4	33%	33%	0%	67%	0%	A5	A6	33%	33%	0%	67%	0%
A5	A7	50%	50%	0%	50%	0%	A5	A8	50%	50%	0%	50%	0%	A5	A9	50%	50%	0%	50%	0%
A6	A0	57%	57%	0%	43%	0%	A6	A1	57%	57%	0%	43%	0%	A6	A2	57%	57%	0%	43%	0%
A6	A3	29%	29%	0%	71%	0%	A6	A4	29%	29%	0%	71%	0%	A6	A5	29%	29%	0%	71%	0%
A6	A7	43%	43%	0%	57%	0%	A6	A8	43%	43%	0%	57%	0%	A6	A9	57%	57%	0%	43%	0%
A7	A0	79%	79%	0%	14%	7%	A7	A1	71%	71%	0%	29%	0%	A7	A2	79%	79%	0%	21%	0%
A7	A3	29%	29%	0%	71%	0%	A7	A4	29%	29%	0%	71%	0%	A7	A5	29%	29%	0%	71%	0%
A7	A6	79%	79%	0%	21%	0%	A7	A8	36%	36%	0%	43%	21%	A7	A9	86%	79%	7%	7%	7%
A8	A0	57%	57%	0%	14%	29%	A8	A1	14%	14%	0%	86%	0%	A8	A2	43%	43%	0%	57%	0%
A8	A3	29%	29%	0%	71%	0%	A8	A4	29%	29%	0%	71%	0%	A8	A5	57%	43%	14%	14%	29%
A8	A6	29%	29%	0%	43%	29%	A8	A7	43%	43%	0%	57%	0%	A8	A9	43%	43%	0%	57%	0%
A9	A0	71%	71%	0%	29%	0%	A9	A1	57%	57%	0%	43%	0%	A9	A2	57%	57%	0%	43%	0%
A9	A3	43%	43%	0%	57%	0%	A9	A4	43%	43%	0%	57%	0%	A9	A5	57%	57%	0%	43%	0%
A9	A6	57%	57%	0%	0%	43%	A9	A7	71%	71%	0%	29%	0%	A9	A8	57%	57%	0%	43%	0%
Average		58.1%	56.9%	1.2%	38.0%	3.8%	Precision: 97.9%, Recall: 93.7%, Accuracy: 94.9%													

From	To	Migrate	TP	FP	TN	FN	From	To	Migrate	TP	FP	TN	FN	From	To	Migrate	TP	FP	TN	FN
B0	B1	58%	58%	0%	42%	0%	B0	B2	33%	33%	0%	50%	17%	B0	B3	33%	25%	8%	50%	17%
B0	B4	33%	33%	0%	67%	0%	B0	B5	67%	58%	8%	33%	0%	B0	B6	67%	50%	17%	17%	17%
B0	B7	33%	33%	0%	50%	17%	B0	B8	33%	33%	0%	67%	0%	B0	B9	33%	25%	8%	67%	0%
B1	B0	76%	76%	0%	24%	0%	B1	B2	41%	41%	0%	59%	0%	B1	B3	35%	35%	0%	65%	0%
B1	B4	41%	35%	6%	59%	0%	B1	B5	65%	65%	0%	35%	0%	B1	B6	59%	59%	0%	6%	35%
B1	B7	41%	41%	0%	18%	41%	B1	B8	41%	41%	0%	59%	0%	B1	B9	41%	41%	0%	59%	0%
B2	B0	50%	50%	0%	12%	38%	B2	B1	50%	50%	0%	12%	38%	B2	B3	44%	44%	0%	25%	31%
B2	B4	31%	31%	0%	69%	0%	B2	B5	50%	50%	0%	12%	38%	B2	B6	62%	62%	0%	38%	0%
B2	B7	88%	88%	0%	12%	0%	B2	B8	31%	31%	0%	69%	0%	B2	B9	38%	25%	12%	62%	0%
B3	B0	64%	45%	18%	36%	0%	B3	B1	64%	45%	18%	36%	0%	B3	B2	36%	36%	0%	64%	0%
B3	B4	27%	27%	0%	73%	0%	B3	B5	55%	45%	9%	45%	0%	B3	B6	45%	36%	9%	55%	0%
B3	B7	36%	36%	0%	64%	0%	B3	B8	27%	27%	0%	73%	0%	B3	B9	45%	27%	18%	55%	0%
B4	B0	67%	67%	0%	33%	0%	B4	B1	83%	83%	0%	17%	0%	B4	B2	67%	67%	0%	33%	0%
B4	B3	67%	67%	0%	33%	0%	B4	B5	83%	83%	0%	17%	0%	B4	B6	67%	67%	0%	33%	0%
B4	B7	67%	67%	0%	33%	0%	B4	B8	67%	67%	0%	33%	0%	B4	B9	67%	67%	0%	33%	0%
B5	B0	67%	67%	0%	0%	33%	B5	B1	100%	100%	0%	0%	0%	B5	B2	50%	50%	0%	50%	0%
B5	B3	67%	67%	0%	33%	0%	B5	B4	50%	50%	0%	50%	0%	B5	B6	67%	67%	0%	33%	0%
B5	B7	50%	50%	0%	50%	0%	B5	B8	67%	67%	0%	33%	0%	B5	B9	50%	50%	0%	50%	0%
B6	B0	89%	89%	0%	11%	0%	B6	B1	89%	89%	0%	11%	0%	B6	B2	89%	89%	0%	11%	0%
B6	B3	78%	78%	0%	11%	11%	B6	B4	67%	67%	0%	33%	0%	B6	B5	89%	89%	0%	11%	0%
B6	B7	100%	100%	0%	0%	0%	B6	B8	67%	67%	0%	33%	0%	B6	B9	67%	67%	0%	22%	11%
B7	B0	64%	50%	14%	21%	14%	B7	B1	64%	50%	14%	21%	14%	B7	B2	43%	43%	0%	57%	0%
B7	B3	57%	57%	0%	7%	36%	B7	B4	43%	43%	0%	57%	0%	B7	B5	64%	50%	14%	21%	14%
B7	B6	43%	43%	0%	57%	0%	B7	B8	64%	64%	0%	36%	0%	B7	B9	43%	36%	7%	57%	0%
B8	B0	80%	70%	10%	20%	0%	B8	B1	70%	50%	20%	30%	0%	B8	B2	40%	40%	0%	60%	0%
B8	B3	50%	50%	0%	50%	0%	B8	B4	40%	40%	0%	60%	0%	B8	B5	80%	60%	20%	20%	0%
B8	B6	70%	50%	20%	30%	0%	B8	B7	50%	40%	10%	50%	0%	B8	B9	40%	40%	0%	60%	0%
B9	B0	50%	50%	0%	50%	0%	B9	B1	50%	50%	0%	50%	0%	B9	B2	50%	50%	0%	50%	0%
B9	B3	83%	83%	0%	17%	0%	B9	B4	33%	33%	0%	67%	0%	B9	B5	50%	50%	0%	50%	0%
B9	B6	50%	50%	0%	50%	0%	B9	B7	50%	50%	0%	50%	0%	B9	B8	33%	33%	0%	67%	0%
Average		56.3%	53.3%	2.9%	39.0%	4.7%	Precision: 94.7%, Recall: 91.9%, Accuracy: 92.3%													

Furthermore, the migration results suggest SOLMIGRATOR's abilities in revealing functional differences between the source and target smart contracts, which could have implications for bug-finding, differential testing, and design choice refining. Specifically, in 31.9% (12.3% of all migration cases) of true negatives, the test cases from the source contract can be correctly matched, but fail to execute in the target contract. The manual investigation of these cases demonstrates that some of them can reveal non-standard and potentially defective implementations in the target contracts. For example, the PepeToken contract disables token transfers by default and allows the contract owner to enable or disable them arbitrarily. However, previous research [41] has identified such control as a potential risk for *Rug Pull* attacks: If the contract owner unintentionally or maliciously disables transfers, users are prevented from selling their tokens, resulting in direct financial losses. Unlike the PepeToken contract, other ERC20 smart contracts do not require additional steps to enable transfers. Therefore, the test cases migrated from these contracts will attempt direct token transfers without requiring other operations.

During the migration process, these direct transfer operations will be reverted by the PepeToken contract, leading to failures in the migration process. Developers can further analyze these failures to determine whether the implementation that allows the owner to arbitrarily disable or enable transfers is a bug or an intentional design choice. Please refer to our online supplement material for more detailed descriptions [34].

D. RQ3: Effectiveness of the Migrated Test Cases

While RQ2 demonstrates that the augmented test cases can be accurately migrated from the source to the target contracts, it does not investigate their effectiveness for the target contract. Therefore, we further introduced RQ3 to check whether the migrated test cases can cover commonly used functions of the target contract.

To achieve this, we executed real-world transactions of each target contract and the test cases migrated to it, and compared the functions called by them. Note that since SOLMIGRATOR is the first migration technique for smart contracts, there are no available tools for comparison. However, the baselines used in the experiments, *i.e.*, real-world transactions of the target contract, are actually more suitable for our evaluation, because practical real-world usage is precisely what SOLMIGRATOR aims to cover. We first replayed the transactions of each target contract in Table I and recorded the functions called by each transaction. In line with Section VI-B, the number of analyzed transactions is 1,000. After that, we executed the test cases migrated from each source contract and analyzed the functions they tested. Then, we chose the migration source that achieves the highest function coverage, along with the test cases migrated from it, for comparison with real-world transactions of each target contract. Since developers typically perform test migration between contracts that are as similar as possible, choosing the most suitable source contract better simulates the real-world application scenarios of test migration.

The results are shown in Table III. The *Target* column is the target contract that the test cases are migrated to, and the *Source* column is the migration source. The *Func* column shows the number of functions covered by the migrated test cases. The *Tx* column shows the proportion of real-world transactions that called these covered transactions. Our results show that **for 13 out of 20 (65.0%) target contracts, the migrated test cases can cover the functions used in over 85% of real-world transactions.** This indicates that after migration, test cases extracted from the real-world usage of the source contract can also effectively cover the common usage of the target contract.

After inspecting the remaining contracts, we found that they have several commonly used functions specific to these contracts, which prevents SOLMIGRATOR from migrating test cases from other contracts to cover these functions. For example, the most frequently used function in the contract CoolmanUniverse is the *mintAllowlist* function, which requires users to provide a Merkle proof before minting. Since all other smart contracts do not have similar functions, no test cases for this functionality are migrated. However, considering the non-exhaustive set of migration source contracts, the results in Table III are only lower bounds: it is possible to cover these uncovered functions by migrating test cases from other on-chain source contracts. In practice, developers can rely on existing similarity-based search tools [42], [43] and services [44], to select more suitable migration sources and achieve better migration results.

TABLE III: Migrated test cases vs. real-world transactions

ERC20 Category				ERC721 Category			
Target	**Source**	**#Func**	**Tx(%)**	**Target**	**Source**	**#Func**	**Tx(%)**
A0	A9	4	99.2%	B0	B1	8	89.1%
A1	A6	4	99.9%	B1	B5	6	94.1%
A2	A3	6	100.0%	B2	B6	5	17.0%
A3	A4	3	53.7%	B3	B7	6	11.4%
A4	A3	6	96.7%	B4	B0	4	1.7%
A5	A4	6	97.5%	B5	B1	7	99.7%
A6	A4	6	99.7%	B6	B2	6	99.7%
A7	A1	4	99.6%	B7	B2	7	28.8%
A8	A5	5	2.8%	B8	B7	7	20.0%
A9	A0	5	99.8%	B9	B3	4	99.9 %

VII. DISCUSSION

A. Implications

Our study has demonstrated that it is viable to extract and migrate test cases from real-world usages of existing smart contracts and use them to test other newly developed smart contracts. Considering the commonly shared functionalities among smart contracts [14], SOLMIGRATOR could have practical implications across broad and realistic application scenarios. For example, it can be used when developing and testing smart contracts in a field that has several established on-chain projects to refer to. While developers often reuse the code of established contracts to enhance development efficiency [14], SOLMIGRATOR enables them to also reuse the human knowledge from real-world usages of these contracts to enhance testing efficiency. Beyond migrating test cases across different contracts, SOLMIGRATOR can also be applied to different versions of the same contract, such as testing a new contract intended to upgrade an existing on-chain contract.

By using SOLMIGRATOR, developers can reduce the effort required for manually writing test cases for new contracts. They can identify similar on-chain contracts through their knowledge or using similarity-based search tools [42], [43] and services [44], and then use SOLMIGRATOR to migrate test cases from these contracts. These migrated test cases enable developers to efficiently simulate potential real-world usages of contracts and confirm their functional correctness, thus reducing the manual effort required in testing processes. Beyond ensuring functional correctness, test cases migrated from established smart contracts can also assist developers in refining their design choices, such as suggesting missing features or improving usage patterns for new smart contracts.

B. Threats to Validity

The main internal threat to validity is the potential mistakes in the manual analysis of the evaluation results. To mitigate the threat, we employed a double-checking process and carefully reviewed the manual label results. We have made

these results publicly available for external inspection [34]. The main external threat to validity lies in the selection of our datasets. To ensure real-world relevance, our dataset includes two most popular categories of smart contracts. These have well-established datasets [33] and have been commonly used in previous studies [45]–[47]. The size of our dataset is also adequate, compared to previous studies on test case migration [39], [40].

C. Limitations

As the first test migration tool for smart contracts, SOLMIGRATOR uses a matching process based on function signature matching and static analysis. Although its effectiveness is justified by evaluations on applications with similar specifications, it can be improved to better handle diverse functions in more complex scenarios. We plan to enhance SOLMIGRATOR's matching process by integrating it with LLMs for improved semantic understanding in the future. Additionally, SOLMIGRATOR's test augmentation process, which uses execution paths for test case selection and deduplication, could filter out potentially interesting test cases. We plan to implement a more fine-grained selection approach to better balance the number and representativeness of the augmented test cases. Furthermore, in line with previous test migration tools [39], [40], SOLMIGRATOR currently employs a one-to-one migration process, *i.e.*, from one specific source contract to one target contract. Although existing similarity-based search tools [42], [44] can help developers in selecting a suitable migration source, the non-exhaustive nature of the one-to-one migration process might lead to potentially sub-optimal results, impacting the effectiveness of test case generation. Future studies could improve SOLMIGRATOR to support many-to-one migration, where multiple migration sources can be automatically identified through similarity-based search techniques.

VIII. Related Work

A. Test Migration for Other Software

Several previous studies have focused on migrating test cases between other software with similar functionalities, such as mobile apps and web applications [39], [48]–[50]. For example, Behrang *et al.* [48] propose GUITestMigrator to transfer test cases across mobile apps that follow the same specifications. This tool allows tests developed for one app to be migrated to other apps, thereby helping to automate the assessment of mobile app coding assignments. AppTestMigrator [6] extends the functionality of GUITestMigrator by employing static analysis techniques to enable test migration between mobile apps that share only part of their functionality.

Our work differs from them in two main aspects. First, our work is the first migration-based technique in the new context of smart contracts. It incorporates a set of static analysis techniques tailored for smart contracts, which could inherently differ from those used in mobile apps. Second, unlike existing techniques that can only transfer already developed test cases, SOLMIGRATOR does not require manually developed test cases as inputs. Instead, it can automatically extract test cases from millions of on-chain contracts, thus could be applicable to more test case generation scenarios.

B. Test Generation Techniques for Smart Contracts

With the rapid development and emerging applications of smart contracts [51]–[53], there has been a substantial body of previous studies [54], [55] focusing on using test generation techniques like fuzzing to detect security vulnerabilities in smart contracts, such as re-entrancy [11]. They typically introduce pre-defined detection patterns for each vulnerability and monitor the contract's underlying behavior during execution to determine whether these vulnerabilities are triggered. For example, ContractFuzzer [56] is one of the first fuzzing-based techniques for smart contracts. It introduces detection patterns for seven types of vulnerabilities and employs fuzzing to generate test cases to exploit them. Smartian [57] incorporates static and dynamic data-flow analyses into smart contract fuzzing, in order to detect security vulnerabilities in hard-to-reach branches. However, the primary goal of these existing test generation techniques is to produce a large number of random test inputs to detect certain types of vulnerabilities, which is orthogonal to our goal of generating a set of expressive and function-relevant test cases that represent canonical usages of the smart contracts.

IX. Conclusion and Future Work

We have proposed SOLMIGRATOR, the first migration-based technique that generates test cases for smart contracts by extracting and migrating the on-chain usages of existing smart contracts with similar functionalities. We incorporated a set of new approaches in SOLMIGRATOR, including test case augmentation based on on-chain transaction replay and dependency analysis, and test case migration based on fine-grained static analysis. Our empirical evaluation provides initial, yet promising, evidence for the feasibility of using test cases extracted and migrated from existing on-chain contracts to test new smart contracts. Experimental results show that SOLMIGRATOR can effectively extract and migrate test cases, achieving an accuracy of 93.6% in migrating 1,719 augmented test cases. Moreover, it demonstrates that the test cases migrated from source contracts can effectively represent the potential usage of the target contracts. As future work, we are planning to conduct empirical studies using more smart contracts and a larger number of test cases to confirm our initial results. We also plan to incorporate techniques such as similarity-based search to improve the usability and effectiveness of SOLMIGRATOR.

Acknowledgement

This work is partially supported by Primary Research & Developement Plan of Jiangsu Province (BE2023025, BE2023025-5), National Natural Science Foundation of China (62202011, 62172010, 62332004), and the Open Research Fund of The State Key Laboratory of Blockchain and Data Security, Zhejiang University. John Grundy is supported by ARC Laureate Fellowship FL190100035. We also thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges," *Empirical Software Engineering*, vol. 26, no. 6, p. 117, 2021.

[2] Z. Wan, X. Xia, D. Lo, J. Chen, X. Luo, and X. Yang, "Smart contract security: a practitioners' perspective," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1410–1422.

[3] L. Palechor and C.-P. Bezemer, "How are solidity smart contracts tested in open source projects? an exploratory study," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, 2022, pp. 165–169.

[4] Hardhat, "Ethereum development environment for professionals," 2024. [Online]. Available: https://hardhat.org/

[5] R. Verma, N. Dhanda, and V. Nagar, "Application of truffle suite in a blockchain environment," in *Proceedings of Third International Conference on Computing, Communications, and Cyber-Security: IC4S 2021*. Springer, 2022, pp. 693–702.

[6] F. Behrang and A. Orso, "Test migration between mobile apps with similar functionality," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 54–65.

[7] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 42–53.

[8] M. Ye, Y. Nan, Z. Zheng, D. Wu, and H. Li, "Detecting state inconsistency bugs in dapps via on-chain transaction replay and fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 298–309.

[9] C. Shou, S. Tan, and K. Sen, "Ityfuzz: Snapshot-based fuzzer for smart contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 322–333.

[10] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[11] Z. Wang, J. Chen, P. Zheng, Y. Zhang, W. Zhang, and Z. Zheng, "Unity is strength: Enhancing precision in reentrancy vulnerability detection of smart contract analysis tools," *IEEE Transactions on Software Engineering*, 2024.

[12] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 664–676.

[13] T. Chen, Z. Li, Y. Zhu, J. Chen, X. Luo, J. C.-S. Lui, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–32, 2020.

[14] X. Chen, P. Liao, Y. Zhang, Y. Huang, and Z. Zheng, "Understanding code reuse in smart contracts," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 470–479.

[15] K. Sun, Z. Xu, C. Liu, K. Li, and Y. Liu, "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*, 2023, pp. 796–807.

[16] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[17] Ethereum, "Ethereum improvement proposals," 2023. [Online]. Available: https://eips.ethereum.org

[18] V. Fabian and B. Vitalik, "Erc-20: Token standard," 2016. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

[19] E. William, S. Dieter, E. Jacob, and S. Nastassia, "Erc-721: Non-fungible token standard," 2018. [Online]. Available: https://eips.ethereum.org/EIPS/eip-721

[20] Ethereum, "Opcodes for the evm," 2023. [Online]. Available: https://ethereum.org/en/developers/docs/evm/opcodes

[21] L. Zamprogno, B. Hall, R. Holmes, and J. M. Atlee, "Dynamic human-in-the-loop assertion generation," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2337–2351, 2022.

[22] Openzepplin, "The standard for secure blockchain applications," 2023. [Online]. Available: https://www.openzeppelin.com/

[23] Go-Ethereum, "debug namespace," 2024. [Online]. Available: https://geth.ethereum.org/docs/interacting-with-geth/rpc/ns-debug

[24] Ethereum, "Ethereum archive node," 2023. [Online]. Available: https://ethereum.org/en/developers/docs/nodes-and-clients/archive-nodes

[25] Openzepplin, "Implementation of the erc-20 permit extension," 2023. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC20Permit.sol

[26] A. Arcuri, "A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage," *IEEE Transactions on Software Engineering*, vol. 38, no. 3, pp. 497–519, 2011.

[27] Foundry, "Foundry," 2024. [Online]. Available: https://github.com/foundry-rs/foundry

[28] T. Chen, Z. Li, X. Luo, X. Wang, T. Wang, Z. He, K. Fang, Y. Zhang, H. Zhu, H. Li *et al.*, "Sigrec: Automatic recovery of function signatures in smart contracts," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 3066–3086, 2021.

[29] Slither, "Slither ir," 2024. [Online]. Available: https://github.com/crytic/slither/wiki/SlithIR

[30] S. Documentation, "Contract abi specification," 2024. [Online]. Available: https://docs.soliditylang.org/en/latest/abi-spec.html#contract-abi-specification

[31] S. Chaliasos, M. A. Charalambous, L. Zhou, R. Galanopoulou, A. Gervais, D. Mitropoulos, and B. Livshits, "Smart contract and defi security tools: Do they meet the needs of practitioners?" in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[32] CryptoRank, "Ethereum cryptocurrency," 2024. [Online]. Available: https://console.cloud.google.com/marketplace/product/ethereum/crypto-ethereum-blockchain

[33] G. BigQuery, "Ethereum tokens," 2024. [Online]. Available: https://cryptorank.io/blockchains/ethereum

[34] Anonymous, "Online supplement material," 2024. [Online]. Available: https://github.com/Jiashuo-Zhang/SolMigrator

[35] S. Yang, J. Chen, and Z. Zheng, "Definition and detection of defects in nft smart contracts," in *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.

[36] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 615–627.

[37] Etherscan, "Top token list," 2024. [Online]. Available: https://etherscan.io/tokens

[38] ——, "Top nft list," 2024. [Online]. Available: https://etherscan.io/nft-top-contracts

[39] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 42–53.

[40] F. Behrang and A. Orso, "Test migration between mobile apps with similar functionality," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 54–65.

[41] Z. Lin, J. Chen, J. Wu, W. Zhang, Y. Wang, and Z. Zheng, "Crpwarner: Warning the risk of contract-related rug pull in defi smart contracts," *IEEE Transactions on Software Engineering*, 2024.

[42] H. Liu, Z. Yang, C. Liu, Y. Jiang, W. Zhao, and J. Sun, "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*, 2018, pp. 900–903.

[43] Z. Tian, Y. Huang, J. Tian, Z. Wang, Y. Chen, and L. Chen, "Ethereum smart contract representation learning for robust bytecode-level similarity detection." in *SEKE*, 2022, pp. 513–518.

[44] Etherscan, "Similar contracts search," 2024. [Online]. Available: https://etherscan.io/find-similar-contracts

[45] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 1503–1520.

[46] Z. He, S. Song, Y. Bai, X. Luo, T. Chen, W. Zhang, P. He, H. Li, X. Lin, and X. Zhang, "Tokenaware: Accurate and efficient bookkeeping recognition for token smart contracts," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 1–35, 2023.

[47] J. Chen, X. Xia, D. Lo, and J. Grundy, "Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–37, 2021.

[48] F. Behrang and A. Orso, "Test migration for efficient large-scale assessment of mobile app coding assignments," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 164–175.

[49] A. Rau, J. Hotzkow, and A. Zeller, "Efficient gui test generation by learning from tests of other apps," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 370–371.

[50] S. Talebipour, Y. Zhao, L. Dojcilović, C. Li, and N. Medvidović, "Ui test migration across mobile platforms," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 756–767.

[51] K. Yang, B. Yang, T. Wang, and Y. Zhou, "Zero-cerd: A self-blindable anonymous authentication system based on blockchain," *Chinese Journal of Electronics*, vol. 32, no. 3, pp. 587–596, 2023.

[52] K. Shang, W. He, S. Zhang, and Z.-h. Zhou, "Review on security defense technology research in edge computing environment," *Chinese Journal of Electronics*, vol. 33, no. 1, pp. 1–18, 2024.

[53] Y. Li and Y. Zhang, "Digital twin for industrial internet," *Fundamental Research*, vol. 4, no. 1, pp. 21–24, 2024.

[54] S. Wu, Z. Li, L. Yan, W. Chen, M. Jiang, C. Wang, X. Luo, and H. Zhou, "Are we there yet? unraveling the state-of-the-art smart contract fuzzers," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[55] N. Ivanov, C. Li, Q. Yan, Z. Sun, Z. Cao, and X. Luo, "Security threat mitigation for smart contracts: A comprehensive survey," *ACM Computing Surveys*, 2023.

[56] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 259–269.

[57] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 227–239.