SMIFIER: A Smart Contract Verifier for Composite Transactions

Yu Dong*, Yue Li*, Dongqi Cui†, Jianbo Gao*, Zhi Guan‡, and Zhong Chen*
*School of Computer Science, Peking University, China
†National Engineering Research Center for Software Engineering, Peking University, China
Email: *{dongyu1101, liyue_cs, gaojianbo, zhongchen}@pku.edu.cn, †cdq@stu.pku.edu.cn, ‡guan@pku.edu.cn,

***Abstract*—Ensuring functional correctness of smart contracts is a pressing security concern to blockchain-based systems. With the development of blockchain application, the trading scenarios and function implementation of smart contracts have become increasing complex, containing several interacted contracts or related functions. However, the existing contracts verifiers for proving functional correctness focus on verifying isolated contract or function but ignore the interactions between them, which makes it difficult to verify correctness of *composite transactions*, i.e., complex transaction scenarios that invoke multiple contracts or trigger a set of transactions. In this paper, we present SMIFIER, a formal verification tool for smart contracts to prove functional properties in composite transactions. SMIFIER defines a set of specifications for composite transactions and can automatically specify properties in these multiple complex transactions. Based on states extraction and mapping, SMIFIER translates annotated Solidity program into Boogie program and verifies relations between functions and properties for interacted contracts. Our experimental evaluation on 12 real-world projects and 65 properties, demonstrates that SMIFIER is practically effective in ensuring functional correctness of properties in composite transactions.**

***Index Terms*—Smart Contract, Formal Verification, Composite Transaction**

I. INTRODUCTION

Ensuring correctness of smart contracts, programs that stored and executed on the blockchain, is an urgent security concern. Smart contracts always store and manage millions of monetary assets, making their security highly sensitive. The pressing need for contract correctness has gained via several recent high-value incidents resulting in massive losses [1]. Unfortunately, most existing security analyzers [2]–[4] focus on universal security vulnerabilities such as contract reentrancy and integer overflow, but ignore functional correctness of contracts, i.e., the functional obligations that the program is intended to implement. In practice, the functionality of smart contracts implies transaction logic and supports a variety of blockchain applications. If developers make mistakes and implement functionality incompletely, the contracts will not behave as intended and will expose vulnerabilities with potentially devastating financial effects. Therefore, it is critical to prove functional correctness of smart contracts.

With the development of blockchain applications, the function implementation of smart contracts has become increasing complex, containing the combination of interacted contracts or transactions. The implementation of a trading scenario always means exploiting a transaction sequence, which involves several interacted contracts externally calling each other, or successive transactions sent to related functions of a contract. As for the contracts in such *composite transactions*, code might appear secure, yet expose vulnerabilities when it interacts with other code. Existing security analyzers always ensure functional correctness of code in isolation such as single contract or function, but cannot verify an entire transaction sequence executing as intended.

Checking for functional correctness of smart contracts in composite transactions should overcome these challenges: 1) composite transactions contain multiple transaction rules and specific contract functionality, thus it is difficult to give a general and explicit definition of composite transactions. Also, to specify properties in these transactions, it is essential to design multiple specifications to describe relations of functions and interactions of contracts. 2) to verify the interaction of contracts, we must check the behaviors of called contracts after invocation. Unfortunately, it is difficult to trace and explore states of called contracts.

In this work, we introduce SMIFIER, an automated formal verifier for proving functional correctness of smart contracts in composite transactions. To achieve this, we give an explicit definition of composite transaction and define multiple kinds of specifications for it. Based on translating Solidity [5] program into an intermediate verification language Boogie [6], SMIFIER can specify function relations and interacted contract properties. In particular, SMIFIER takes advantage of well-engineered pipeline and automated verification conditions generator of Boogie. We describe SMIFIER through examples and evaluate SMIFIER on real-world contracts in composite transactions. We also compare SMIFIER with state-of-the-art safety analysis tools and demonstrate SMIFIER greatly outperforms other tools in terms of specifying properties in composite transactions.

The contribution of our work is as follows:

- We give a definition of composite transaction which can provide an explicit description of multiple complex trading scenarios involving several interacted functions or contracts.
- We impose a framework called SMIFIER, which can automatically verify functional correctness of contracts in composite transactions. For conveniently specifying properties in interacted functions or contracts, we design a set of specifications applied for composite transactions.

DOI reference number: 10.18293/SEKE2022-128

```
1   /// @notice invariant enableTransfer ==> transferFrom
2   contract DaiToken {
3     mapping(address => uint) public balances;
4     address public owner;
5     uint public totalSupply;
6     bool public transferEnabled;
7
8     constructor() public {
9       owner = msg.sender;
10      balances[msg.sender] = totalSupply;
11    }
12    function enableTransfer() public {
13      transferEnabled = true;
14    }
15    function transferFrom(address _from, address _to, uint
          _value) public {
16      require(transferEnabled == true);
17      balances[_from] -= _value;
18      balances[_to] += _value
19    }
20  }
```

```
21  /// @notice invariant sum(DaiToken.balances) == DaiToken.
        totalSupply
22  contract TokenFarm {
23    DaiToken public daiToken;
24
25    constructor(DaiToken _daiToken) public {
26      daiToken = _daiToken;
27    }
28
29    /// @notice precondition DaiToken.transferEnabled ==
          true
30    function deposit(uint _amount) public {
31      daiToken.transferFrom(msg.sender, address(this),
            _amount);
32      _mint(msg.sender, _amount);
33    }
34  }
```

Fig. 1: Simplified Token Farm source code.

- We implement the framework and demonstrate the validation of our tool. We perform the evaluation over 12 real-world projects and 65 properties defined for composite transactions, showing that SMIFIER can effectively verify properties for smart contracts in composite transactions.

II. MOTIVATING EXAMPLE

In this section, we present a motivating example of contracts in composite transaction. Fig. 1 presents the simplified source code from Token Farm, a common DeFi (Decentralised Finance) application [7]. Through Token Farm, users can deposit Dai tokens (a stablecoin cryptocurrency) to the contract and it will mint and transfer Farm Tokens (cryptocurrency provided by the application) to them. The users can later withdraw their Dai tokens by burning their Farm Token on smart contract and the Dai tokens will be transferred back to them. As mentioned, the annotations (beginning with "`/// @notice`") in code will be illustrated in the following section.

In contract DaiToken, the function `enableTransfer` sets the state variable `transferEnabled` to true (line 13 in Fig. 1). Only after the owner makes transfers enabled, the function `transferFrom` can transfer the specified amount of tokens from the address `_from` to the address `_to` (line 15-19). The successive transactions sent to function `enableTransfer` and function `transferFrom` make up a transaction sequence. If the developer forgets writing the `require` clause (line 16), the functionality that tokens are tradeable only after the operation of owner will be incomplete. However, the result of vulnerable `transferFrom` would be identical to the correct one when verifying the single function. Therefore, the proving of this functionality can only be achieved by putting the two transactions together to validate whether the entire transaction sequence is executed in order.

In contract TokenFarm, function `deposit` externally calls function `transferFrom` to transfer several Dai tokens from msg.sender to this address (line 31). Contract DaiToken is a token contract and implements standard interfaces. The developers always assume the implementation is consistent with the standard interfaces and satisfies intended functionality. However, if the token contract makes fake implementation by only invoking standard interfaces without correct code writing, developers will mistakenly call the false function. For example, if `transferFrom` deliberately mistakenly writes the plus and minus operators by writing `balances[_from] += _value` and `balances[_to] -= _value`, "deposit" will become "withdraw" when TokenFarm calls this wrong function, which makes the meaning completely opposite and may lead to unexpected financial loss. Therefor, it is essential to verify functional correctness of the called contract before sending message calls.

III. SMIFIER

In this section, we will describe the definition of composite transaction and the details of SMIFIER in verifying smart contracts. Our verification architecture is summarized in Fig. 2, which consists of three phases. In the first phase, SMIFIER parses annotated smart contracts and generates an abstract syntax tree (AST). In the second phase, SMIFIER traverses the AST and converts Solidity program into Boogie by modular program reasoning. For specifications for composite transactions, we instrument the Boogie program by states recognition, extraction and mapping. Finally, SMIFIER relies on Boogie verifier to prove correctness or reports the violated annotations in Solidity program.

A. Definition of Composite Transactions

Composite transaction means a trading scenario consisting of multiple transactions involving several related functions in a contract or several interacted contracts. We define two kinds of transaction sequences as composite transaction, one *inter-txn* and another *intra-txn*.

Inter-txn. Inter-txn means a transaction sequence containing successive transactions sent to related functions in a contract. Inter-txn is operated by the message sender who sends transactions to contract. We let S represent a transaction sequence,

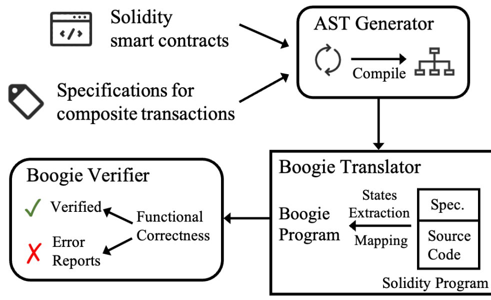

Fig. 2: Schematic workflow of SMIFIER.

$C.f$ represent invoking function f in contract C. The inter-txn can be expressed as $S = \langle C.f_1, C.f_2, C.f_3, \cdots \rangle$ where f_1, f_2, f_3 have relation r. We define r as $f_1 \otimes_v f_2 = f_2 \rightarrow (f_1 \sim v)$, where two functions refer to the same state variable v and the requirement in f_2 depends on the modification in f_1. We divide the relations into two types, **precedence relation** and **exclusion relation**, respectively expressed as $f_1 \otimes_v f_2 = f_2 \overset{=}{\rightarrow} (f_1 \sim v)$ and $f_1 \otimes_v f_2 = f_2 \overset{\neq}{\rightarrow} (f_1 \sim v)$. Precedence relation requires v is equal to the value set by f_2, while exclusion relation requires not equal to. The error handling statements(e.g., assert, require) in f_2 ensures f_2 is executed after f_1 was invoked. For example, function `enableTransfer` (in Fig. 1) modifies `transferEnabled` to true and the execution of `transferFrom` requires the modified value is equal to true. We say the transactions sent to these two functions in sequence form the inter-txn and two functions have precedence relation.

Intra-txn. Intra-txn means a transaction sequence involving a set of contracts which externally call each other. Intrax-txn is operated by the program developer. The intra-txn can be expressed as $S = \langle C_1.f_1, C_2.f_2, C_3.f_3, \cdots \rangle$ where C_1, C_2, C_3 are different contracts and $C_1.f_1$ calls $C_2.f_2$ and $C_2.f_2$ calls $C_3.f_3$. For example, the function `deposit` in contract TokenFarm externally calls the function `transferFrom` in contract DaiToken (in Fig. 1). The transaction sent to contract TokenFarm and then exposing message calls to another contract DaiToken forms the intra-txn.

B. Specifications for Composite Transactions

We design multiple specifications so that SMIFIER could verify function relations for inter-txn and called contract properties for intra-txn, as shown in Table. I. To be mentioned, the specifications are inserted as in-code annotations supported by Solidity and consist of three kinds of statements, *pre- and post-conditions* and *invariants*. Contract-level invariants must hold before and after the execution of every public function. Functions are specified with pre- and post-conditions, which hold before entering functions and specify final states of functions.

For inter-txn, we design two kinds of specifications respectively for two function relations. We let "$\Longrightarrow$" represent precedence relation and "$||$" for exclusion relation. $func1 \Longrightarrow func2$ means $func2$ should be called after $func1$ was invoked because $func2$ requires var is equal to the value assigned in $func1$ while $func1 \,||\, func2$ means $func1$ and $func2$ cannot be called simultaneously. For example, the precedence relation in Fig. 1 is expressed as annotation "`enableTransfer` $\Longrightarrow$ `transferFrom`" (line 1).

TABLE I: Specifications for inter-txn and intra-txn. The first two are for inter-txn while the last one is for intra-txn.

Property	Notation	Description
Precedence Relation	$func1 \Longrightarrow func2$	$func2$ requires the variable var is equal to the value set by $func1$. Insert as contract-level invariant.
Exclusion Relation	$func1 \,||\, func2$	$func2$ requires the variable var is not equal to the value set by $func1$. Insert as contract-level invariant.
Other Contract Property	$Spec\,\{C.var\}$	Specify state variable var of contract C. Insert as contract-level invariant, function-level pre- and post-conditions.

For intra-txn, we create a new format $Spec\,\{C.var\}$. $Spec$ contains all provided forms of property expressions, as well as the specifications designed for inter-txn described above. Different with message calls to contracts supported by Solidity, SMIFIER uses $C.var$ to represent the variable of called contract, where C is the called contract name rather than instance name and var is the variable belonging to C. Intra-txn specifications are inserted in the current contract as all three kinds of specification statements. For example, we insert contract-level invariant in contract TokenFarm (line 21 in Fig. 1) to ensure the sum of individual balance is equal to the total supply of DaiToken. The special function `sum` is provided to express the sum of collections (arrays and mappings). Besides, we can also insert pre-condition before where the call happens to verify the function `enableTransfer` (line 29).

C. Transform and Verification

We implement a transform algorithm for translating Solidity to Boogie program in composite transactions. As shown in Algorithm. 1, given a program P as a list of contracts source codes and S as a list of specifications, the algorithm produces an Boogie program $P^{'}$ that inserted with transformed annotations. The algorithm has three parts. Line 1 transfers the Solidity program into Boogie program without specifications. Lines 3-16 handles the inter-txn specifications in S, while lines 17-27 handles the intra-txn specifications in S.

For each specification s in S, the algorithm first identifies the type of the specification. If s contains the function relation operators "$\Longrightarrow$" or "$||$", we predicate s is for inter-txn and define the two functions before and after the relation operator as f_1 and f_2. Otherwise we identify s is for intra-txn if s contains the name of another contract. For inter-txn, the algorithm extracts requirement in f_2 and assignment in f_1, and then respectively converts into pre- and post-condition. To verify the priority and exclusion of execution, the algorithm generates a new procedure $proc_1$ calling two functions in order and declares another procedure $proc_2$ calling only f_2 if we verify precedence relation. If $proc_1$ is proved and $proc_2$ fails verification, we indicate that f_2 cannot be executed alone and

Algorithm 1: Transform algorithm

Input: Solidity program P as a list of contracts source codes and S as a list of specifications
Output: Transformed Boogie program $P^{'}$

```
1  Transfer Solidity program P to Boogie program P' without S
2  for s ∈ S do
3      if s contains "⟹" or "||" then
4          r ← "⟹" or "||"
5          f1 ← Function before r in s
6          f2 ← Function after r in s
7          pre ← Rewrite require statement in f2
8          Insert pre before f2 into P'
9          post ← Transfer assignment statements in f1
10         Insert post after f1 into P'
11         Declare a new procedure proc1 in P'
12         proc1 ← "call f1; call f2;"
13         if r is "⟹" then
14             Declare a new procedure proc2 in P'
15             proc2 ← "call f2;"
16         end
17         Insert proc1 (and proc2) into P'
18     end
19     else if s contains another contract name then
20         C ← The current contract
21         C' ← The called contract
22         vars ← The variables names of C' in s
23         func ← The called function of C'
24         Create a new specification s' and copy s to s'
25         for var in vars do
26             bg_var ← Locate Boogie variable in func in P'
                   according to type and name of var
27             Replace C.var in s' with bg_var
28         end
29         Insert s' in P' in the same position as s in P
30     end
31     else
32         Transfer and insert s in P'
33     end
34 end
35 Return P'
```

the invocation of f_2 requires f_1, i.e., the precedence relation property is verified. Also, we predicate exclusion relation is proved when the procedure calling two functions fails verification, as two functions cannot execute simultaneously.

For intra-txn, the algorithm locates the called variable names of called contract and replaces them with the translated Boogie representation. We identify variables in Boogie program by first determining whether they are state variables or local variables and then traversing the global or local variable list (of called function) to search them. After research, we replace $C.var$ with variables in Boogie and insert the replaced specification in front of the called contract or function and keep the specification types the same.

Fig. 3 presents the simplified transformed Boogie program of Token Farm contracts (Fig. 1). According to our algorithm, we create two procedures `proc1` and `proc2` in DaiToken respectively calling two functions and only the latter function. The new procedure parameters are made up of arguments of functions called in the procedure. For intra-txn, we move specifications (previously inserted in contract TokenFarm) to the front of called contract DaiToken and called function `transferFrom`. We also replace variables in properties (e.g., `DaiToken.balances`) with Boogie expressions.

```
1  // invariant sum(balances) == totalSupply
2  contract DaiToken {
3    var balances: [address]int;
4    var transferEnabled: bool;
5    procedure enableTransfer(...) { ... }
6    // precondition transferEnabled == true
7    procedure transferFrom(_from: address, _to: address,
         _value: uint) { ... }
8    procedure proc1(_from: address, _to: address, _value:
         uint) {
9      call enableTransfer();
10     call transferFrom(_from, _to, _value);
11   }
12   procedure proc2(_from: address, _to: address, _value:
         uint) {
13     call transferFrom(_from, _to, _value);
14   }
15 }
16 contract TokenFarm {
17   DaiToken public daiToken;
18   procedure deposit(_amount: uint) {
19     // call transferFrom() in DaiToken
20   }
21 }
```

Fig. 3: Simplified transformed Boogie program of Token Farm.

After being transferred to Boogie program, SMIFIER leverages Boogie verifier to transform the program into verification conditions and discharge them using SMT solvers. SMIFIER verifies each procedure of each contract and outputs the verification result of each procedure. If Boogie proves the correctness of program, the result will display "OK". Otherwise if there are vulnerabilities in program, SMIFIER will report "Error" and map the violated annotations back to the Solidity code (e.g., line numbers, function names).

IV. Evaluation

We now present out evaluation of SMIFIER on real-world Ethereum projects. We focus on the following key questions: 1) What types of properties are common for composite transactions in real world? 2) How effective is SMIFIER in verifying smart contracts in composite transactions? 3) How does SMIFIER compare with other smart contract safety analysis tools? All experimental results reported in this section are conducted on a server running Ubuntu 20.04 LTS with 32 AMD EPYC CPUs at 2.8GHz and 64GB of physical memory.

A. Benchmark and Properties

Benchmark. We have collected in total 12 Ethereum projects in ERC-20, ERC-721 standards and DeFi applications. We focus on these projects because 1) they are most widely used contracts, 2) they contain several related functions in a contract or several interacted contracts which satisfy definition of composite transactions, 3) we want to focus our analysis on contracts that manipulate critical digital assets. Our benchmark includes the top five ERC-20 contracts and the top five ERC-721 contracts ranked by Etherscan [8] which define the interface and specification for implementing fungible and non-fungible tokens respectively. We also include two decentralized exchanges (DEX), Uniswap and Sushiswap, which

TABLE II: Properties for composite transactions and concrete examples taken from the benchmarks.

Type	Description	Example
User-based access control	Only particular users have privileges to perform critical actions.	`function setOwner() {owner = msg.sender;}` `function withdraw() {require(msg.sender == owner);}`
Token-based circulation control	The circulation of token is allowed at certain states.	`function enableTransfer() {transferEnabled = true;}` `function transferFrom() {require(transferEnabled == true);}`
Contract life circle	The transaction of contract is allowed at certain states.	`function pause() {paused = true;}` `function unpause() {require(paused == true);}`
Balances consistence	The sum of individual balance keeps invariant.	`invariant sum(ERC20.balance) == ERC20.totalSupply`
State-based properties	Defines states which invariants must hold or variables must satisfy.	`invariant ERC721._approve ==> ERC721.transferFrom`

use decentralized network protocols to facilitate automated transactions between cryptocurrency tokens.

Properties. To focus on verifying composite transactions, we have summarized common properties and classified them into five distinct categories, as shown in Table. II.

1) *User-based access control* defines only particular users have privileges to perform critical actions. This property is expressed as function relation, where a user has permissions to invoke another function only after being granted access in one function. The example in Table. II, taken from CK contract, stipulates that only the owner set in function `setOwner` can invoke `withdraw`.
2) *Token-based circulation control* means that the circulation (minting, release and transaction) of token is allowed at certain states. Only after the tokens are set accessible in one function, the manipulation of tokens in another function can be invoked. The example in Table. II, taken from LEO contract, shows that function `transferFrom` can be invoked only after `enableTransfer` enables transfer.
3) *Contract life circle* defines at which states the transactions of contracts are allowed. The contracts usually have emergency stop mechanism or can be deprecated, controlled by state variables. The example in Table. II, taken from USDT contract, states that function `pause` triggers stopped state of the contract and the function `unpause` returns the contract to normal state.
4) *Balances consistency* indicates that the sum of individual balance keeps invariant after transfers occur. This property is applied to verifying functional correctness of the called contract. The example in Table. II ensures the sum of account balances is equal to the total supply in Uniswap contract.
5) *State-based properties* defines states which invariants must hold and variables must satisfy. This property can be inserted in single contract or function. It can also be expressed in the current contract as specification for the called contract. The example in Table. II shows the verification of precedence relation in MCHH contract.

B. Verifying Contracts using SMIFIER

We now report on the effectiveness of SMIFIER in verifying our benchmarks. For each project, we manually insert specifications using the representation defined in Section.III, which contain all kinds of properties described above. We represent out results in Table. III.

The key result is that SMIFIER can successfully verify 60 of the 65 properties (92.3%) in the benchmarks. The reason why SMIFIER cannot verify the remaining five properties is that the specification we designed is difficult to express some properties in ERC-721 contracts. There are several functions in ERC-721 depending on parameters but not state variables. If we specify parameters in properties, we are unable to extract them from functions as state variables when we implement transform algorithm. We will solve this problem in future work. The average verification time of SMIFIER is 3.72 seconds and the time is related to the lines of code. In general, the longer the program, the more contracts and functions the program contains, the more time SMIFIER takes to verify it. Because SMIFIER transforms and verifies each function of each contract in sequence.

Finally, we compare SMIFIER to state-of-the-art smart contract analysis tools: KEVM [9], SOLC-VERIFY [10], VERISOL [11] and VERX [12], as shown in Table. IV. Our benchmark consists the first four types of properties defined in the Table. II which are all for composite transactions, and the last property for single contract and function. Results indicate that other tools can only analyze properties in single function or contract or some properties about function relations while SMIFIER can verify other contract properties (balances consistence), which is very important for verifying complete functional correctness.

V. RELATED WORK

In recent years, there has been great interest in formally verifying the correctness of smart contracts. For instance, KEVM [9] translates EVM bytecode to KEVM and leverages the K framework [13] for checking contracts against given specifications while Ahrendt [14] translates Solidity into Java and uses KeY [15], a deductive Java verification tool. SOLC-VERIFY [10] and VERISOL [11] are two verifiers require users to manually provide annotations and check contracts based on translation to Boogie. Except formal verification, there are other systems using symbolic execution approach for verifying properties. VERX [12] can automatically prove temporal safety properties of smart contracts since it extracts predicates

TABLE III: Experimental results of SMIFIER verification on benchmarks. **LOC**: the number of lines of code, **Contracts**: the number of contracts in the program, **Functions**: the number of functions in the program, **Properties**: the number of properties we verified, **Verified**: the number of properties that were successfully proved, **Avg.time**: the average analysis time in seconds.

Contract	LOC	Contracts	Functions	Properties	Verified	Avg.Time (s)
USDT	447	7	25	4	4	2.46
CRO	641	8	30	5	5	2.87
WBTC	663	11	32	5	5	2.89
LEO	734	7	36	6	6	3.03
Fantom	624	6	46	7	7	2.92
LAND	1118	6	62	7	6	4.24
CK	1977	14	82	9	7	5.33
AXIE	891	7	53	4	4	3.24
MCHH	1192	10	56	6	5	4.21
Sherbet	1407	6	53	3	2	4.57
Uniswap	1256	7	52	5	5	4.46
Sushiswap	1353	8	47	4	4	4.42
Overall	12303	97	574	65	60	3.72

TABLE IV: Comparison of SMIFIER with other analysis tools. "✓" represents the tool can verify the property while "✗" represents the tool cannot verify the property.

Benchmark	KEVM	SOLC-VERIFY	VERISOL	VERX	SMIFIER
Single Contract Property	✓	✓	✓	✓	✓
Single Function Property	✓	✓	✓	✓	✓
User-based Access Control	✗	✗	✓	✓	✓
Token-based Circulation Control	✗	✗	✗	✓	✓
Contract Life Circle	✗	✗	✗	✓	✓
Balances Consistence	✗	✗	✗	✗	✓

automatically from the contract's source code. SMARTPULSE [16] models the contract's execution environment and uses CFGAR-based approach to check liveness properties.

VI. CONCLUSIONS

We presented SMIFIER, the first formal verification tool of Solidity smart contracts for composite transactions. This paper gives a definition of composite transaction which involves several related functions or interacted contracts and defines the two types it contains. SMIFIER presents a set of specifications and transform algorithm which specify properties and transfer Solidity program to Boogie program. After transform, SIMIFIER relies on Boogie verifier to discharge verification conditions and prove function correctness. We demonstrated that SMIFIER is effective in specifying properties in composite transactions and proving functional correctness over 12 real-world Ethereum projects and 65 properties.

VII. ACKNOWLEDGEMENT

Zhi Guan is the corresponding author. Zhi Guan is supported by National Key R&D Program of China (NO.2020YFB1005800) and Beijing Natural Science Foundation(M21040).

REFERENCES

[1] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International conference on principles of security and trust*. Springer, 2017, pp. 164–186.

[2] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[3] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," *HITB SECCONF Amsterdam*, vol. 9, p. 54, 2018.

[4] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[5] "Solidity documentation," 2022, https://docs.soliditylang.org/en/v0.8.12/.

[6] K. R. M. Leino, "This is boogie 2," *manuscript KRML*, vol. 178, no. 131, p. 9, 2008.

[7] "Decentralized finance," 2017, https://ethereum.org/en/defi/.

[8] "Ethereum (eth) blockchain explorer," 2015, https://etherscan.io/.

[9] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 204–217.

[10] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2019, pp. 161–179.

[11] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, I. Naseer, and K. Ferles, "Formal verification of workflow policies for smart contracts in azure blockchain," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2019, pp. 87–106.

[12] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 1661–1677.

[13] G. Roșu and T. F. Șerbănută, "An overview of the k semantic framework," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[14] W. Ahrendt, R. Bubel, J. Ellul, G. J. Pace, R. Pardo, V. Rebiscoul, and G. Schneider, "Verification of smart contract business logic," in *International Conference on Fundamentals of Software Engineering*. Springer, 2019, pp. 228–243.

[15] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, "Deductive software verification-the key book," *Lecture notes in computer science*, vol. 10001, 2016.

[16] J. Stephens, K. Ferles, B. Mariano, S. Lahiri, and I. Dillig, "Smartpulse: automated checking of temporal properties in smart contracts," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 555–571.