




Isabelle/Solidity: A Tool for the Verification of Solidity Smart Contracts (tool paper)

Asad Ahmed   

University of Exeter, Exeter EX4 4PY, UK

Diego Marmosoler  

University of Exeter, Exeter EX4 4PY, UK

Abstract

Smart contracts are an important innovation in Blockchain which allow to automate financial transactions. Every day, hundreds of thousands of new contracts are deployed managing millions of dollars' worth of transactions. Thus, bugs in smart contracts may lead to high financial losses and it is important to get them right before deploying them to the Blockchain. To address this problem we developed Isabelle/Solidity, a tool for the verification of smart contracts in Isabelle. The tool is implemented as a definitional extension for the Isabelle proof assistant and thus complements existing tools in this area which are mostly based on axiomatic approaches. In this paper we describe Isabelle/Solidity and demonstrate it by verifying a casino contract from the VerifyThis long term verification challenge.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases Program Verification, Smart Contracts, Isabelle, Solidity

Digital Object Identifier 10.4230/OASICS.CVIT.2016.23

Funding This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/X027619/1]

1 Introduction

One important innovation which comes with Blockchain are so-called *smart contracts*. These are digital contracts which are automatically executed once certain conditions are met and which are used to automate transactions on the Blockchain. For instance, a payment for an item might be released instantly once the buyer and seller have met all specified parameters for a deal. Every day, hundreds of thousands of new contracts are deployed [9] managing millions of dollars' worth of transactions [15].

Technically, a smart contract is *code which is deployed to a Blockchain* and which can be executed by sending special transactions to it. Thus, as for every computer program, smart contracts may contain bugs which can be exploited. However, since smart contracts are often used to automate financial transactions, such exploits may result in huge economic losses. In general, it is estimated that since 2019, more than \$5B was stolen due to vulnerabilities in smart contracts [5].

The high impact of vulnerabilities in smart contracts together with the fact that once deployed to the Blockchain, they cannot be updated or removed easily, makes it important to “*get them right*” before they are deployed. To address this problem, we developed Isabelle/Solidity, a tool for the deductive verification of Solidity smart contracts implemented as a definitional extension for the Isabelle [?] proof assistant. The main aim of this paper is to describe the tool and demonstrate it in terms of an example. To this end its contribution is two-fold:

- We provide an overview of the technical architecture of Isabelle/Solidity.
- We use it to verify the casino contract from the VerifyThis verification challenge.



© Jane Open Access and Joan R. Public;
licensed under Creative Commons License CC-BY 4.0

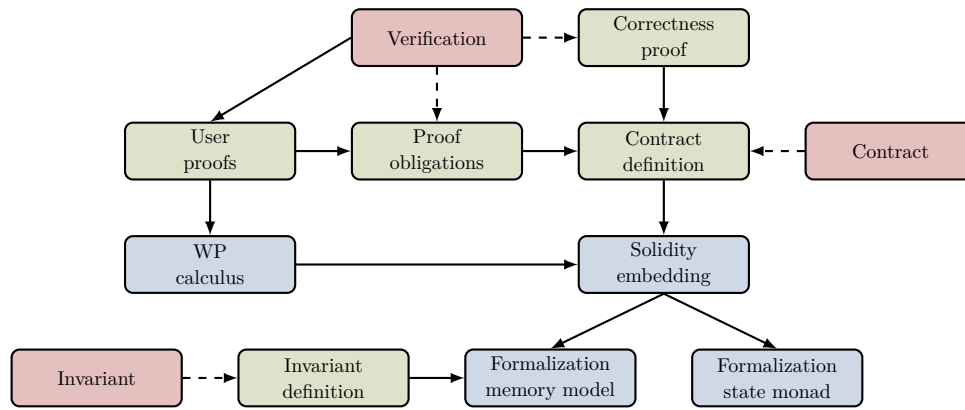
42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:8

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Isabelle/Solidity: theories (blue), commands (red), and generated artefacts (green).

2 Isabelle/Solidity

Isabelle/Solidity is available as a definitional package for the Isabelle proof assistant. It is available online and can be downloaded from www.marmsoler.com/assets/isabelle-solidity.zip

Figure 1 shows the architecture of Isabelle/Solidity. Isabelle/Solidity is based on four Isabelle theories which are represented by the blue rectangles:

Formalization state monad We model Solidity programs as functions which manipulate states. Such functions are usually described in terms of state monads [14] and this theory contains our formalization of it based on [7].

Formalization memory model Solidity has a quite particular memory model with different types of stores which support different types of data structures. This theory contains our formalization of the Solidity memory model. It also defines the notion of a Solidity state as a collection of different stores.

Solidity embedding A Solidity statement is defined as a particular state monad manipulating a Solidity state. This theory contains definitions for all of our Solidity statements.

WP calculus To support a user in the verification of Solidity programmes, Isabelle/Solidity comes with a verification condition generator (VCG). The VCG is based on a weakest precondition calculus [8] for our Solidity statements and which is formalized in this theory.

In addition to the theories described above, Isabelle/Solidity extends the Isabelle proof assistant by means of three new definitional commands represented by the red rectangles in Figure 1. Each of the commands generates lower-level definitions and theorems for Isabelle/HOL which are represented as green rectangles.

Contract This command allows a user to specify a new contract. It requires a them to specify a list of member variables, a constructor, and a list of methods. For each method the user can specify a list of parameters as well as a method body. The body is provided as a state monad using the monads defined in the corresponding Isabelle theory. The command then generates definitions for the *contract's methods*. To this end each method is mapped to a corresponding partial function definition [10].

Invariant This command supports a user in the specification of an invariant for the contract. An invariant is specified as a HOL formula over the store and balance of the contract. The command then uses the typing information of member variables to generate a corresponding *invariant definition*.

Verification This command triggers the verification of a contract. It requires a user to provide an invariant as well as postconditions for the methods. Then it presents the user with a list of *proof obligations* for each of the contract's methods. The users then need to discharge these proof obligations by providing a corresponding *proof*. To this end they can use the WP calculus provided by our framework. After discharging these obligations, Isabelle/Solidity proves an overall *correctness theorem* which guarantees that the invariant as well as postconditions are not violated.

3 Case Study

To demonstrate Isabelle/Solidity in action, we use it to verify the casino contract from the VerifyThis long-term verification challenge [1]. The version of the contract used here is provided in Appendix A. The casino contract implements a betting game based on guessing the outcome of coin-tossing. At any time, the game is characterized by three states, i.e., `IDLE`, `GAME_AVAILABLE` and `BET_PLACED`. Initially, the game is in the `IDLE` state. The game has been implemented in Solidity using the following functions.

- The operator can create a new game by calling the `creatGame` function and provide the hash value of a secret number. The function stores the secret number and changes the state to `GAME_AVAILABLE`.
- Once in state `GAME_AVAILABLE`, a player can place a bet by invoking the `placeBet` function and provide a guess (`HEADS` or `TAILS`). This function stores the player's address, its guess, and the amount of the bet, and changes the state to `BET_PLACED`.
- Now, the operator can decide the bet anytime by calling the `decideBet` function and providing the secret number. The secret number is then used to decide the outcome of the coin tossing (`HEADS` or `TAILS`). If the player's guess is correct, the double of the amount of the bet is transferred to their address. Otherwise, the amount equal to the bet is added to the pot. The function also changes the state of the game to `IDLE`.
- The operator may add money to the bet, at anytime, using `addToPot` but can only remove money if the game is in not in state `BET_PLACED` by calling `removeFromPot`.

4 Specification

To verify a smart contract in Isabelle/Solidity we first need to specify it. To this end Isabelle/Solidity provides the `contract` command which supports a user in this task. The command requires a user to specify a list of member variables and corresponding types, followed by a specification of the constructor and the contract's methods.

Storage Variables

Listing 1 shows the specification of storage variables for the casino contract in Isabelle/Solidity.

Listing 1: Isabelle/Solidity data types for Casino

```
1 contract Casino
2   for state: StateT
3   and operator: AddressT
4   and player: AddressT
5   and pot: IntT
6   and hashedNumber: BytesT
7   and bet: IntT
8   and guess: CoinT
```

112 In general, Isabelle/Solidity supports most of the basic Solidity data types, such as bit-
 113 sized integers, bytes, and addresses. Enums can be encoded as integers with corresponding
 114 abbreviations.

115 Methods

116 We do not show the specification for all of the methods of the casino contract here but we
 117 only discuss one. The others can be similarly translated from the original Solidity contract.

118 The specification of a new method can be done using the keyword `emethod`. To allow
 119 a method to receive funds we can set the payable flag by using the corresponding `payable`
 120 keyword. What follows is a specification of stack (`param`), memory (`memory`), and calldata
 121 (`calldata`) parameters and corresponding types. Finally, one can provide the body of the
 122 function using the `where` keyword followed by a corresponding monad specification (using
 123 `"do {...}"`, notation).

124 Listing 2 shows the specification of the function `decideBet`. It is declared to be payable
 125 and accepts one parameter `secretNumber`, of integer type. Lines 5-7, implement preconditions
 126 for deciding a bet, i.e., only the operator can call the function, the game should be in state
 127 `BET_PLACED` and `secretNumber` should be equal to the `hashedNumber`. For this purpose,
 128 Isabelle/Solidity employs `assert` which models the Solidity `require` command.

129 Isabelle/Solidity also supports other Solidity statements, such as control structures and
 130 assignment operators. For example, in Line 9, `IF...THEN...ELSE` reveals `HEADS` or `TAILS`
 131 by taking the modulus (`(%)`) of `secretNumber`. For assignment operators, Isabelle/Solidity
 132 distinguishes between stack (`::=`) and storage (`::=s`) assignments along with corresponding
 133 lookup operators (`~` and `~s`).

Listing 2: Isabelle/Solidity method for Casino

```

1  emethod decideBet payable
2  param secretNumber: IntT
3  where
4  do {
5    byOperator;
6    inState (Sint BET_PLACED);
7    <assert> (hashedNumber ~s [] <=> (<keccak256> (secretNumber ~ [])));
8    decl TSint secret;
9    secret [] ::= IF ((secretNumber ~ []) <%(sint) 2> <=> (<sint> 0)
10      THEN <sint> HEADS ELSE <sint> TAILS;
11    IF (secret ~ []) <=> (guess ~s []) THEN
12      do {
13        pot [] ::=s ((pot ~s []) <-> (bet ~s []));
14        bet [] ::=s <sint> 0;
15        <transfer> (player ~s []) ((bet ~s []) <*> (<sint> 2))
16      }
17    ELSE
18      do {
19        pot [] ::=s pot ~s [] <+> bet ~s [] ;
20        bet [] ::=s <sint> 0
21      };
22    state [] ::=s <sint> IDLE
23  },

```

5 Verification

Isabelle/Solidity facilitates the specification of invariants using the `invariant` command. This command requires a user to provide the name of the invariant, followed by its specification in terms of a predicate formulated over the contract's store and balance. It then generates a definition for the invariant which, in addition to the predicate specified by the user, also requires the value of member variables to adhere to their types. The command also proves corresponding introduction and elimination rules which can be invoked during verification.

Example 5.1 (Invariant). Assume that we want to ensure our contract has always enough funds to cover the payout of players. More formally, we want to ensure that, whenever the game is in the `BET_PLACED` state, the contract's internal balance satisfies:

$$\text{pot_balance}(s, b) = b \geq s(\text{"pot"}) + s(\text{"bet"}) \wedge s(\text{"bet"}) \leq s(\text{"pot"}) \quad (1)$$

and if it is not in `BET_PLACED`, then

$$\text{pot_balance}(s, b) = b \geq \text{pot} \quad (2)$$

The corresponding specification in Isabelle/Solidity is given in Listing 3.

Listing 3: Invariant in Isabelle/Solidity

```
1 invariant pot_balance sb where
2   (fst sb state = Value (Sint BET_PLACED)
3    → snd sb ≥ unat (sint (vt (fst sb pot)))
4      + unat (sint (vt (fst sb bet)))
5      ∧ sint (vt (fst sb bet)) ≤ sint (vt (fst sb pot))) ∧
6   (fst sb state ≠ Value (Sint BET_PLACED)
7    → snd sb ≥ unat (sint (vt (fst sb pot))))
8   for "casino"
```

To formally verify the invariant, Isabelle/Solidity provides the `verification` command. It requires the user to provide a name followed by an invariant specified using the `invariant` keyword. Moreover, a user can provide postconditions for the constructor and each of the contract's methods. The corresponding specification for our casino contract is shown in Listing 4. The postconditions require a method to update the corresponding state of the game properly.

Listing 4: Verification in Isabelle/Solidity

```
1 verification pot_balance:
2   pot_balance
3   "K (K (K True))"
4   "createGame" "createGame_post" and
5   "placeBet" "placeBet_post" and
6   "decideBet" "decideBet_post" and
7   "addToPot" "K (K (K True))" and
8   "removeFromPot" "K (K (K (K True)))"
9   for "casino"
```

The verification command tries to prove a general correctness theorem for our contract. To this end, it provides the user with a set of proof obligations which are required to be discharged to verify the contract. For our example, the verification command provides us with six different proof obligations (one for each of the contract's functions). The one for `decideBet` is shown in Listing 5.

Listing 5: Verifying casino contract in Isabelle/Solidity

```

1  show  $\bigwedge$  call secretNumber. ( $\bigwedge$  x h r. effect (call x) h r  $\Rightarrow$  vcond x) h r)
2   $\Rightarrow$  effect (decideBet call secretNumber) s r
3   $\Rightarrow$  inv_state pot_balance s
4   $\Rightarrow$  post s r pot_balance (K True) (decideBet_post secretNumber)
5  unfolding decideBet_def
6  apply (erule post_exc_true, erule_tac post_wp)
7  unfolding inv_state_def
8  apply (vcg | solve<auto simp add: wpsimps>)+
9  ...

```

The goal basically requires to show that the invariant is preserved by the function. To discharge it, a user can use our verification condition generator and general Isabelle reasoning infrastructure.

6 Related Work

Early approaches to the verification of smart contracts are mostly based on automatic verification techniques. A popular example of research in this area is solc-verify [?] which is based on the boogie verifier [?]. While tools in this category are fully automated, they are often limited in expressiveness compared to interactive verification approaches.

One line of research in this area focusses on the development of tools to verify smart contracts independent of the programming language. One example in this area is the work of Cassez et al. [?, ?] about deductive verification of smart contracts with Dafny. While these tools can also be used to verify Solidity smart contracts, they may reach their limits when it comes to contracts involving more specialized language features.

Other tools focus on the verification of contracts written in a particular language, such as Solidity. One example here is SolidiKeY [2] which allows to formally specify and verify Solidity smart contracts using the KeY prover. SolidiKeY is based on an axiomatic semantics of Solidity which is the main difference to Isabelle/Solidity, which is based on a denotational semantics [12] and implemented in a definitional approach.

7 Conclusion

In this paper, we present Isabelle/Solidity, a tool for the formal specification and verification of Solidity smart contracts. The tool is implemented as a definitional extension of the Isabelle proof assistant and provides features for the specification of contracts and invariants, and the verification of invariants and postconditions. To support the user in the verification it also provides a verification condition generator based on a weakest precondition calculus.

Isabelle/Solidity currently supports many features of Solidity, including domain specific expressions and advanced data types such as maps and arrays and their representation in different types of stores (storage, memory, calldata). There are, however, more advanced features of the language, such as inheritance, which is currently not supported and a task for future work.

Another limitation is the lack of a verified compiler for Isabelle/Solidity. Thus, it cannot be guaranteed that the verified properties hold on the level of EVM bytecode. Thus, another direction for future work is the development of a verified compiler for Isabelle/Solidity.

References

- 1 Wolfgang Ahrendt. Ethereum transactions per day, 12 2024. URL: <https://web.archive.org/web/20241209135636/https://verifythis.github.io/02casino/>.
- 2 Wolfgang Ahrendt and Richard Bubel. Functional verification of smart contracts via strong data integrity. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part III* 9, pages 9–24. Springer, 2020.
- 3 Pedro Antonino and AW Roscoe. Formalising and verifying smart contracts with solidifier: a bounded model checker for solidity. *arXiv preprint arXiv:2002.02710*, 2020.
- 4 Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pages 91–96, 2016.
- 5 CipherTrace. Cryptocurrency crime and anti-money laundering report. Technical report, 2021.
- 6 Edmund M Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings* 17, pages 54–56. Springer, 1997.
- 7 David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 167–182. Springer, 2008.
- 8 Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, aug 1975.
- 9 Etherscan. Ethereum daily deployed contracts chart, 02 2025. URL: <https://etherscan.io/chart/deployed-contracts>.
- 10 Alexander Krauss. Recursive definitions of monadic functions. *Electronic Proceedings in Theoretical Computer Science*, 43:1–13, 2010. URL: <http://dx.doi.org/10.4204/EPTCS.43.1>, doi:10.4204/eptcs.43.1.
- 11 Diego Marmosler, Asad Ahmed, and Achim D Brucker. Secure smart contracts with isabelle/solidity. In *International Conference on Software Engineering and Formal Methods*, pages 162–181. Springer, 2024.
- 12 Diego Marmosler and Achim D Brucker. Isabelle/solidity: a deep embedding of solidity in isabelle/hol. *Formal Aspects of Computing*, 2022.
- 13 Maria Ribeiro, Pedro Adão, and Paulo Mateus. Formal verification of ethereum smart contracts using isabelle/hol. In *Logic, Language, and Security: Essays Dedicated to Andre Scedrov on the Occasion of His 65th Birthday*, pages 71–97. Springer, 2020.
- 14 Philip Wadler. Monads for functional programming. In Manfred Broy, editor, *Program Design Calculi*, pages 233–264. Springer, 1993.
- 15 Ycharts. Ethereum transactions per day, 02 2025. URL: https://ycharts.com/indicators/ethereum_transactions_per_day.
- 16 Jakub Zakrzewski. Towards verification of ethereum smart contracts: a formalization of core of solidity. In *Verified Software. Theories, Tools, and Experiments: 10th International Conference, VSTTE 2018, Oxford, UK, July 18–19, 2018, Revised Selected Papers* 10, pages 229–247. Springer, 2018.

A Casino: Solidity Smart Contract

Listing 6: Solidity source code for the Casino

```

1  contract Casino {
2      enum Coin { HEADS, TAILS } ;
3      enum State { IDLE, GAME_AVAILABLE, BET_PLACED }
4      State private state;
5      address public operator, player;
6      uint public pot;
7      bytes32 public hashedNumber;
8      uint public bet;
9      Coin guess;
10
11     function createGame(bytes32 hashNum)
12         public byOperator, inState(IDLE) {
13         hashedNumber = hashNum;
14         state = GAME_AVAILABLE;
15     }
16
17     function placeBet(Coin _guess) public payable inState(GAME_AVAILABLE) {
18         require (msg.sender != operator);
19         require (msg.value <= pot);
20         state = BET_PLACED;
21         player = msg.sender;
22         bet = msg.value;
23         guess = _guess;
24     }
25
26     function decideBet(uint secretNumber)
27     public byOperator, inState(BET_PLACED) {
28         require (hashedNumber == keccak256(secretNumber));
29         Coin secret = (secretNumber % 2 == 0)? HEADS : TAILS;
30         if (secret == guess) {
31             pot = pot - bet;
32             bet = 0;
33             player.transfer(bet*2);
34         } else {
35             pot = pot + bet;
36             bet = 0;
37         }
38         state = IDLE;
39     }
40
41     function addToPot() public payable byOperator {
42         pot = pot + msg.value;
43     }
44
45     function removeFromPot(uint amount) public byOperator, noActiveBet {
46         operator.transfer(amount);
47         pot = pot - amount;
48     }
49 }

```