Proceedings of the First Workshop on Formal Methods for Blockchains, FMBC, 2019

Néstor Cataño, E-mail: `nestor.catano@gmail.com`
Diego Marmsoler, E-mail: `diego.marmsoler@tum.de`
Bruno Bernardo, E-mail: `bruno@nomadic-labs.com`

Foreword

We are glad to present the articles of the 1st Workshop on Formal Methods for Blockchains (FMBC), which is part of the Formal Methods congress that is held this year in the city of Porto, Portugal. We expect to hold this workshop in the coming years.

This pre-proceedings consists of 10 articles with a conditional acceptance for publication in the post-proceedings, 2 lighting talk articles. The versions hereby submitted are preliminary versions of the final ones that will be published in the post-proceedings.

Acknowledgements

We would like to express our most sincere gratitude to the FM chairs, all the PC members, the external reviewers, the Invited Speaker, and all the authors who submitted their work and made this workshop possible.

We particularly would like to express our gratitude to our sponsor Nomadic Labs for its generous support for the realization of FMBC 2019.

Contents

1 Deductive Proof of Industrial Smart Contracts Using Why3 3

2 Statistical Model Checking of RANDAO's Resilience to Pre-computed Reveal Strategies 15

3 Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts 27

4 Call me Back, I have a Type Invariant 39

5 A Distributed Blockchain Model of Selfish Mining 48

6 Towards a Verified Model of the Algorand Consensus Protocol in Coq 60

7 Smart Contracts: Application Scenarios for Deductive Program Verification 66

8 Formal specification of a security framework for smart contracts 72

9 Verifying Smart Contracts with Cubicle 84

10 Smart Contract Interactions in Coq 96

11 Smart Contract Interactions in Coq 110

12 Towards a Smart Contract Verification Framework in Coq 116

Deductive Proof of Industrial Smart Contracts Using Why3

Zeinab Nehaï[1,2] and François Bobot[2]

[1] Université Paris Diderot, Paris, France
[2] CEA LIST, Palaiseau, France
zeinab.nehai@univ-paris-diderot.fr
{zeinab.nehai, francois.bobot}@cea.fr

Abstract. In this paper, we use a formal language that performs deductive verification on industrial smart contracts, which are self-executing digital programs. Because smart contracts manipulate cryptocurrency and transaction information, if a bug occurs in such programs, serious consequences can happen, such as a loss of money. The aim of this paper is to show that a language dedicated to deductive verification, called *Why3*, can be a suitable language to write correct and proven contracts. We first encode existing contracts into the *Why3* program; next, we formulate specifications to be proved as the absence of RunTime Error and functional properties, then we verify the behaviour of the program using the *Why3* system. Finally, we compile the *Why3* contracts to the Ethereum Virtual Machine (EVM). Moreover, our approach estimates the cost of gas, which is a unit that measures the amount of computational effort during a transaction.

Keywords: deductive verification, why3, smart contracts, solidity.

1 Introduction

Smart Contracts [21] are sequential and executable programs that run on Blockchains [17]. They permit trusted transactions and agreements to be carried out among parties without the need for a central authority while keeping transactions traceable, transparent, and irreversible. These contracts are increasingly confronted with various attacks exploiting their execution vulnerabilities. Attacks lead to significant malicious scenarios, such as the infamous *The DAO* attack [7], resulting in a loss of ∼\$60M. In this paper, we use formal methods on smart contracts from an existing Blockchain application. Our motivation is to ensure safe and correct contracts, avoiding the presence of computer bugs, by using a deductive verification language able to write, verify and compile such programs. The chosen language is an automated tool called *Why3* [13], which is a complete tool to perform deductive program verification, based on Hoare logic. A first approach using *Why3* on *Solidity* contracts (the Ethereum smart contracts language) has already been undertaken [2]. The author uses *Why3* to formally verify *Solidity* contracts based on code annotation. Unfortunately, that work remained at the prototype level. We describe our research approach through a use case that has already been the subject of previous work, namely the Blockchain Energy Market Place (BEMP) application [19]. In summary, the contributions of this paper are as follows:

1. Showing the adaptability of *Why3* as a formal language for writing, checking and compiling smart contracts.
2. Comparing existing smart contracts, written in *Solidity* [11], and the same existing contracts written in *Why3*.
3. Detailing a formal and verified *Trading* contract, an example of a more complicated contract than the majority of existing *Solidity* contracts.
4. Providing a way to prove the quantity of *gas* (fraction of an Ethereum token needed for each transaction) used by a smart contract.

The paper is organized as follows. Section 2 describes the approach from a theoretical and formal point of view by explaining the choices made in the study, and section 3 is the proof-of-concept of compiling *Why3* contracts. A state-of-the-art review of existing work concerning the formal verification of smart contracts is described in section 4. Finally, section 5 summarizes conclusions.

2 A New Approach to Verifying Smart Contracts

2.1 Background of the study

Deductive approach & Why3 tool. A previous work aimed to verify smart contracts using an abstraction method, model-checking [19]. Despite interesting results from this modelling method, the approach to property verification was not satisfactory. Indeed, it is well-known that model-checking confronts us either with limitation on combinatorial explosion, or limitation with invariant generation. Thus, proving properties involving a large number of states was impossible to achieve because of these limitations. This conclusion led us to consider applying another formal methods technique, deductive verification, which has the advantage of being less dependent on the size of the state space. In this approach, the user is asked to write the invariants. We chose the automated *Why3* tool [13] as our platform for deductive verification. It provides a rich language for specification and programming, called *WhyML*, and relies on well-known external theorem provers such as Alt-ergo [10], Z3 [16], and CVC4 [8]. *Why3* comes with a standard library[3] of logical theories and programming data structures. The logic of *Why3* is a first-order logic with polymorphic types and several extensions: recursive definitions, algebraic data types and inductive predicates.

Case study: Blockchain Energy Market Place. We have applied our approach to a case study provided by industry [19]. It is an Ethereum Blockchain application (BEMP) based on *Solidity* smart contracts language. Briefly, this Blockchain application makes it possible to manage energy exchanges in a peer-to-peer

[3] http://why3.lri.fr/

way among the inhabitants of a district as shown in Figure 1. The figure illustrates (1) & (1') energy production (Alice) and energy consumption (Bob). (2) & (2') Smart meters provide production/consumption data to Ethereum blockchain. (3) Bob pays Alice in *ether* (Ethereum's cryptocurrency) for his energy consumption. For more details about the application, please refer to [19].

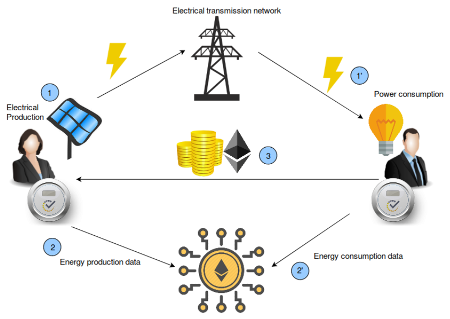


Fig. 1. BEMP Process

In our initial work, we applied our method on a simplified version of the application, that is, a one-to-one exchange (1 producer and 1 consumer), with a fixed price for each kilowatt-hour. This first test allowed us to identify and prove RTE properties. The simplicity of the unidirectional exchange model did not allow the definition of complex functional properties to show the importance and utility of the *Why3* tool. In a second step, we extended the application under study to an indefinite number of users, and then enriched our specifications. The use of *Why3* is quite suitable for this order of magnitude. In this second version, we have a set of consumers and producers willing to buy or to sell energy. Accordingly, we introduced a simple trading algorithm that matches producers with consumers. In addition to transferring *ether*, users transfer crypto-Kilowatthours to reward consumers consuming locally produced energy. Hence, the system needs to formulate and prove predicates and properties of functions handling various data other than cryptocurrency. For a first trading approach, we adopted, to our case study, an order book matching algorithm [12]. Please refer to [18], the technical report, for the complete BEMP application.

2.2 Why3 features intended for Smart Contracts

Library modelling. *Solidity* is an imperative object-oriented programming language, characterized by static typing[4]. It provides several elementary types that can be combined to form complex types such as booleans, signed, unsigned, and fixed-width integers, settings, and domain-specific types like addresses. Moreover, the address type has primitive functions able to transfer *ether* (`send()`, `transfer()`) or manipulate cryptocurrency balances (`.balance`). *Solidity* contains elements that are not part of the *Why3* language. One could model these as additional types or primitive features. Examples of such types are `uint256` and `address`. For machine integers, we use the range feature of Why3: `type uint256 = <range 0 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFF... >` because it exactly

[4] Ethereum foundation: Solidity, the contract-oriented programming language. https://github.com/ethereum/solidity

represents the set of values we want to represent. Moreover, why3 checks that the constants written by the user of these types are inside the bounds and converts in specifications automatically range types to the mathematical integers, e.g., `int` type. Indeed it is a lot more natural and clearer to express specification with mathematical integers, for example with wrap-around semantic `account = old account - transfer` doesn't express that the account loses money (if the account was empty it could now have the maximum quantity of money).

Based on the same reasoning, we have modelled the type `Int160`, `Uint160` (which characterizes type `uint` in *Solidity*). We also model the `address` type and its members. We choose to encode the private storage (`balance`) by a Hashtable having as a key value an address, and the associated value a `uint256` value. The current value of the balance of addresses would be `balance[address]`. In addition, the `send` function is translated by a `val` function, which performs operations on the `balance` hashtable. Moreover, we model primitive features such as the `modifier` function, whose role is to restrict access to a function; it can be used to model the states and guard against incorrect usage of the contract. In *Why3* this feature would be an exception to be raised if the condition is not respected, or a precondition to satisfy. We will explain it in more details with an example later. Finally, we give a model of *gas*, in order to specify the maximum amount of *gas* needed in any case. We introduce a new type: `type gas = int`. The quantity of *gas* is modelled as a mathematical integer because it is never manipulated directly by the program. This part is detailed later.

It is important to note that the purpose of our work is not to achieve a complete encoding of *Solidity*. The interest is rather to rely on the case study in our possession (which turns out to be written in *Solidity*), and from its contracts, we build our own *Why3* contracts. Therefore, throughout the article, we have chosen to encode only *Solidity* features encountered through our case study. Consequently, notions like `revert` or `delegatecall` are not treated. Conversely, we introduce additional types such as `order` and `order_trading`, which are specific to the BEMP application. The `order` type is a record that contains `orderAddress` which can be a seller or a buyer, `tokens` that express the crypto-Kilowatthours (wiling to buy or to sell), and `price_order`. The `order_trading` type is a record that contains seller ID; `seller_index`, buyer ID; `buyer_index`, the transferred amount `amount_t`, and the trading price `price_t`.

Remark: In our methodology, we make the choice to encode some primitives of *Solidity* but not all. For example, the `send()` function in *Solidity* can fail (return `False`) due to an out-of-gas, e.g. an overrun of 2300 units of *gas*. The reason is that in certain cases the transfer of *ether* to a contract involves the execution of the contract fallback, therefore the function might consume more *gas* than expected. A fallback function is a function without a signature (no name, no parameters), it is executed if a contract is called and no other function matches the specified function identifier, or if no data is supplied. As we made the choice of a *private* blockchain type, all users can be identified and we have control on who can write or read from the blockchain. Thus, the *Why3* `send()` function does not need a fallback execution, it only transfers *ether* from one address to another.

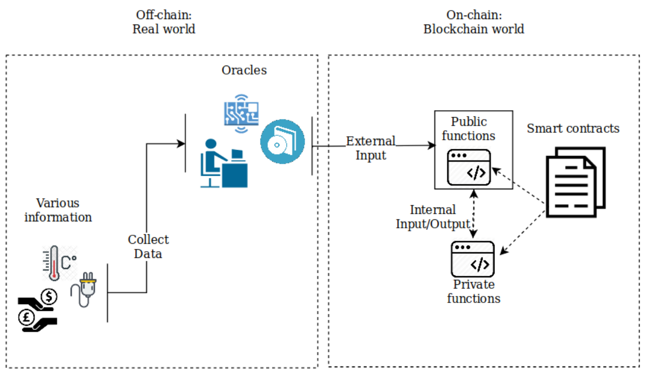


Fig. 2. Link between on-chain and off-chain

The *Why3* `send()` function does not return a boolean, because we require that the transfer is possible (enough *ether* in the sending contract and not too much in the receiving) and we want to avoid Denial-of-service attack [3]. Indeed, if we allow to propagate errors and accept to send to untrusted contracts, it could always make our contract fail and revert. So we cannot prove any property of progress of our contract. In *Tezos* blockchain [14], call to other contracts is postponed to after the execution of the current contract. So another contract should not be able to make the calling contract fail.

Encoding and verifying functions from the BEMP application.

Oracle notions. Developping smart contracts often rely on the concept of *Oracles* [1]. An oracle can be seen as the link between the blockchain and the "real world". Some smart contracts functions have arguments that are external to the blockchain. However, the blockchain does not have access to information from an off-chain data source which is untrusted. Accordingly, the oracle provides a service responsible for entering external data into the blockchain, having the role of a trusted third party. However, questions arise about the reliability of such oracles and accuracy of information. Oracles can have unpredictable behaviour, e.g. a sensor that measures the temperature might be an oracle, but might be faulty; thus one must account for invalid information from oracles. Figure 2 illustrates the three communication stages between various systems in the real world with the blockchain: *(1)* the collection of off-chain raw data; *(2)* this data is collected by oracles; and finally, *(3)* oracles provide information to the blockchain (via smart contracts). Based on this distinction, we defined two types of functions involved in contracts, namely *Private functions* and *Public functions*. We noted that some functions are called internally, by other smart contracts functions, while others are called externally by oracles. Functions that interact with oracles are defined as *public* functions. The proof approach of the two types is different. For the *private* functions one defines pre-conditions and post-conditions, and then we prove that no error can occur and that the function behaves as it should. It is thus not necessary to define exceptions to be raised throughout the program; they are proved to never occur. Conversely, the *public* functions are called by oracles, the behaviour of the function must, therefore, take into account any input values and it is not possible to require conditions upstream of the call. So in contrast, the exceptions are necessary; we use so-called *defensive proof* in order to protect ourselves from the errors that can be generated by oracles. No constraints are applied on post-conditions. Thus, valid data (which does not raise exceptions) received by a public function will satisfy the pre-conditions of the public function that uses it, because pre-conditions are proved.

Methodology of proving BEMP functions. To illustrate our methodology, we take an example from BEMP.

```
function transferFromMarket(address _to, uint _value) onlyMarket returns (
    bool success) {
        if (exportBalanceOf[market] >= _value)
        {/* Transferring _value from market to _to  */}
        else {success = false;
              Error("Tokens couldn't be transferred from market");}}
```

The function allows transferring `_value` (expressing cryptokwh) from the `market` to `_to` address. The mapping `exportBalanceOf[]` stores balances corresponding to addresses that export tokens. The function can be executed solely by the market (the modifier function `onlyMarket`). The program checks if the market has enough tokens to send to `_to`. If this condition is verified, then the transfer is done. If the condition is not verified, the function returns `false` and triggers an `Error` event (a feature that allows writing logs in the blockchain) [5]. This process is internal to the blockchain, there is no external exchange, hence the function is qualified as *private*. According to the modelling approach, we define complete pre-conditions and post-conditions to verify and prove the function. The corresponding *Why3* function is:

```
1 let transferFromMarket (_to : address) (_value : uint) : bool
2    requires {!onlymarket ∧ _value > 0 }
3    requires {marketBalanceOf[market] ≥ _value }
4    requires {importBalanceOf[_to] ≤ max_uint - _value}
5    ensures {(old marketBalanceOf[market]) + (old importBalanceOf[_to]) = marketBalanceOf[
      market] +  importBalanceOf[_to]}
6    = (* The program *)
```

The pre-condition in line 2 expresses the `modifier onlyMarket` function. Note that `marketBalanceOf` is the hashtable that records crypto-Kilowatthours balances associated with market addresses, and `importBalanceOf` is the hashtable that records the amount of crypto-Kilowatthours intended for the buyer addresses. From the specification, we understand the behaviour of the function without referencing to the program. To be executed, `transferFromMarket` must respect RTE and functional properties:

[5] https://media.consensys.net/technical-introduction-to-events-and-logs-in-ethereum-a074d65dd61e

- RTE properties: *(1) Positive values*; a valid amount of crypto-Kilowatthours to transfer is a positive amount (Line 2). *(2) Integer overflow*; no overflow will occur when `_to` receives `_value` (Line 4).
- Functional properties: *(1) Acceptable transfer*; the transfer can be done, if the market has enough crypto-Kilowatthours to send (Line 3). *(2) Successful transfer*; the transaction is completed successfully if the sum of the sender and the receiver balance before and after the execution does not change (Line 5). *(3)* `modifier` *function*; the function can be executed only by the market (Line 2).

The set of specifications is necessary and sufficient to prove the expected behaviour of the function.

The following function illustrates a *Solidity* public function.

```
function registerSmartMeter(string _meterId, address _ownerAddress) onlyOwner
      {  addressOf[_meterId] = _ownerAddress;
         MeterRegistered(_ownerAddress, _meterId);}
```

The function `registerSmartMeters()` is identified by a name (`meterID`) and an owner (`ownerAddress`). Note that all meter owners are recorded in a hashtable `addressOf` associated with a key value `meterID` of the `string` type. The main potential bug in this function is possibly registering a meter twice. When a meter is registered, the function broadcasts an event `MeterRegistered`. Following the modelling rules, there are no pre-conditions, instead, we define exceptions. The corresponding *Why3* function is:

```
1 Exception OnlyOwner, ExistingSmartMeter
2 let registerSmartMeter (meterID : string) (ownerAddress : address)
3     raises { OnlyOwner→ !onlyOwner = False  }
4     raises {ExistingSmartMeter → mem addressOf meterID}
5     ensures { (size addressOf) = (size (old addressOf) + 1 ) }
6     ensures { mem addressOf meterID}
7     = (*The program*)
```

The exception `OnlyOwner` represents the `modifier` function which restricts the function execution to the owner, the caller function. It is not possible to precondition inputs of the function, so we manage exceptional conditions during the execution of the program. To be executed, `registerSmartMeter` must respect RTE and functional properties:

- RTE properties: *Duplicate record*; if a smart meter and its owner is recorded twice, raise an exception (Line 4)
- Functional properties: *(1)* `modifier` *function*; the function can be executed only by the owner, thus we raise `OnlyOwner` when the caller of the function is not the owner (Line 3). *(2) Successful record*; at the end of the function execution, we ensure (Line 5) that a record has made. *(3) Existing record*; the registered smart meter has been properly recorded in the hashtable `addressOf` (Line 6).

The set of specifications is necessary and sufficient to prove the expected behaviour of the function.

Trading contract. The trading algorithm allows matching a potential consumer with a potential seller, recorded in two arrays `buy_order` and `sell_order` taken as parameters of the algorithm. In order to obtain an expected result at the end of the algorithm, properties must be respected. We define specifications that make it possible throughout the trading process. The algorithm is a private function. The Trading contract has no *Solidity* equivalent because it is a function added to the original BEMP project. Below is the set of properties of the function:

```
1  let trading (buy_order : array order) (sell_order : array order) : list order_trading
2        requires { length buy_order > 0 ∧ length sell_order > 0}
3        requires {sorted_order buy_order}
4        requires {sorted_order sell_order}
5        requires {forall j:int. 0 ≤ j < length buy_order → 0 < buy_order[j].tokens }
6        requires {forall j:int. 0 ≤ j < length sell_order → 0 < sell_order[j].tokens  }
7        ensures { correct result (old buy_order) (old sell_order) }
8        ensures { forall l. correct l (old buy_order) (old sell_order) → nb_token l ≤
        nb_token result }
9        ensures { !gas ≤ old !gas + 374 + (length buy_order + length sell_order) * 928 }
10       ensures { !alloc ≤ old !alloc + 35 + (length buy_order + length sell_order) * 384 }
11       = (* The program *)
```

– RTE properties: *positive values*; parameters of the functions must not be empty (empty array) (Line 2), and a trade cannot be done with null or negative tokens (Lines 5, 6).

– Functional requirements: *sorted orders*; the orders need to be sorted in a decreasing way. Sellers and buyers asking for the most expensive price of energy will be at the top of the array (Lines 3, 4).
– Functional properties: *(1) correct trading* (Lines 7, 8); for a trading to be qualified as correct, it must satisfy two properties:
  • the conservation of buyer and seller tokens that states no loss of tokens during the trading process : `forall i:uint. 0 ≤ i < length sell_order → sum_seller (list_trading) i ≤ sell_order[i].tokens`. For the buyer it is equivalent by replacing seller by buyer.
  • a successful matching; a match between a seller and a buyer is qualified as correct if the price offered by the seller is less than or equal to that of the buyer, and that the sellers and buyers are valid indices in the array.

  (2) Best tokens exchange; we choose to qualify a trade as being one of the best if it maximize the total number of tokens exchanged. Line 8 ensures that no correct trading list can have more tokens exchanged than the one resulting from the function. The criteria could be refined by adding that we then want to maximize or minimize the sum of paid (best for seller or for buyer). *(3) Gas consumption*; Lines 9 and 10 ensures that no extra-consumption of gas will happen (see the following paragraph).

Proving the optimality of an algorithm is always challenging (even on paper), and we needed all the proof features of *Why3* such as ghost code and lemma function. Proving fairness is an important property that users of a smart-contract will desire. So we believe it is worth the effort.

Gas consumption proof. Overconsumption of *gas* can be avoided by the *gas* model. Instructions in EVM consume an amount of *gas*, and they are categorized by level of difficulty; e.g., for the set $W_{verylow} = \{ADD,\ SUB,\ ...\}$, the amount to pay is $G_{verylow} =\ 3\ units\ of\ gas$, and for a create operation the amount to pay is $G_{create} =\ 32000\ units\ of\ gas$ [21]. The price of an operation is proportional to its difficulty. Accordingly, we fix for each *Why3* function, the appropriate amount of *gas* needed to execute it. Thus, at the end of the function instructions, a variable `gas` expresses the total quantity of *gas* consumed during the process. We introduce a `val ghost` function that adds to the variable `gas` the amount of *gas* consumed by each function calling `add_gas` (see section 3 for more details on *gas* allocation).

```
1 val ghost add_gas (used : gas) (allocation: int): unit
2     requires { 0 ≤ used ∧ 0 ≤ allocation }
3     ensures  { !gas = (old !gas) + used }
4     ensures  { !alloc = (old !alloc) + allocation }
5     writes   { gas, alloc}
```

The specifications of the function above require *positive values* (Line 2). Moreover, at the end of the function, we ensure that there is no extra *gas* consumption (Lines 3, 4). Line 5 specifies the changing variables. In the trading algorithm, we can see that a lot of allocations are performed, they are in fact not necessary and we could change our code to only allocate a fixed quantity of memory.

3 Compilation and Proof of Gas Consumption

The final step of the approach is the deployment of *Why3* contracts. EVM is designed to be the runtime environment for the smart contracts on the Ethereum blockchain [21]. The EVM is a stack-based machine (word of 256 bits) and uses a set of instructions (called opcodes)[6] to execute specific tasks. The EVM features two memories, one volatile that does not survive the current transaction and a second for storage that does survive but is a lot more expensive to modify. The goal of this section is to describe the approach of compiling *Why3* contracts into EVM code and proving the cost of functions. The compilation[7] is done in three phases: *(1)* compiling to an EVM that uses symbolic labels for jump destination and macro instructions. *(2)* computing the absolute address of the labels, it must be done inside a fixpoint because the size of the jump addresses has an impact on the size of the instruction. Finally, *(3)* translating the assembly code to pure EVM assembly and printed. Most of *Why3* can be translated, the proof-of-concept compiler allows using algebraic datatypes, not nested pattern-matching, mutable records, recursive functions, while loops, integer bounded arithmetic (32, 64,128, 256 bits). Global variables are restricted to mutable records with fields of integers. It could be extended to hashtables using the hashing technique of the keys used in *Solidity*. Without using specific instructions, like for C, *Why3* is extracted to garbage collected language, here all the allocations are done in the

[6] https://ethervm.io
[7] The implementation can be found at http://francois.bobot.eu/fmbc2019/

volatile memory, so the memory is reclaimed only at the end of the transaction. We have not formally proved yet the correction of the compilation, we only tested the compiler using reference interpreter [8] and by asserting some invariants during the transformation. However, we could list the following arguments for the correction: *(1)* the compilation of why3 (ML-language) is straightforward to stack machine. *(2)* The precondition on all the arithmetic operations (always bounded) ensures arithmetic operations could directly use 256bit operations. *(3)* Raise accepted only in public function before any mutation so the fact they are translated into revert does not change their semantics. `try with` are forbidden. *(4)*only immutable datatype can be stored in the permanent store. Currently, only integers can be stored, it could be extended to other immutable datatye by copying the data to and from the store. *(5)* The send function in why3 only modifies the state of balance of the contracts, requires that the transfer is acceptable and never fail, as discussed previously. So it is compiled similarly to the *Solidity* function send function with a gas limit small enough to disallow modification of the store. Additionally, we discard the result. *(6)* The *public* functions are differenciated from *private* one using the attribute `[@ evm:external ]`. The *private* functions doesn't appear in the dispatching code at the contract entry point so they can be called only internally.

The execution of each bytecode instruction has an associated cost. One must pay some *gas* when sending a transaction; if there is not enough *gas* to execute the transaction, the execution stops and the state is rolled back. So it is important to be sure that at any later date the execution of a smart contract will not require an unreasonable quantity of *gas*. The computation of WCET is facilitated in EVM by the absence of cache. So we could use techniques of [6] which annotate in the source code the quantity of *gas* used, here using a function "`add_gas used allocations`". The number of allocations is important because the real *gas* consumption of EVM integrates the maximum quantity of volatile memory used. The compilation checks that all the paths of the function have a cost smaller than the sum of the "`add_gas g a`" on it. The paths of a function are defined on the EVM code by starting at the function-entry and loop-head and going through the code following jumps that are not going back to loop-head.

```
let rec mk_list42 [@ evm:gas_checking] (i:int32) : list int32
 requires { 0 ≤ i }  ensures { i = length result }  variant { i }
 ensures { !gas - old !gas ≤ i * 185 + 113 }
 ensures { !alloc - old !alloc ≤ i * 96 + 32 } =
   if i ≤ 0 then (add_gas 113 32; Nil)
   else (let l = mk_list42 (i-1) in add_gas 185 96; Cons (0x42:int32) l)
```

Currently, the cost of the modification of storage is over-approximated; using specific contract for the functions that modify it we could specify that it is less expansive to use a memory cell already used.

[8] https://github.com/ethereum/go-ethereum

4 Related Work

Since the *DAO* attack, the introduction of formal methods at the level of smart contracts has increased. Static analysis tools are very common to achieve this task. There exists several frameworks, and one of them is called *Raziel*. It is a framework to prove the validity of smart contracts to third parties before their execution in a private way [20]. In that paper, the authors also use a deductive proof approach, but their concept is based on Proof-Carrying Code (PCC) infrastructure, which consists of annotating the source code, thus proofs can be checked before contract execution to verify their validity. Our method does not consist in annotating the *Solidity* source code but in writing the contract in a language designed for program verification in order to tackle harder properties. With a slightly different approach we have *Oyente*. It has been developed to analyze Ethereum smart contracts to detect bugs. In the corresponding paper [15], the authors were able to run *Oyente* on 19,366 existing Ethereum contracts, and as a result, the tool flagged 8,833 of them as vulnerable. Although that work provides interesting conclusions, it uses symbolic execution, analyzing paths, so it does not allow to prove functional properties of the entire application. We can also mention the work undertaken by the *F** community [9] where they use their functional programming language to translate *Solidity* contracts to shallow-embedded F* programs. Just like [5] where the authors perform static analysis by translating *Solidity* contracts into Java using *KeY* [4]. We believe it is easier for the user to be as close as possible to the proof tool, if possible, and in the case of smart-contract the current paper showed it is possible. The initiative of the current paper is directly related to a previous work [19], which dealt with formally verifying the smart contracts application by using model-checking. However, because of the limitation of the model-checker used, ambitious verification could not be achieved (e.g., a model for m consumers and n producers). This present work aims to surpass the limits encountered with model-checking, by using a deductive approach on an Ethereum application using *Why3*.

5 Conclusions

In this paper, we applied concepts of deductive verification to a computer protocol intended to enforce some transaction rules within an Ethereum blockchain application. The aim is to avoid errors that could have serious consequences. Reproducing, with *Why3*, the behaviour of *Solidity* functions showed that *Why3* is suitable for writing and verifying smart contracts programs. The presented method was applied to a use case that describes an energy market place allowing local energy trading among inhabitants of a neighbourhood. The resulting modelling allows establishing a trading contract, in order to match consumers with producers willing to make a transaction. In addition, this last point demonstrates that with a deductive approach it is possible to model and prove the operation of the BEMP application at realistic scale (e.g. matching m consumers with n producers), contrary to model-checking in [19], thus allowing the verifying of more realistic functional properties.

References

1. Ethereum foundation : Ethereum and oracles. https://blog.ethereum.org/2014/07/22/ethereum-and-oracles/
2. Formal verification for solidity contracts. https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts
3. Solidity hacks and vulnerabilities. https://hackernoon.com/hackpedia-16-solidity-hacks-vulnerabilities-their-fixes-and-real-world-examples-f3210eba5148
4. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive software verification-the key book. lncs, vol. 10001 (2016)
5. Ahrendt, W., Bubel, R., Ellul, J., Pace, G.J., Pardo, R., Rebiscoul, V., Schneider, G.: Verification of smart contract business logic (2019)
6. Amadio, R.M., Ayache, N., Bobot, F., Boender, J.P., Campbell, B., Garnier, I., Madet, A., McKinna, J., Mulligan, D.P., Piccolo, M., Pollack, R., Régis-Gianas, Y., Sacerdoti Coen, C., Stark, I., Tranquilli, P.: Certified complexity (cerco). In: Dal Lago, U., Peña, R. (eds.) Foundational and Practical Aspects of Resource Analysis. pp. 1–18. Springer International Publishing, Cham (2014)
7. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts. In: Principles of Security and Trust, pp. 164–186. Springer (2017)
8. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of the 23rd International Conference on Computer Aided Verification. Springer (2011)
9. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Short paper: Formal verification of smart contracts (2016)
10. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S., Mebsout, A.: The alt-ergo automated theorem prover (2008), http://alt-ergo.lri.fr/
11. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper (2014)
12. Domowitz, I.: A taxonomy of automated trade execution systems. Journal of International Money and Finance **12**, 607–631 (1993)
13. Filliâtre, J.C., Paskevich, A.: Why3 – where programs meet provers. In: European Symposium on Programming. pp. 125–128. Springer (2013)
14. Goodman, L.: Tezos: A self-amending crypto-ledger position paper (2014)
15. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. ACM (2016)
16. de Moura, L., Bjørner, N.: Z3, an efficient SMT solver, http://research.microsoft.com/projects/z3/
17. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
18. Nehaï, Z., Bobot, F.: Deductive Proof of Ethereum Smart Contracts Using Why3. Research report, CEA DILS (Apr 2019), https://hal.archives-ouvertes.fr/hal-02108987
19. Nehaï, Z., Piriou, P.Y., Daumas, F.: Model-checking of smart contracts. In: The 2018 IEEE International Conference on Blockchain. IEEE (2018)
20. Sánchez, D.C.: Raziel: Private and verifiable smart contracts on blockchains. Cryptology ePrint Archive, Report 2017/878 (2017), http://eprint.iacr.org/2017/878.pdf, accessed:2017-09-26
21. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)

Statistical Model Checking of RANDAO's Resilience to Pre-computed Reveal Strategies

Musab A. Alturki[1,2] and Grigore Roşu[3,1]

[1] Runtime Verification Inc., Urbana, IL 61801
[2] King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia
`musab.alturki@kfupm.edu.sa`
[3] University of Illinois at Urbana-Champaign, Urbana, IL 61801
`grosu@illinois.edu`

Abstract. RANDAO is a commit-reveal scheme for generating pseudo-random numbers in a decentralized fashion. The scheme is used in emerging blockchain systems as it is widely believed to provide randomness that is unpredictable and hard to manipulate by maliciously behaving nodes. However, RANDAO may still be susceptible to look-ahead attacks, in which an attacker (controlling a subset of nodes in the network) may attempt to pre-compute the outcomes of (possibly many) reveal strategies, and thus may bias the generated random number to his advantage. In this work, we formally evaluate resilience of RANDAO against such attacks. We first develop a probabilistic model in rewriting logic of RANDAO, and then apply statistical model checking and quantitative verification algorithms (using MAUDE and PVESTA) to analyze two different properties that provide different measures of bias that the attacker could potentially achieve using pre-computed strategies. We show through this analysis that unless the attacker is already controlling a sizable percentage of nodes while aggressively attempting to maximize control of the nodes selected to participate in the process, the expected achievable bias is quite limited.

Keywords: RANDAO · Rewriting logic · Maude · Statistical Model Checking · Blockchain

1 Introduction

Decentralized pseudo-random value generation is a process in which participants in a network, who do not necessarily trust each other, collaborate to produce a random value that is unpredictable to any individual participant. It is a core process of many emerging distributed autonomous systems, most prominently proof-of-stake (PoS) consensus protocols, which include the upcoming Ethereum 2.0 (a.k.a. Serenity) protocol [11, 8]. A commonly accepted implementation scheme for decentralized random value generation is a commit-reveal scheme, known as RANDAO (due to Youcai Qian [16]), in which participants first make commitments by sharing hash values of seeds, and then, at a later stage, they reveal their seeds, which can then be used for generating the random value. In a PoS protocol, and in particular in Serenity [11], the scheme is used repeatedly in a sequence of rounds in such a way that the outcome of a round is used as a seed for generating the random value of the following round. Moreover, the scheme is usually coupled with a reward system that incentivizes successful participation and discourages deviations from the protocol. Several other distributed protocols have also adopted this scheme primarily for its simplicity and flexibility.

However, this approach may still be susceptible to *look-ahead* attacks, in which a malicious participant may choose to refrain from revealing his seed if skipping results in randomness that is more favorable to him. In general, a powerful attacker may attempt to pre-compute the outcomes of (possibly many) reveal strategies, which are sequences of reveal-or-skip decisions, and thus may anticipate the effects of his contribution to the process and bias the generated random number to his advantage.

While this potential vulnerability is known and has been pointed to in several works in the literature (e.g. [7, 6, 4]), the extent to which it may be exploited by an attacker and how effective the attack could be in an actual system, such as a PoS system like Serenity, have not yet been thoroughly investigated, besides the exploitability arguments made in [7] and [6], which were based on abstract analytical models. While the high-level analysis given there is useful for gaining a foundational understanding of the vulnerability and the potential of the attack, a lower-level formalization that captures the interactions of the different components of the RANDAO process and the environment could provide deeper insights into how realizable the attack is in an actual system.

In this work, we develop a computational model of the RANDAO scheme as a probabilisitic rewrite theory [12, 1] in rewriting logic [13] to formally evaluate resilience of RANDAO to pre-computed reveal strategies. The model gives a formal, yet natural, description of (possibly different designs of) the RANDAO process and the environment. Furthermore, the model is both *timed*, capturing timing of events in the process, and *probabilistic*, modeling randomized protocol behaviors and environment uncertainties. Being executable, the model facilitates automated formal analysis of *quantitative properties*, specified as real-valued formulas in QuaTEx (Quantitative Temporal Expressions Logic) [1], through efficient statistical model checking and quantitative analysis algorithms using both Maude [9] (a high-performance rewriting system) and PVeStA [2] (a statistical verification tool that interfaces with Maude). Using the model, we analyze two properties that provide different measures of bias that the attacker could potentially achieve using pre-computed strategies: (1) the *matching score*, which is the expected number of proposers that the attacker controls, and (2) the *last-word score*, which is the length of the longest tail of the proposers list that the attacker controls.

We show through this analysis that unless the attacker is already controlling a sizable portion of validators and is aggressively attempting to maximize the number of last compromised proposers in the proposers list, or what we call the *compromised tail* of the list, the expected achievable bias of randomness of the RANDAO scheme is quite limited. However, an aggressive attacker who can afford to make repeated skips for very extended periods of time (e.g. in thousands of rounds), or an attacker who already controls a fairly large percentage (e.g. more than 30%) of participants in the network will have higher chances of success.

The rest of the paper is organized as follows. In Section 2, we quickly review rewriting logic and statistical model checking. In Section 3, we introduce in some detail the RANDAO scheme. This is followed in Section 4 by a description of our model of RANDAO in rewriting logic. Section 5 the analysis properties and results. The paper concludes with a discussion of future work in Section 6.

2 Background

Rewriting Logic [14] is a general logical and semantic framework in which systems can be formally specified and analyzed. A unit of specification in Rewriting Logic is a *rewrite theory* $\mathcal{R}$, which formally describes a concurrent system including its static structure and dynamic behavior. It is a tuple $(\Sigma, E \cup A, R)$ consisting of: (1) a membership equational logic (MEL) [15] signature Σ that declares the kinds, sorts and operators to be used in the specification; (2) a set E of Σ-sentences, which are universally quantified Horn clauses with atoms that are either equations ($t = t'$) or memberships ($t : s$); (3) A a set of equational axioms, such as commutativity, associativity and/or identity axioms; and (4) a set R of rewrite rules $t \longrightarrow t'$ if C specifying the computational behavior of the system (where C is a conjunction of equational or rewrite conditions). Operationally, if there exists a substitution θ such that $\theta(t)$ matches a subterm s in the state of the system, and $\theta(C)$ is satisfied, then s may rewrite to $\theta(t')$. While the MEL sub-theory $(\Sigma, E \cup A)$ specifies the user-defined syntax and equational axioms defining the system's state structure, a rewrite rule in R specifies a *parametric transition*, where each instantiation of the rule's variables that satisfies its conditions yields an actual transition (See [5] for a detailed account of generalized rewrite theories).

Probabilistic rewrite theories extend regular rewrite theories with probabilistic rules [17, 1]. A probabilistic rule ($t \longrightarrow t'$ if C with probability π) specifies a transition that can be taken with a probability that may depend on a probability distribution function π parametrized by a t-matching substitution satisfying C. Probabilistic rewrite theories unify many different probabilistic models and can express systems involving both probabilistic and nondeterministic features.

Maude [9] is a high-performance rewriting logic implementation. An equational theory $(\Sigma, E \cup A)$ is specified in Maude as a functional module, which may consist of sort and subsort declarations for defining type hierarchies, operator declarations, and unconditional and conditional equations and memberships. Operator declarations specify the operator's syntax (in mixfix notation), the number and sorts of the arguments and the sort of its resulting expression. Furthermore, equational attributes such as associativity and commutativity axioms may be specified in brackets after declaring the input and output sorts. A rewrite theory is specified as a *system module*, which may additionally contain rewrite rules declared with the `rl` keyword (`crl` for conditional rules).

Furthermore, probabilistic rewrite theories, specified as system modules in Maude [9], can be simulated by sampling from probability distributions. Using PVeStA [2], randomized simulations generated in this fashion can be used to statistically model check quantitative properties of the system. These properties are specified in a rich, quantitative temporal logic, QuaTEx [1], in which real-valued state and path functions are used instead of boolean state and path predicates to quantitatively specify properties about probabilistic models. QuaTEx supports parameterized recursive function declarations, a standard conditional construct, and a *next* modal operator $\bigcirc$, allowing for an expressive language for real-valued temporal properties (Example QuaTEx expressions appear in Section 5). Given a QuaTEx path expression and a Maude module specifying a probabilistic rewrite theory, statistical quantitative analysis is performed by estimating the *expected value* of the path expression against computation paths obtained by Monte Carlo simulations. More details can be found in [1].

3 The RANDAO Scheme

The RANDAO scheme [16] is a commit-reveal scheme consisting of two stages: (1) the commit stage, in which a participant p_i first commits to a seed s_i (by announcing the hash of the seed h_{s_i}), and then (2) the reveal stage, in which the participant p_i reveals the seed s_i. The sequence of revealed seeds $s_0, s_1, \cdots, s_{n-1}$ (assuming n participants) are then used to compute a new seed s (e.g. by taking the XOR of all s_i), which is then used to generate a random number.

In the context of the Serenity protocol [11], the RANDAO scheme proceeds in rounds corresponding to epochs in the protocol. At the start of an epoch i, the random number r_{i-1} generated in the previous round (in epoch $i-1$) is used for sampling from a large set of validators participating in the protocol an ordered list of block proposers $p_0, p_1, \cdots p_{k-1}$, where k is the cycle length of the protocol (a fixed number of time slots constituting one epoch in the protocol). Each proposer p_i is assigned the time slot i of the current round (epoch). During time slot i, the proposer p_i is expected to submit the pair (c_{p_i}, s_{p_i}), with c_{p_i} a commitment on a seed to be used for the next participation in the game (in some future round when p_i is selected again as a proposer), and s_{p_i} the seed to which p_i had previously committed in the last participation in the game (or when p_i first joined the protocol's validator set). The RANDAO contract keeps track of successful reveals in the game, which are those reveals that arrive in time and that pass the commitment verification step. Towards the end of an epoch i, the RANDAO contract combines the revealed seeds in this round by computing their XOR s_i, which is used as the seed for the next random number r_{i+1} to be used in the next round $i+1$. To discourage deviations from the protocol and encourage proper participation, the RANDAO contract penalizes proposers who did not successfully reveal (by discounting their Ether deposits) and rewards those proposers who have been able to successfully reveal their seeds (by distributing dividends in Ether).

4 A Rewriting Model of RANDAO

We use Rewriting Logic [14], and its probabilistic extensions [12, 1], to build a generic and executable model of the RANDAO scheme. The model is specified as a probabilistic rewrite theory $\mathcal{R} = (\Sigma_{\mathcal{R}}, E_{\mathcal{R}} \cup A_{\mathcal{R}}, R_{\mathcal{R}})$, implemented in MAUDE as a system module. By utilizing different facilities provided by its underlying formalism, the model $\mathcal{R}$ is both *probabilistic*, specifying randomized behaviors and environment uncertainties, and *real-time*, capturing time clocks and message transmission delays. Furthermore, the model is *parametric* to a number of parameters, such as the attack probability, the size of the validator set and the network latency, to enable capturing different scenarios and attack behaviors.

In this section, we describe generally the most fundamental parts of the model. A more detailed description of the model can be found in [3].

4.1 Protocol State Structure

The structure of the model, specified by the MEL sub-theory $(\Sigma_{\mathcal{R}}, E_{\mathcal{R}} \cup A_{\mathcal{R}})$ of $\mathcal{R}$, is based on a representation of actors in rewriting logic, which builds on its underlying object-based modeling facilities. In this model, the state of the protocol is a *configuration* consisting of a multiset of actor objects and messages in transit. Objects communicate asynchronously by message passing. An object is a term of the form `<name: O | A >`, with `O` the actor object's unique name (of the sort `ActorName`) and `A` its set of attributes, constructed by an associative and commutative comma operator `_,_` (with `mt` as its identity element). Each attribute is a name-value pair of the form `attr : value`. A message destined for object `O` with payload `C` is represented by a term of the form `O <- C`, where the payload `C` is a term of the sort `Content`.

Objects The three most important objects in the model are: (1) the blockchain object, (2) the RANDAO contract object, and (3) the attacker object.

The blockchain object. This object, identified by the actor name operator `bc`, models abstractly the public data maintained in a blockchain:

```
<name: bc | vapproved: VHL,  vapproved-size: N,
            vpending:  VHL', vpending-size:  N',
            seed: S >
```

The object maintains a list of validator records of all approved and participating validators in the system in an attribute `vapproved`, with its current length in the `vapproved-size` attribute. As new validators arrive and request to join the system, the blockchain object accumulates these incoming requests as a growing list of validator records in its attribute `vpending`, along with its current size in the attribute `vpending-size`. Finally, this object maintains the seed value that was last computed by the previous round of the game in its `seed` attribute.

The RANDAO object. This object, identified by the operator `r`, models a RANDAO contract managing the RANDAO process:

```
<name: r | status:    U, balance:    N,  precords: PL,
           prop-size: M, prop-ilist: IL, pnext: I >
```

It maintains a `status` attribute, indicating its current state of processing, and a `balance` attribute, keeping track of the total contract balance. Moreover, the object manages the proposers list for the current round of the game using the attributes `prop-ilist`, a list of indices identifying the proposers, and `precords`, a list of proposer records of the form `[v(I), B]` with `B` a Boolean flag indicating whether the proposer `v(I)` has successfully revealed. Additionally, the size of the proposers list is stored in `prop-size`. Finally, the object also keeps track of the next time slot (in the current round) to be processed in the attribute `pnext`.

The attacker object. The attacker is modeled by the attacker object, identified by the operator `a`:

```
<name: a | vcomp:    CVL, vcomp-ilist: IL, vcomp-size: N,
           strategy: G >
```

The full list of the compromised validator indices is maintained by the attacker object in the attribute `vcomp-ilist`. This list is always a sublist of the active validators maintained by the blockchain object above. Its length is maintained in the attribute `vcomp-size`. Since in every round of the game, a portion of validators selected as proposers may be compromised, the attacker object creates compromised validator records for all such validators to assign them roles for the round and maintains these records in its attribute `vcomp`. If any one of these compromised validators is at the head of the longest compromised tail of the proposers list, the computed reveal strategy (whenever it becomes ready during the current round) is recorded in the attribute `strategy`.

The Scheduler In addition to objects and messages, the state (configuration) includes a *scheduler*, which is responsible for managing time and the scheduling of message delivery. The scheduler is a term of the form `{T | L}`, with `T` the current global clock value of the configuration and `L` a time-ordered list of scheduled messages, where each such message is of the form `[T,M]`, representing a message `M` scheduled for processing at time `T`. As time advances, scheduled messages in `L` are delivered (in time-order) to their target objects, and newly produced messages by objects are appropriately scheduled into `L`.

4.2 Protocol Transitions

The protocol's state transitions are modeled using the (possibly conditional and/or probabilistic) rewrite rules $R_{\mathcal{R}}$ of the rewrite theory $\mathcal{R} = (\Sigma_{\mathcal{R}}, E_{\mathcal{R}} \cup A_{\mathcal{R}}, R_{\mathcal{R}})$. The rules specify: (1) the actions of the RANDAO contract, which are advancing the time slot, advancing the round and processing validator reveals, and (2) the behaviors of both honest and compromised validators. For space consideration, we only list and describe the rule for advancing the time slot below, while omitting some of the details. Complete descriptions of all the rules can be found in the extended report [3].

The transition for advancing the time slot specifies the mechanism with which the RANDAO contract object checks if a successful reveal was made by the proposer assigned for the current time slot:

```
rl [RAdvanceSlot] :
<name: bc | vapproved-size: N, vpending-size: N',
             seed: S, AS >
<name: r | status: ready, precords: ([ VID , B ] ; CL),
             prop-ilist: IL, pnext: K, AS' >
{ TG | SL } (RID <- nextSlot(L)) ...
=>
<name: bc | vapproved-size: N, vpending-size: N',
             seed: S, AS >
if L > #CYCLE-LENGTH then
  <name: r | status: processing,
               precords: ([ VID , B ] ; CL),
               prop-ilist:
             sampleIndexList(N + N', #CYCLE-LENGTH, S, nilIL),
               pnext: 1, AS' >
  { TG | SL } (RID <- nextRound)
else
  if L == K then
    <name: r | status: ready,
                 precords: ([ VID , B ] ; CL),
                 prop-ilist: IL, pnext: K, AS' >
  else
    <name: r | status: ready,
                 precords: (CL ; [ VID , false ]),
                 prop-ilist: IL, pnext: s(K), AS' >
  fi
  insert({ TG | SL }, [TG + 1.0, (RID <- nextSlot(s(L)))])
fi ... .
```

When the current time slot `L` is about to end, the message `nextSlot(L)` becomes ready for the RANDAO object to consume, which initiates the process of advancing the state of the protocol to the next slot. There are three cases that need to be considered depending on the value of `L`:

1. `L > #CYCLE-LENGTH`, meaning that the message's time slot number exceeds the number of slots in a round (slot numbering begins at 1), and thus, the protocol has already processed all slots of the current round, and progressing to the next slot would require advancing the the current round of the game first. Therefore, The RANDAO contract object changes its status to `processing` and samples a new list of proposers for the next round using the seed `S` that was computed in the current round. The object resets the

time slot count back to 1 and emits a self-addressed, zero-delay `nextRound` message.
2. `L == K`, where `K` is the next-slot number stored in the RANDAO object, which means that the slot number `K` was already advanced by successfully processing a reveal some time earlier during this slot's time window. In this case, the state is not changed and a `nextSlot(s(L))` message (with `s` the successor function) is scheduled as normal to repeat this process for the next time slot.
3. Otherwise, the slot number `K` stored in the object has not been advanced before and, thus, either a reveal for the current time slot `L` was attempted and failed or that a reveal was never received. In both cases, the RANDAO object records that as a failure in the proposers record list, advances the slot number `K` and schedules a `nextSlot(s(L))` message in preparation for the next time slot.

These cases are specified by the nested conditional structure shown in the rule.

5 Statistical Verification

We use the model $\mathcal{R}$ to formally and quantitatively evaluate how much an attacker can bias randomness of the RANDAO process assuming various attacker models and protocol parameters. In the analysis presented below, we assume a 95% confidence interval with size at most 0.02. We also assume no message drops and random message transmission delays in the range [0.0, 0.1] time units (so reveals, if made, are guaranteed to arrive on time).

5.1 Matching Score (MS)

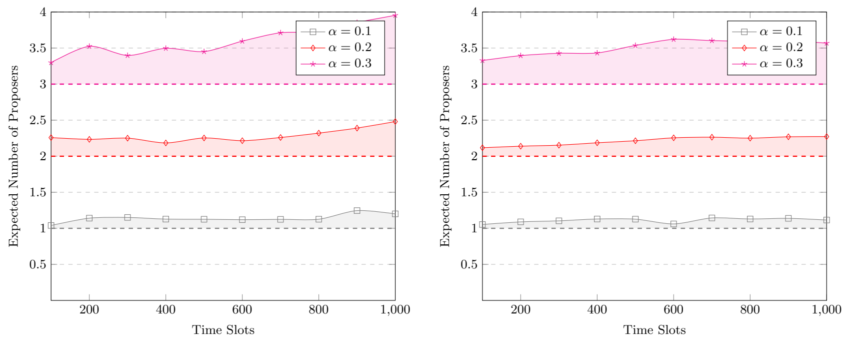
The *Matching score* (MS) is the number of attacker-controlled validators selected as proposers in a round of the RANDAO process. The baseline value for MS (assuming no attack) is given by the expectation of a binomial random variable X with success probability p (the probability of a validator being compromised) in k repeated trials (k is the length of the proposers list), which is:

$$EX[X] = kp \tag{1}$$

As a *temporal* formula in QuaTEx, the property MS is expressed as:

$$\begin{aligned} ms(t) = {} & \underline{\textbf{if}}\ \mathit{time}() > t\ \underline{\textbf{then}}\ \mathit{countCompromised}() \\ & \underline{\textbf{else}}\ \bigcirc\, ms(t)\ \underline{\textbf{fi}}\ ; \\ & \underline{\textbf{eval}}\ \mathbf{E}[ms(t_0)] \end{aligned} \tag{2}$$

$ms(t)$ is a recursively defined path expression that uses two state functions: (1) *time*(), which evaluates to the time value of the current state of the protocol (given by the scheduler object), and (2) *countCompromised*(), which evaluates



(a) MS (10x500) (b) MS (10x1000)

Fig. 1. The expected number of attacker-controlled proposers in the proposers list against execution time in time slots, assuming the attacker is attempting to maximize the number of compromised proposers. The dashed lines represent the base values (with no active attack) computed using Equation (1). The shaded areas visualize the expected bias achievable by the attacker for the three different attack probabilities plotted. We assume a proposers list of size 10, and a validator set of size (a) 10×500 and (b) 10×1000.

to the number of compromised proposers in the current state of the RANDAO object. Therefore, given an execution path, the path expression $ms(t)$ evaluates to $countCompromised()$ in the current state if the protocol run is complete (reached the time limit); otherwise, it returns the result of evaluating itself in the next state, denoted by the next-state temporal operator $\bigcirc$. The number of compromised proposers that an attacker achieves (on average) within the time limit specified can be approximated by estimating the expected value of the formula over random runs of the protocol, denoted by the query $\underline{\textbf{eval}}\ \mathbf{E}[ms(t_0)]$.

The analysis results for MS are plotted in the charts of Figure 1. We use the notation $a \times b$ to denote the fact that the length of the proposers list (`CYCLE-LENGTH`) is a and that there are a total of $a \times b$ participating validators in the configuration[4]. The dashed lines in the charts represent the base values (with no active attack) computed using Equation (1) for different attack probabilities p, while the plotted data points are the model's estimates.

As the charts show, the attacker can reliably but minimally bias randomness with this strategy. This, however, assumes that the attacker is able to afford all the skips that will have to be made in the process, since only after about 80 rounds or so, the attacker is able to gain an advantage of about 20% (over the

[4] The specific values for a and b used in this section and Section 5.2 are chosen so that the total size of the validator set $a \cdot b$ is large enough relative to the length of the proposers list a so that the probability of picking a compromised proposer stays the same (recall that the attack probability is fixed), while not too large to allow efficient analysis. This has the important consequence that the analysis results obtained are representative of actual setups (where the set of validators is much larger than that of the proposers), regardless of the exact proportion of proposers to validators.

baseline). Nevertheless, an attacker that already controls a significant portion of the validators can capitalize on that to speed up his gains, as can be seen from the $p = 0.3$ attacker at around 100 rounds, compared with the weaker attackers. Furthermore, by comparing the charts in Figure 1, we note that results obtained for different proportions of proposers to validators are generally similar.

5.2 Last-Word Score (LWS)

This is the length of the longest attacker-controlled tail of the proposers list in a round of the RANDAO process. We first compute a baseline value for LWS (assuming no attack). Let a be the event of picking an attacker-controlled validator, which has probability p, and b the event of picking an honest validator b, having probability $(1 - p)$. Let the length of the proposers list be k. A compromised tail in the proposers list corresponds to either a sequence of events a of length $j < k$ followed immediately by exactly one occurrence of event b, or a sequence of events a of length exactly k (the whole list is controlled by the attacker). Therefore, letting X be a random variable corresponding to the length of the longest compromised tail, we have:

$$Pr[X = i] = \begin{cases} p^i(1-p) & i < k \\ p^i & i = k \end{cases}$$

Therefore, the expected value of X is

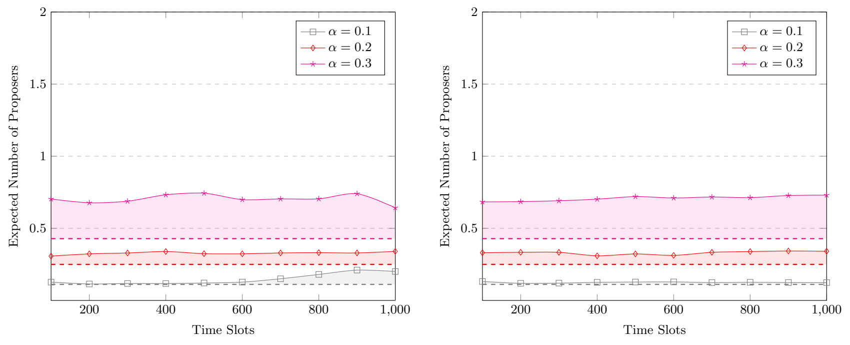
$$EX[X] = \sum_{i=0}^{k-1} i \cdot p^i(1-p) + k \cdot p^k \tag{3}$$

We then specify the property LWS using the following formula:

$$\begin{aligned} lws(t) = &\ \underline{\mathbf{if}}\ time() > t\ \underline{\mathbf{then}}\ countCompromisedTail() \\ &\ \underline{\mathbf{else}}\ \bigcirc\ lws(t)\ \underline{\mathbf{fi}}\ ; \\ &\ \underline{\mathbf{eval}}\ \mathbf{E}[lws(t_0)] \end{aligned} \tag{4}$$

The formula uses the state function *countCompromisedTail*(), which counts the number of proposers in the longest compromised tail in the proposers list of the current state of the RANDAO object. As before, estimating the expectation expression $\mathbf{E}[lws(t_0)]$ gives an approximation of the expected length of the longest compromised tail that an attacker can achieve within the specified time limit.

The results are plotted in the charts of Figure 2. As Figure 2 shows, maximizing the length of the compromised tail can result in a steady and reliable effect on the proposers list. As the attack probability increases, the bias achieved can be greater within shorter periods of time. For example, at around 60 rounds, the bias achieved by a 0.1 attacker is negligible, while a 0.2 attacker is expected to achieve 20% gains over the baseline (at around 0.32 compared with 0.25), and a 0.3 attacker achieves 60% gains (at around 0.7 compared with 0.43). Nevertheless, even at high attack rates, the charts do not show strong increasing trends, suggesting that any gains more significant than those would require applying reveal strategies for very extended periods of time.



(a) LWS (10x500) (b) LWS (10x1000)

Fig. 2. The expected number of attacker-controlled proposers in the proposers list against execution time in time slots, assuming the attacker is attempting to maximize the length of the compromised tail. The dashed lines represent the base values (with no active attack) computed using Equation (3). The shaded areas visualize the expected bias achievable by the attacker for the three different attack probabilities plotted. We assume a proposers list of size 10, and a validator set of size (a) 10×500 and (b) 10×1000.

6 Conclusion

We presented an executable formalization of the commit-reveal RANDAO scheme as a probabilistic rewrite theory in rewriting logic. Through its specification in MAUDE, we used the model to analyze resilience of RANDAO against precomputed reveal strategies by defining two quantitative measures of achievable bias: the matching score (MS) and the last-word score (LWS), specified as temporal properties in QuaTEx and analyzed using statistical model checking and quantitative analysis with PVESTA. Further analysis could consider other scenarios with dynamic validator sets, unreliable communication media and extended network latency. Furthermore, the analysis presented does not explicitly quantify the costs to the attacker, which can be an important economic defense against mounting these reveal strategies. An extension of the model could keep track of the number of skips, or specify a limit on these skips, so that the success of an attack strategy can be made relative to the cost of executing it. Finally, a holistic approach to analyzing quantitative properties of Serenity looking into availability and attack resilience properties makes for an interesting longer-term research direction.

Acknowledgements. We thank Danny Ryan and Justin Drake from the Ethereum Foundation for their very helpful comments. This work was performed under the first Ethereum Foundation security grant "Casper formal verification"[10].

References

1. Agha, G., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. Electronic Notes in Theoretical Computer Science **153**(2), 213–239 (2006)
2. Alturki, M.A., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) Algebra and Coalgebra in Computer Science, Lecture Notes in Computer Science, vol. 6859, pp. 386–392. Springer Berlin / Heidelberg (2011)
3. Alturki, M.A., Roşu, G.: Statistical model checking of RANDAO's resilience against pre-computed reveal strategies. Tech. rep., University of Illinois at Urbana Champaign (November 2018), http://hdl.handle.net/2142/102076
4. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: In proceedings of Crypto 2018. pp. 757–788 (2018)
5. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theor. Comput. Sci. **360**(1-3), 386–414 (2006)
6. Buterin, V.: Randao beacon exploitability analysis, round 2 (November 2018), https://ethresear.ch/t/randao-beacon-exploitability-analysis-round-2/1980
7. Buterin, V.: Rng exploitability analysis assuming pure randao-based main chain (November 2018), https://ethresear.ch/t/rng-exploitability-analysis-assuming-pure-randao-based-main-chain/1825
8. Buterin, V.: Validator ordering and randomness in pos (November 2018), https://vitalik.ca/files/randomness.html
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework, Lecture Notes in Computer Science, vol. 4350. Springer-Verlag, Secaucus, NJ, USA (2007)
10. Foundation, E.: Announcing beneficiaries of the ethereum foundation grants (11 2018), https://blog.ethereum.org/2018/03/07/announcing-beneficiaries-ethereum-foundation-grants
11. Foundation, E.: Ethereum 2.0 spec—casper and sharding (11 2018), https://github.com/ethereum/eth2.0-specs/blob/master/specs/beacon-chain.md
12. Kumar, N., Sen, K., Meseguer, J., Agha, G.: A rewriting based model for probabilistic distributed object systems. In: Proc. of FMOODS '03. Lecture Notes in Computer Science, vol. 2884, pp. 32–46. Springer (2003)
13. Meseguer, J.: Rewriting as a unified model of concurrency. In: Proceedings of the Concur'90 Conference, Amsterdam, August 1990. Lecture Notes in Computer Science, vol. 458, pp. 384–400. Springer (1990)
14. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992). https://doi.org/http://dx.doi.org/10.1016/0304-3975(92)90182-F
15. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) Proc. WADT'97. Lecture Notes in Computer Science, vol. 1376, pp. 18–61. Springer (1998)
16. Qian, Y.: Randao: A dao working as rng of ethereum (November 2018), https://github.com/randao/randao/
17. Sen, K., Kumar, N., Meseguer, J., Agha, G.: Probabilistic rewrite theories: Unifying models, logics and tools. Tech. Rep. UIUCDCS-R-2003-2347, University of Illinois at Urbana Champaign (May 2003)

Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts

Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson

Nomadic Labs, Paris, France
{first_name.last_name}@nomadic-labs.com

Abstract. Tezos is a blockchain launched in June 2018. It is written in OCaml and supports smart contracts. Its smart contract language is called Michelson and it has been designed with formal verification in mind. In this article, we present Mi-Cho-Coq, a Coq framework for verifying the functional correctness of Michelson smart contracts. As a case study, we detail the certification of a Multisig contract with the Mi-Cho-Coq framework.

Keywords: Certified programming · Programming languages · Blockchains · Smart contracts.

1 Introduction to Tezos

Tezos is a public blockchain launched in June 2018. It is mostly implemented in OCaml [18] and its code is open source [3]. Like Ethereum, Tezos is an account based smart contract platform. This section is a high-level broad overview of Tezos to distinguishe it from similar projects like Bitcoin and Ethereum.

Consensus algorithm Unlike Bitcoin and Ethereum, Tezos' **consensus algorithm** is based on a **Proof-of-Stake** algorithm [2]: rights to produce new blocks are given to accounts that own a stake. More precisely, there is a delegation mechanism and the block-producing rights of each account are given in probabilistic proportion to the number of tokens that have been *delegated* to this account. Block producers have to make a security deposit that is slashed if their behaviour looks malicious, for example if they produce two different blocks for the same level (double spending attack).

On-chain voting Another key point of Tezos is its **on-chain governance mechanism**. The codebase can be changed by a vote of the token holders via their delegates. This helps preventing divisions amongst the community that could lead to forks. Only a delimited part of the codebase, named the *economic ruleset* or the *economic protocol* [15,16], can be changed. This part contains the rules that define what a valid transaction is, what a valid block is, as well as how to choose between multiple chains. Thus, the economic ruleset contains, amongst other things, the consensus algorithm, the language for smart contracts and also the voting rules [4]. It does not contain the network and storage layers. If a proposal is accepted, nodes need not to stop and restart: the new code is downloaded from other peers, dynamically compiled and hot swapped. At the moment, the voting procedure lasts approximately three months but that could be changed in the future via a vote.

Focus on formal verification Our long-term ambition is to have certified code in the whole Tezos codebase[1] as well as certified smart contracts. The choice of OCaml as an implementation language is an interesting first step: OCaml gives Tezos good static guarantees since it benefits from OCaml's strong type system and memory management features. Furthermore, formally verified OCaml code can be produced by a variety of tools such as F* [22], Coq [23], Isabelle/HOL [19], Why3 [14], and FoCaLiZe [20]. Another specificity of Tezos is the use of formally verified cryptographic primitives. Indeed the codebase uses the HACL* library [24], which is certified C code extracted from an implementation of Low*, a fragment of F*. This article presents Mi-Cho-Coq, a framework for formal verification of Tezos smart contracts, written in the Michelson programming language. It is organized as follows: Section 2 gives an overview of the Michelson smart contract language, the Mi-Cho-Coq framework is then presented in Section 3, a case study on a Multisig smart contract is then conducted in Section 4, Section 5 presents some related workd and finally Section 6 concludes the article by listing directions for future work.

The Mi-Cho-Coq framework, including the multisig contract described in Section 4, is available at `https://gitlab.com/nomadic-labs/mi-cho-coq/tree/FMBC_2019`.

2 Overview of Michelson

Smart contracts are Tezos accounts of a particular kind. They have a private access to a memory space on the chain called the *storage* of the smart contract, each transaction to a smart contract account contains some data, the *parameter* of the transaction, and a *script* is run at each transaction to decide if the transaction is valid, update the smart contract storage, and possibly emit new operations on the Tezos blockchain.

Michelson is the language in which the smart contract scripts are written. The Michelson language has been designed before the launch of the Tezos blockchain. The most important parts of the implementation of Michelson, the typechecker and the interpreter, belong to the economic ruleset of Tezos so the language can evolve through the Tezos amendment voting process.

[1] Note that since code changes must be approved by the Tezos community, we can only propose a certified implementation of the economic ruleset.

2.1 Design rationale

Smart contracts operate in a very constrained context: they need to be expressive, evaluated efficiently, and their resource consumption should be accurately measured in order to stop the execution of programs that would be too greedy, as their execution time impacts the block construction and propagation. Smart contracts are non-updatable programs that can handle valuable assets, their is thus a need for strong guarantees on the correctness of these programs.

The need for efficiency and more importantly for accurate account of resource consumption leans toward a low-level interpreted language, while the need for contract correctness leans toward a high level, easily auditable, easily formalisable language, with strong static guarantees.

To satisfy these constraints, Michelson was made a Turing-complete, low level, stack based interpreted language (*à la* Forth), enabling the resource measurement, but with some high level features *à la* ML: polymorphic products, options, sums, lists, sets and maps data-structures with collection iterators, cryptographic primitives and anonymous functions. Contracts are pure functions that take a stack as input and return a stack as output. This side-effect free design is an asset for the conception of verification tools.

The language is statically typed to ensure the well-formedness of the stack at any point of the program. This means that if a program is well-typed, and if it is being given a well typed stack that matches its input expectation, then at any point of the program execution, the given instruction can be evaluated on the current stack.

Moreover, to ease the formalisation of Michelson, ambiguous or hidden behaviours have been avoided. In particular, unbounded integers are used to avoid arithmetic overflows and division returns an option (which is `None` if and only if the divisor is 0) so that the Michelson programmer has to specify the behaviour of the program in case of division by 0; she can however still *explicitly* reject the transaction using the **FAILWITH** Michelson instruction.

2.2 Quick tour of the language

The full language syntax, type system, and semantics are documented in [1], we give here a quick and partial overview of the language.

Contracts' shape A Michelson smart contract script is written in three parts: the parameter type, the storage type, and the code of the contract. Contract's code consists in one block of code that can only be called with one parameter, but multiple entry points can be encoded by branching on a nesting of sum types and multiple parameters can be paired into one.

When the contract is originated on the chain, it is bundled with a data storage which can then only be changed by a contract successful execution. The parameter and the storage associated to the contract are paired and passed to the contract's code at each execution, it has to return a list of operations and the updated storage.

Seen from the outside, the type of the contract is the type of its parameter, as it is the only way to interact with it.

Michelson Instructions As usual in stack-based languages, Michelson instructions take their parameters on the stack.All Michelson instructions are typed as a function going from the expected state of the stack, before the instruction evaluation, to the resulting stack. For example, the **AMOUNT** instruction used to obtain the amount in μtez of the current transaction has type `'S` $\rightarrow$ `mutez:'S` meaning that for any stack type `'S`, it produces a stack of type `mutez:'S`. Some instructions, like comparison or arithmetic operations, exhibit non-ambiguous ad-hoc polymorphism: depending on the input arguments type, a specific implementation of the instruction is selected, and the return type is fixed. For example **SIZE** has the following types:

```
bytes:'S → nat:'S          set 'elt:'S → nat:'S
string:'S → nat:'S         map 'key 'val:'S → nat:'S
                           list 'elt:'S → nat:'S
```

While computing the size of a string or an array of bytes is similarly implemented, under the hood, the computation of map size has nothing to do with the computation of string size.

Finally, the contract's code is required to take a stack with a pair *parameter-storage* and returns a stack with a pair *operation list-storage*:
`(parameter_ty*storage_ty):[]` $\rightarrow$ `(operation list*storage_ty):[]`.

The operations listed at the end of the execution can change the delegate of the contract, originate new contracts, or transfer tokens to other addresses. They will be executed right after the execution of the contract. The transfers can have parameters and trigger the execution of other smart contracts: this is the only way to perform *inter-contract* calls.

A short example - the Vote contract. We want to allow users of the blockchain to vote for their favorite formal verification tool. In order to do that, we create a smart-contract tasked with collecting the votes. We want any user to be able to vote, and to vote as many times as they want, provided they pay a small price (say 5 *tez*). We originate the contract with the names of a selection of popular tools: Agda, Coq, Isabelle and K_framework, which are placed in the long-term storage of the contract, in an associative map between the tool's name and the number of registered votes (of course, each tool starts with 0 votes).

In the figure 1a, we present a voting contract, annotated with the state of the stack after each line of code. When actually writing a Michelson contract, development tools (including an Emacs Michelson mode) can interactively, for any point of the code, give the type of the stack provided by the Michelson typecheck of a Tezos node.

Let's take a look at our voting program: First, the description of the storage and parameter types is given on lines `1-2`. Then the code of the contract is given. On line `5`, **AMOUNT** pushes on the stack the amount of (in μtez) sent to the contract address by the user. The threshold amount (5tez) is also pushed on the stack on line `6` and compared to the amount sent: **COMPARE** pops the two top

```
 1  storage (map string int); # candidates
 2  parameter string; # chosen
 3  code {
 4    # (chosen, candidates):[]
 5    AMOUNT; # amount:(chosen, candidates):[]
 6    PUSH mutez 5000000; COMPARE; GT;
 7    # (5 tez > amount):(chosen, candidates):[]
 8    IF { FAIL } {}; # (chosen, candidates):[]
 9    DUP; DIP { CDR; DUP };
10    # (chosen, candidates):candidates:candidates:[]
11    CAR; DUP; # chosen:chosen:candidates:candidates:[]
12    DIP { # chosen:candidates:candidates:[]
13          GET; ASSERT_SOME;
14          # candidates[chosen]:candidates:[]
15          PUSH int 1; ADD; SOME
16          # (Some (candidates[chosen]+1)):candidates:[]
17        }; # chosen:(Some (candidates[chosen]+1)):candidates:[]
18    UPDATE; # candidates':[]
19    NIL operation; PAIR # (nil, candidates'):[]
20  }
```

(a)

```
{Elt "Agda" 0 ; Elt "Coq" 0 ; Elt "Isabelle" 0 ; Elt "K" 0}
```

(b)

Fig. 1: A simple voting contract a and an example of initial storage b

values of the stack, and pushes either -1, 0 or 1 depending on the comparison between the value. **GT** then pops this value and pushes `true` if the value is 1. If the threshold is indeed greater than the required amount, the first branch of the **IF** is executed and **FAIL** is called, interrupting the contract execution and cancelling the transaction.

If the value was `false`, the execution continues on line `9`, where we prepare the stack for the next action: **DUP** copies the top of the stack, we then manipulate the tail of the stack while preserving it's head using **DIP**: there, we take the right element of the `(chosen, candidates)` pair with **CDR**, and we duplicate it again. By closing the block guarded by **DIP** we recover the former stack's top, and the following line takes its left element with **CAR**, and duplicates it.

On line `12`, we use **DIP** to protect the top of the stack again. **GET** then pops `chosen` and `candidates` from the stack, and pushes an option containing the number of votes of the candidate, if it was found in the map. If it was not found, **ASSERT_SOME** makes the program fail. On line `15`, the number of votes is incremented by **ADD**, and packed into an option type by **SOME**.

We then leave the **DIP** block to regain access to value at the top of the stack (`chosen`). On line `18`, **UPDATE** pops the three values remaining on top of the stack, and pushes the `candidates` map updated with the incremented value for `chosen`. Finally, we push an empty list of operations with `NIL operation`, and pair the two elements on top of the stack to get the correct return type.

3 Mi-Cho-Coq : a Verification Framework in Coq for Michelson

Mi-Cho-Coq consists of an implementation of a Michelson interpreter in Coq as well as a weakest precondition calculus à la Dijkstra [13].

Michelson syntax and typing in Coq Michelson's type system, syntax and semantics, as described in the main documentation, are fully formalised in Mi-Cho-Coq.

The abstract syntax tree of a Michelson script is a term of an inductive type which carries the script type :

```
Inductive instruction : list type → list type → Set :=
| NOOP {A} : instruction A A
| FAILWITH {A B a} : instruction (a :: A) B
| SEQ {A B C} : instruction A B → instruction B C → instruction A C
| IF {A B} : instruction A B → instruction A B → instruction (bool :: A) B
| LOOP {A} : instruction A (bool :: A) → instruction (bool :: A) A ...
```

A Michelson code is usually a sequence of instructions (SEQ), which is one of the instruction constructors. It has type instruction stA stB : stA and stB are respectively the type of the input stack and of the output stack.

The stack type is a list of Michelson type constructions, defined in the type inductive:

```
Inductive comparable_type : Set :=
| nat | int | string | bytes | bool | mutez | address | key_hash | timestamp.

Inductive type : Set :=
| Comparable_type (a : comparable_type) | key | unit | signature | operation
| option (a : type) | list (a : type) | set (a : comparable_type)
| contract (a : type) | pair (a b : type) | or (a b : type) | lambda (a b : type)
| map (key : comparable_type) (val : type)
| big_map (key : comparable_type) (val : type).
```

A full contract, for a given storage type storage and parameter type params is an instruction of type

```
instruction ((pair params storage) :: nil ) ((pair (list operation) storage) :: nil ).
```

Thanks to the indexing of the instruction inductive by the input and output stack types, only well-typed Michelson instructions are representable in Mi-Cho-Coq. This is very similar to the implementation of Michelson in the Tezos node which uses a similar feature in OCaml: generalized algebraic datatypes.

To ease the transcription of Michelson contracts into Mi-Cho-Coq AST we use notations so that contracts in Mi-Cho-Coq look very similar to actual Michelson code. The main discrepancy between Michelson and Mi-Cho-Coq syntax being that due to parsing limitations, the Michelson semi-column instruction terminator has to be replaced by a double semi-column instructions separator.

The ad-hoc polymorphism of Michelson instructions is handled by adding an implicit argument to the corresponding instruction constructor in Mi-Cho-Coq. This argument is a structure that carries an element identifying the actual implementation of the instruction to be used. As the argument is *implicit and maximally inserted*, Coq type unifier tries to fill it with whatever value can fit with the known types surrounding it, ie the type of the input stack. Possible values are declared through the Coq's canonical structures mechanism, which is very similar to (Coq's or Haskell's) typeclasses.

Michelson interpreter in Coq Michelson semantics is formalized in Coq as an evaluator `eval` of type `forall {A B : list type}, instruction A B → nat → stack A → M (stack B)` where `M` is the error monad used to represent the explicit failure of the execution of a contract. The argument of type `nat` is called the *fuel* of the evaluator. It represents a bound on the depth of the execution of the contract and should not be confused with Michelson's cost model which is not yet formalised in Mi-Cho-Coq.

Some domain specific operations which are hard to define in Coq are axiomatized in the evaluator. These include cryptographic primitives, data serialisation, and instructions to query the context of the call to the smart contract (amount and sender of the transaction, current date, balance and address of the smart contract).

A framework for verifying smart contracts To ease the writing of correctness proofs in Mi-Cho-Coq, a weakest precondition calculus is defined as a function `eval_precond` of type `forall {fuel A B}, instruction A B → (stack B → Prop) → (stack A → Prop)` that is a Coq function taking as argument an instruction and a predicate over the possible output stacks of the instruction (the postcondition) and producing a predicate on the possible input stacks of the instruction (the precondition).

This function is proved correct with respect to the evaluator:

```
Lemma eval_precond_correct {A B} (i : instruction A B) fuel st psi :
  eval_precond fuel i psi st ↔
    match eval i fuel st with Failed _ _ ⇒ False | Return _ a ⇒ psi a end.
```

Note that the right-hand side formula is the result of the monad transformer of [6] which here yields a simple expression thanks to the absence of complex effects in Michelson.

A short example - the Vote contract We give below, as an example, a formal specification of the voting contract seen previously. We want the contract to take into account every vote sent in a transaction with an amount superior to 5tez. Moreover, we want to only take into account the votes toward an actual available choice (the contract should fail if the wrong name is sent as a parameter). Finally, the contract should not emit any operation.

In the following specification, the *precondition* is the condition that must be verified for the contract to succeed. The *postcondition* fully describes the new state of the storage at the end of the execution, as well as the potentially emitted operations. `amount` refers to the quantity of μtez sent by the caller for the transaction.

Precondition: $\texttt{amount} \geq 5000000 \land \text{chosen} \in \texttt{Keys}(\text{storage})$
Postconditions: $\text{returned\_operations} = [\,] \land$
$\forall\ c, c \in \texttt{Keys}(\text{storage}) \iff c \in \texttt{Keys}(\text{new\_storage}) \land$
$\text{new\_storage}[\text{chosen}] = \text{storage}[\text{chosen}] + 1 \land$
$\forall\ c \in \texttt{Keys}(\text{storage}), c \neq \text{chosen} \Rightarrow \text{new\_storage}[c] = \text{storage}[c]$

Despite looking simple, proving the correctness of the vote contract still needs a fair number of properties about the map data structure. In particular we need some lemmas about the relations between the `mem`, `get` and `update` functions, which we added to the Mi-Cho-Coq library to prove this contract.

Once these lemmas are available, the contract can easily be proved by studying the three different situations that can arise during the execution : the contract can fail (either because the sender has not sent enough tez or because they have not selected one of the possible candidates), or the execution can go smoothly.

4 A case study : the Multisig Contract

The *multisig* contract is a typical example of access-control smart contract. A multisig contract is used to share the ownership of an account between several owners. The owners are represented by their cryptographic public keys in the contract storage and a pre-defined *threshold* (a natural number between 1 and the number of owners) of them must agree for any action to be performed by the multisig contract.

Agreement of an owner is obtained by requiring a cryptographic signature of the action to be performed. To ensure that this signature cannot be replayed by an attacker to authenticate in another call to a multisig contract (the same contract or another one implementing the same authentication protocol), a nonce is appended to the operation before signing. This nonce consists of the address of the contract on the blockchain and a counter incremented at each call.

Michelson Implementation To be as generic as possible, the possible actions of our multisig contract are:

– produce a list of operations to be run atomically
– change the threshold and the list of owner public keys

The contract features two entrypoints named `default` and `main`. The `default` entrypoint takes no parameter (it has type `unit`) and lets unauthenticated users send funds to the multisig contract. The `main` entrypoint takes as parameters an action, a list of optional signatures, and a counter value. It checks the validity and the number of signatures and, in case of successful authentication, it executes the required action and increment the counter.

The Michelson script of the multisig contract is available at [10]. The code of the `default` entrypoint is trivial. The code for the `main` entrypoint can be divided in three parts: the header, the loop, and the tail.

The header packs together the required action and the nonce and checks that the counter given as parameter matches the one stored in the contract.

The loop iterates over the stored public keys and the optional signatures given in parameter. It counts and checks the validity of all the signatures.

Finally the contract tail checks that the number of provided signatures is at least as large as the threshold, it increments the stored counter, and it runs the required action (it either evaluates the anonymous function passed in the contract parameter and emits the resulting operations or modifies the contract storage to update the list of owner public keys and the threshold).

Specification and Correctness Proof Mi-Cho-Coq is a functional verification framework. It is well-suited to specify the relation between the input and output stacks of a contract such as multisig but it is currently not expressive enough to state properties about the lifetime of a smart contract nor the interaction between smart contracts. For this reason, we have not proved that the mutlisig contract is resistant to replay attacks. However, we fully characterise the behaviour of each call to the multisig contract using the following specification of the multisig contract where `env` is the evaluation environment containing among other data the address of the contract `self env` and the amount of the transaction `amount env`.

```
Definition multisig_spec (parameter : data parameter_ty) (stored_counter : N)
           (threshold : N) (keys : Datatypes.list (data key))
           (new_stored_counter : N) (new_threshold : N)
           (new_keys : Datatypes.list (data key))
           (returned_operations : Datatypes.list (data operation))
           (fuel : Datatypes.nat) :=
  let storage : data storage_ty := (stored_counter , (threshold , keys)) in
  match parameter with
  | inl tt ⇒
    new_stored_counter = stored_counter ∧ new_threshold = threshold ∧
    new_keys = keys ∧ returned_operations = nil
  | inr ((counter, action), sigs ) ⇒
    amount env = (0 Mutez) ∧ counter = stored_counter ∧
    length sigs = length keys ∧
    check_all_signatures sigs keys (fun k sig ⇒
         check_signature env k sig
           (pack env pack_ty (address_ env parameter_ty (self env),
                              (counter, action)))) ∧
```

```
(count_signatures  sigs  >= threshold)%N ∧
new_stored_counter  = (1 + stored_counter)%N ∧
match action with
| inl  lam ⇒
  match (eval lam fuel  (tt ,  tt )) with
  | Return _ (operations ,  tt ) ⇒
    new_threshold  = threshold  ∧ new_keys = keys ∧
    returned_operations  = operations
  | _ ⇒ False
  end
| inr  (nt,  nks) ⇒
  new_threshold  = nt ∧ new_keys = nks ∧ returned_operations  = nil
end end.
```

Using the Mi-Cho-Coq framework, we have proved the following theorem:

```
Lemma multisig_correct   (params : data parameter_ty)
      (stored_counter  new_stored_counter  threshold  new_threshold  :  N)
      (keys new_keys :  list  (data key))
      (returned_operations  :  list  (data operation )) (fuel  :  nat) :
  let storage  :  data storage_ty  := (stored_counter ,  (threshold ,  keys)) in
  let new_storage :  data storage_ty  :=
    (new_stored_counter,  (new_threshold,  new_keys)) in
  17 ∗ length  keys  + 14 ≤ fuel →
  eval  multisig  (23 + fuel) ((params, storage ),  tt )
    = Return _ ((returned_operations ,  new_storage),  tt ) ↔
  multisig_spec  params stored_counter  threshold  keys
    new_stored_counter  new_threshold  new_keys returned_operations  fuel .
```

The proof relies heavily on the correctness of the precondition calculus. The only non-trivial part of the proof is the signature checking loop. Indeed, for efficiency reasons, the multisig contract checks the equality of length between the optional signature list and the public key list only after checking the validity of the signature; an optional signature and a public key are consumed at each loop iteration and the list of remaining optional signatures after the loop exit is checked for emptiness afterward. For this reason, the specification of the loop has to allow for remaining unchecked signatures.

5 Related Work

Formal verification of smart contracts is a recent but active field. The K framework has been used to formalize [17] the semantics of both low-level and high-level smart contract languages for the Ethereum and Cardano blockchains. These formalizations have been used to verify common smart contracts such as Casper, Uniswap, and various implentation of the ERC20 and ERC777 standards.

Ethereum smart contracts, written in the Solidity high-level language, can also be certified using a translation to the F* dependently-typed language[8].

The Zen Protocol[5] directly uses F* as its smart contract language so that smart contracts of the Zen Protocol can be proved directly in F*. Moreover, runtime tracking of resources can be avoided since computation and storage costs are encoded in the dependent types.

The Scilla [21] language of the Zilliqa blockchain has been formalized in Coq. This language is higher-level (but also less featureful) than Michelson. Its formalization includes inter-contract interactions and contract's lifespan properties. This has been used to show safety properties of a crowdfounding smart contract.

6 Limits and Future Work

As we have seen, the Mi-Cho-Coq verification framework can be used to certify the functional correctness of non-trivial smart contracts of the Tezos blockchain such as the multisig. We are currently working on several improvements to extend the expressivity of the framework; Michelson's cost model and the semantics of inter-contract interactions are being formalised.

In order to prove security properties, such as the absence of signature replay in the case of the multisig contract, an adversarial model has to be defined. This task should be feasible in Coq but our current plan is to use specialized tools such as Easycrypt[7] and ProVerif[9].

No code is currently shared between Mi-Cho-Coq and the Michelson evaluator written in OCaml that is executed by the Tezos nodes. We would like to raise the level of confidence in the fact that both evaluators implement the same operational semantics. We could achieve this either by proposing to the Tezos stakeholders to amend the ecomomic protocol to replace the Michelson evaluator by a version extracted from Mi-Cho-Coq or by translating to Coq the OCaml code of the Michelson evaluator using a tool such as CoqOfOCaml [12] or CFML [11] and then prove the resulting Coq function equivalent to the Mi-Cho-Coq evaluator.

Last but not least, to ease the development of certified compilers from high-level languages to Michelson, we are working on the design of an intermediate compilation language called Albert that abstracts away the Michelson stack.

References

1. Michelson: the language of Smart Contracts in Tezos. http://tezos.gitlab.io/mainnet/whitedoc/michelson.html
2. Proof-of-stake in Tezos. https://tezos.gitlab.io//mainnet/whitedoc/proof_of_stake.html
3. Tezos code repository. https://gitlab.com/tezos/tezos
4. Voting in Tezos. https://tezos.gitlab.io//mainnet/whitedoc/voting.html
5. An introduction to the zen protocol. https://www.zenprotocol.com/files/zen_protocol_white_paper.pdf (2017)
6. Ahman, D., Hritcu, C., Martínez, G., Plotkin, G.D., Protzenko, J., Rastogi, A., Swamy, N.: Dijkstra monads for free. CoRR **abs/1608.06499** (2016), http://arxiv.org/abs/1608.06499

7. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.: Easycrypt: A tutorial. In: Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures. Lecture Notes in Computer Science, vol. 8604, pp. 146–166. Springer (2013). https://doi.org/10.1007/978-3-319-10082-1_6
8. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. pp. 91–96. PLAS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2993600.2993611
9. Blanchet, B.: Modeling and verifying security protocols with the applied pi calculus and proverif. Foundations and Trends in Privacy and Security **1**(1-2), 1–135 (2016). https://doi.org/10.1561/3300000004
10. Breitman, A.: Multisig contract in Michelson. `https://github.com/murbard/smart-contracts/blob/master/multisig/michelson/generic_multisig.tz`
11. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. pp. 418–430. ICFP '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2034773.2034828
12. Claret, G.: Program in Coq. Theses, Université Paris Diderot - Paris 7 (Sep 2018), `https://hal.inria.fr/tel-01890983`
13. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (Aug 1975). https://doi.org/10.1145/360933.360975
14. Filliâtre, J.C., Paskevich, A.: Why3 – Where Programs Meet Provers. In: ESOP'13 22nd European Symposium on Programming. LNCS, vol. 7792. Springer, Rome, Italy (Mar 2013), `https://hal.inria.fr/hal-00789533`
15. Goodman, L.M.: Tezos: A self-amending crypto-ledger. position paper. `https://tinyurl.com/tezospp` (2014)
16. Goodman, L.M.: Tezos: A self-amending crypto-ledger. white paper. `https://tinyurl.com/tezoswp` (2014)
17. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Moore, B., Zhang, Y., Park, D., Ştefănescu, A., Roşu, G.: Kevm: A complete semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium. pp. 204–217. IEEE (2018)
18. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system release 4.08: Documentation and user's manual. User manual, Inria (Jun 2019), `http://caml.inria.fr/pub/docs/manual-ocaml/`
19. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer-Verlag, Berlin, Heidelberg (2002)
20. Pessaux, F.: FoCaLiZe: Inside an F-IDE. In: Workshop F-IDE 2014. Proceedings F-IDE 2014, Grenoble, France (May 2014). https://doi.org/10.4204/EPTCS.149.7, `https://hal.archives-ouvertes.fr/hal-01203501`
21. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language. CoRR **abs/1801.00687** (2018), `http://arxiv.org/abs/1801.00687`
22. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoué, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: POPL. pp. 256–270. ACM (Jan 2016), `https://www.fstar-lang.org/papers/mumon/`
23. The Coq development team: The Coq Reference Manual, version 8.9 (Nov 2018), `http://coq.inria.fr/doc`
24. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: Hacl*: A verified modern cryptographic library. Cryptology ePrint Archive, Report 2017/536

Call me Back, I have a Type Invariant

M. Anthony Aiello[1], Johannes Kanig[1], Taro Kurita[2]

[1]AdaCore Inc, {aiello, kanig}adacore.com
[2]Sony Corporation, taro.kurita@sony.com

Abstract

Callbacks in Smart Contracts on blockchain-based distributed ledgers are a potential source of security vulnerabilities: callbacks may lead to reentrancy, which has been previously exploited to steal large sums of money. Unfortunately, analysis tools for Smart Contracts either fail to support callbacks or simply detect and disallow patterns of callbacks that may lead to reentrancy. As a result, many authors of Smart Contracts avoid callbacks altogether, and some Smart Contract programming languages, including Solidity, recommend using primitives that avoid callbacks. Nevertheless, reentrancy remains a threat, due to the utility of and frequent reliance on callbacks in Smart Contracts.

In this paper, we propose the use of *type invariants*, a feature of some languages supporting formal verification, to enable proof of correctness for Smart Contracts, including Smart Contracts that permit or rely on callbacks. Our result improves upon existing research because it neither forbids reentrancy nor relies on informal, meta-arguments to prove correctness of reentrant Smart Contracts. We demonstrate our approach using the SPARK programming language, which supports type invariants and moreover can be compiled to relevant blockchains.

Introduction

Smart contracts (Szabo, 1997) are protocols that are intended to facilitate, verify or enforce the negotiation or performance of a contract. Within a blockchain-based distributed ledger (Nakamoto, 2008), smart contracts are realized by autonomous agents that always execute specific functionality in response to defined events, such as the receipt of a message or transaction (Ethereum Foundation, 2019). Smart contracts make their behavior available through public interfaces, are typically small, and have no global state. However, their composition, especially through callbacks, leads to complex and difficult-to-predict behavior — behavior that may be malicious, as illustrated in the high-profile attack on the Ethereum DAO (Daian, 2016).

Callbacks arise when one or more of the parties interacting with a smart contract is another smart contract: a message or transaction sent to a public interface of the first smart contract, S1, may result in the sending of a message or transaction to a public interface of the second smart contract, S2, and so on. Callbacks are useful and quite common (Grossman, et al., 2017), but may also lead to reentrancy: a public interface of S1 may send a message to the public interface of S2, which may send a message back to S1 through the same public interface.

Reentrancy may lead to data-integrity violations. Data local to S1 may be in an inconsistent state when control flow is transferred to S2. S2 then calls S1. Upon reentry to S1, the data remains in an inconsistent state, likely violating the assumptions under which the smart contract was judged to be correct. The attack on the Ethereum DAO was essentially an attack based on reentrancy that exploited inconsistent data. The chain of callbacks leading to reentrancy may be arbitrarily long. Moreover, new smart contracts may be added to the blockchain at any time. Thus, in general, authors of smart contracts must assume that any potential callback may result in reentrancy. Current approaches that address this problem focus on identifying potential callbacks with static analysis (Kalra, Goel, Dhawan, & Sharma, 2018), use gas limits to restrict the computation of a callback and limit recursion depth (Ethereum Foundation, 2019), or rely on meta-arguments to prove correctness of smart contracts with external calls (Grossman, et al., 2017).

However, the threat to data integrity posed by reentrancy from callbacks or by recursion is not new and has received considerable attention by the formal verification community, although not in the context of smart contracts. In object-oriented languages, for example, callbacks are ubiquitous and play an important role in many design patterns (Gamma, Helm, Johnson, & Vlissides, 1994). Likewise in procedural languages, if the entirety of the code cannot be assumed to be available for analysis, any call to unknown code may result in recursion. The answer presented by the formal-verification community to address this issue is the use of invariants on critical state. Invariants ensure that, outside of specifically identified sections of code, data-integrity properties are always enforced.

In this paper, we apply invariants to critical data local to smart contracts and thus derive a means to prove the correctness of smart contracts, even in the presence of callbacks and potential reentrancy. We demonstrate our approach using the SPARK programming language and also show that our method can be applied in other languages that support formal verification.

Context

The Token contract

A typical smart contract is the *token contract*. It allows sharing money or other *tokens* accessed by several accounts. At its heart the token contract is a map from accounts (identified by addresses) to the amount of tokens they have access to:

```
type Balances_Type is array (Address) of Natural;
Balances : Balances_Type;
```

There are functions to transfer tokens between users, adding tokens to one's allowance, and getting tokens paid out.

In the simplest case, tokens are just ether. But note that the map above is only the contract's view of reality and of how many ether every user has access to. Bugs in the smart contract can cause a deviation between the reality and the token contract's vision of reality. For example, it is implicit in the above definition that the token contract (which is itself an account) holds at least as much money as the sum of the balances for all users. If there is a bug in the smart contract, this may not actually be the case.

Reentrancy, the problem

A possible version of the token contract's payout procedure looks like this:

```
Procedure Payout (Sender : Address; Amount : Natural) is
begin
   if Balance (Sender) >= Amount then
      Send (Sender, Amount);
      Balance (Sender) := Balance (Sender) - Amount;
    end if;
end Payout;
```

The `Sender` object is automatically filled out by the calling mechanism of the language and corresponds to the invoking entity. We assume here that the sender is also a smart contract, because that is the interesting case. The Send procedure is also a primitive of the smart contract infrastructure. In the solidity language, there are several ways to send ether, we assume here that Send sends the money in such a way that the fallback function of the receiving contract is triggered. We ignore discussions of gas limits here and assume that the fallback function has as much gas available as required.

This version of the program is vulnerable to a reentrancy attack. In detail, it works this way. The attacker creates another smart contract as follows:

1. The attacking contract first puts some small amount of money inside the token contract using a Deposit functionality (not shown here).
2. The attacking contract calls the `Payout` procedure of the token contract, setting the `Sender` object to the smart contract itself, using an `Amount` which is less or equal to the deposited value.
3. This causes the condition in the Payout procedure to succeed and the token contract to send money to the attacking contract;
4. Sending money triggers the fallback code of the attacking contract. The fallback code simply calls Payout again, with the same amount.
5. As Payout has not yet updated the internal state of the token contract, the if- condition is still true and the token contract sends money again.
6. This again triggers the fallback code of the attacking contract …

The recursion continues until the transaction chain runs out of gas or the token contract doesn't have any ether any more.

A fix for the token contract is to update the internal state before sending any money:

```
Procedure Payout (Sender : Address; Amount : Natural) is
begin
   if Balance (Sender) >= Amount then
      Balance (Sender) := Balance (Sender) - Amount;
      Send (Sender, Amount);
    end if;
end Payout;
```

Reentrancy can still occur, but now the attacker cannot circumvent the protecting condition, so he will be able only to withdraw his own balance.
Of course the Token contract is just one example of a contract that may be vulnerable to a reentrancy attack. Any contract that calls code of unknown other contracts is potentially vulnerable.

Existing Protections and Related Work

Reentrancy attacks in the Ethereum network are real and have cost a lot of money for the victims in the past. So measures have been taken to avoid the problem. To the best of our knowledge, these measures mostly consist of excluding reentrancy altogether (see also the section concerning related work). For example, the most commonly used functions for transferring ether now have a gas limit, so that the receiving contract can do only very few actions (in particular not call other contracts). However, as Grossman et al (Grossman, et al., 2017) show, callbacks are used in a large number of contracts and cannot be completely avoided.

Related Work

There is a vast body of work related to data invariants in object-oriented programming (Barnett, DeLine, Fähndrich, Leino, & Schulte, 2004; Leino & Müller, 2005), which lead to the formulation of class invariants in JML (Leavens, 2019). Our work is also based on this existing research and applies it to callbacks in smart contracts.

Existing formal methods for smart contracts that address this vulnerability simply forbid all reentrancy. This is usually achieved by only supporting transfers of ether that do not trigger the fallback code of the recipient (or specify a very small gas limit that does not allow much code execution), or excluding the problem from the analysis. This is the case for the ZEUS system of (Kalra, Goel, Dhawan, & Sharma, 2018), where a warning is signaled for any code that may contain reentrancy.

Bhargavan et al (Bhargavan, Delignat-Lavaud, Fournet, Gollamudi, & Gonthier, 2016) present an F*-based method to apply formal verification to smart contracts. They discuss reentrancy and show that their system detects reentrancy, but do not present any solution to the issue.

Grossman et al (Grossman, et al., 2017) present the theoretical notion of *effectively callback free* contracts, whose state changes, even in the presence of callbacks (or reentrancy) happen in an order that can also be achieved by a callback-free sequence of calls. Such callbacks are harmless wrt. the type of reentrancy attack discussed here. Their paper mainly concentrates on online detection of violations of this property, but they also have a section of formal verification of a contract using Dafny (Leino K. , 2010). However, Dafny does not support object or class invariants, so their Dafny-verification uses a meta-argument to remove some effects from calls that might contain callbacks. Our method can be seen as an in-language way to show that a contract is effectively callback free, which does not rely on meta-arguments.

Using SPARK type invariants to deal with Callbacks

Quick overview over SPARK

SPARK (McCormick & Chapin, 2015) is a subset of Ada (Barnes, 2012) and targets mainly embedded applications. It has strong support for formal verification.

Basic annotations for proof

SPARK has built-in support for formal verification. One basic feature is pre- and postconditions, as well as global annotations that can be attached to a procedure declaration:

```
procedure Add_In_Z (X, Y : Integer)
with Global     => (In_Out => Z),
     Post       => (Z = Z'Old + X + Y);
```

Extra information can be attached to a procedure using the `with` keyword. This is used to attach the information Global, which says that this procedure reads and writes the global variable Z. Also, we attached the information Post, which says that the new value of this variable Z is the sum of the old value of Z and the values of X and Y (we ignore concerns of arithmetic overflow in this example).

SPARK can formally verify that functions indeed respect the attached information such as Global and Post, similar to e.g. Dafny or Why3; in fact the formal verification engine in SPARK is based on Why3.

Private types

SPARK allows the user to separate a project into *packages*, each package having a package specification, visible by others, and an implementation which is private to this package. The package specification can contain so-called *private types,* or abstract types in other languages, where clients of the package cannot see the actual implementation of the type, only that the type exists:

```
type T is private;
…
type T is new Integer;
```

Type invariants

In SPARK, one can attach to type invariants to the implementation of a private type, for example as follows:

```
type T is private;
…
type T is record
   A : Integer;
   B : Integer;
End record
With Type_Invariant => T.A < T.B;
```

The idea is that type invariants must be maintained by the package. The package is allowed to *assume* the type invariant on input of any of its procedures or functions, and is allowed to temporarily break the invariant. However, it has to reestablish the invariant whenever an object leaves the scope of the package. This can be either by returning such an object to the caller, or by passing the object to a procedure or function that belongs to another package. The SPARK tool can prove that type invariants are correctly used and enforced by the package. SPARK type invariants have many restrictions; we mostly ignore these restrictions in the paper to keep a natural flow to the paper, but a dedicated section explains how we circumvented them to be able to actually use the SPARK tool.

Ghost code

Any declaration in SPARK (e.g. a type, object or procedure) can be annotated as *ghost.* This means that the declaration is only used for the purposes of verification, and does not contribute to the functionality of the code. This property is checked by the compiler and SPARK tools. A well-defined set of statements, such as assignments to ghost objects and calls to ghost procedures, are considered *ghost code* by the compiler and removed when compiling the program[1]. The following code example makes sure that the procedure `Do_Some_Work` is called after calling `Initialize` first.

```
Initialized : Boolean := False with Ghost;

procedure Initialize with
   Post => Initialized;

procedure Do_Some_Work with
  Pre => Initialized;

procedure Initialize is
begin
   ...  Do some initialization here ...
   Initialized := True;
end Initialize;
```

Adding annotations to the Payout procedure

The first step to apply formal verification would be to add pre- and postconditions to the payout procedure. Here is a first attempt:

```
Procedure Payout (Sender : Address; Amount : Natural)
with Post =>
   (for all Addr of Address =>
       if Addr = Sender and then Balance'Old (Addr) >= Amount
       then Balance (Addr) = Balance'Old (Addr) - Amount
```

[1] The compiler can also be configured to compile the application with ghost code enabled, which can be useful for dynamic checking of properties e.g. during unit testing.

```
              else Balance (Addr) = Balance’Old (Addr));
```

This postcondition summarizes the naive understanding of what `Payout` does: it sends a fixed amount of money to `Sender`, updating the `Balance` variable as well. As is common in systems that are based on deductive verification, one needs to specify also what remains unchanged. Here, Balance is only changed for the Sender, and only when the amount is actually sent (that is, actually available to be paid out).

To prove this postcondition, we also need to explain what Send is doing. In SPARK terms this means writing pre- and postconditions for the Send procedure, and Global annotations. Global annotations are *frame conditions*, a fundamental element of proof tools for imperative languages. They say what global state can potentially be modified by the procedure. It turns out Send can have quite a large effect, given that Send can execute completely unknown code. The Payout procedure might well attempt to send money to some smart contract that was added to the blockchain at a later stage. Also we have seen that via reentrancy, Send can even modify our own state. We don’t really care about the state of any other smart contract here, but we do care about the state of our own contract. So we have no choice but to annotate Send with this global annotation:

```
Procedure Send (Addr : Address; Amount : Natural)
with Global => (In_Out => Balance);
```

At this stage, we can’t really add any information to Send in the form of a postcondition on *how* it changes the state. After all, Send may call any procedure of the token contract, via any of the public procedures or functions of the contract.

However, now there is no chance that we can prove the postcondition of Payout, because Send can change our own state, and in an unknown way! Moreover, looking at the postcondition we wrote for Payout, it is wrong anyway, even for the corrected code. Via reentrancy, the sender can transfer more money than just Amount, though in the corrected version the attacker cannot exceed his balance. We need to go back to the basics and understand the difference between the original version of Payout and the corrected one.

Why the fix works

For the following, we now assume that Send contains two actions, that are executed in this order:

1. The actual sending of ether from the sender to the recipient;
2. The execution of the fallback code of the recipient.

The issue with the original version of Payout was that in the second step, the global state of the token contract was in an inconsistent state. The money was already sent, but the Balance map hadn’t been updated yet. The fallback code can be executed in this inconsistent state, and that’s why the if-condition in Payout becomes useless.

Now it is easy to see why the fix works: Now both the update to the token state as well as the ether transfer have been done when calling the fallback code. When the fallback code is executed, the state of the token contract is consistent. While the fallback code can still cause reentrancy, there should be no more “surprises”.

Grossman et al (Grossman, et al., 2017) call the corrected version *effectively call-back free*, because the state changes happen in such a way that they could also be achieved by a sequential series of calls to the interface of the object, without any reentrancy. This is not the case in the incorrect version, where the inconsistent sequence of state changes can only be achieved via reentrancy.

In SPARK, we can model the two steps of Send as follows. We introduce ghost state for sent tokens and wrap the Send procedure as follows:

```
Sent : Balances_Type with Ghost;

procedure Wrap_Send (Addr : Address; Amount : Natural) is
begin
  Sent (Addr) := Sent (Addr) + Amount;
  Send (Addr, Amount);
```

```
end Send;
```

This also requires to update the global effect of Send to include Sent. Here is the a summary of the changes:

```
procedure Send (Addr : Address; Amount : Natural)
With Global => (In_Out => (Balance, Sent));

procedure Wrap_Send … --  as above

procedure Payout (Sender : Address; Amount : Natural) is
begin
   if Balance (Sender) >= Amount then
      Balance (Sender) := Balance (Sender) - Amount;
      Wrap_Send (Sender, Amount);
    end if;
end Payout;
```

The solution in SPARK

So to prove the correctness of the corrected version, we need to:

1. Come up with a criterion for the data to be consistent;
2. Prove that the data is consistent whenever the control flow leaves the token contract, either via a regular return statement, or via a call to other code.

For (1), concentrating only on the Payout procedure, it is enough to say that for each address, the sum of the money sent and the balance should remain constant. A way of saying that it stays constant is to say that it is equal to some other quantity which stays unmodified during the whole computation. Let's represent this quantity by a new array K:

```
K : Balances_Type with Ghost;
```

For (2), luckily, the SPARK language already has a construct that does exactly that. It is called a type invariant, that is a property attached to a type, that should hold at certain points. Simply expressing the property of (1) as a type invariant and attaching it to the right type will do exactly what we need. We can express our invariant of the relevant data like this:

```
(for all A of Address => K (A) = Balance (A) + Sent (A))
```

That is all. We can now remove the postcondition of Payout[2]. The final version can be proved by SPARK in a few seconds; the incorrect version (by switching the two statements in the if-block) is correctly not proved, because the type invariant cannot be established before calling Send.

Some limitations of SPARK and their workarounds

As mentioned, we have described a solution which uses some features that SPARK doesn't actually support (but could). First, type invariants are attached to types, while we would like to attach them to objects, or maybe to the package itself. Then, type invariants can't mention global objects, while the invariant we showed mentions the three global objects Balance, Sent, and K.

[2] We can't really express anything useful in the postcondition here. Any public function of the Token contract might be called via reentrancy, updating the state in a consistent but unknown way.

We can work around these two annoyances simply by creating a record type which contains these three variables as fields. We attach the type invariant to this type:

```
Type Data_Type is private;
...
type Data_Type is record
   Balance : Balance_Type;
   Sent    : Balance_Type;
   K       : Balance_Type;
end record
with Type_Invariant =>
  (for all A of Address => K (A) = Balance (A) + Sent (A));
```

This works well. One further limitation is that we now cannot specify the Ghost status of Sent and K anymore, because currently Ghost status cannot be set for individual fields of a record. But the entire record cannot be ghost, because the Balance data is required to be present during execution.

The last limitation is that we cannot create global variables of a type which has a type invariant. So we need to add a parameter of type Data_Type to all relevant procedures, including the Send wand Wrap_Send procedures.

Reasoning in SPARK is strict on a per-procedure basis; this means that adding a wrapper such as Wrap_Send potentially increases the verification effort, as the wrapper would need annotations and separate proofs. However, local procedures with no annotations are automatically inlined by SPARK. So we deliberately do *not* add any pre- and postconditions to Wrap_Send.

Compilation of SPARK to Blockchain virtual machines

SPARK is a subset of Ada, so if we can compile Ada to a blockchain, we are good. A direct compiler from Ada to (say) EVM does not exist, but various indirect ways are possible. The easiest way is to use go from Ada to LLVM via the gnat-llvm (Charlet, 2018) tool. The LLVM intermediate representation can then be translated to Solidity using Solidify, a tool that can generate Solidity code from LLVM (Kothapalli, 2017). Finally, we can use the Solidity compiler to translate to EVM bytecode. (Ethereum Foundation, 2019)

Other languages that support reasoning about callbacks

SPARK is not the only tool to have both type invariants and ghost code. We give a non-exhaustive overview over other languages and tools that would also support this style of reasoning.

Why3 (Filliâtre & Paskevich, 2013) is a well-known research tool for formal verification. There is ongoing work to support compilation to the EVM bytecode (Nehai & Bobot, 2019). Also, Why3 has support for type invariants and ghost code, although the rules are a bit different from the ones in SPARK. One main difference is that Why3 has no notion package encapsulation of abstract types, that is, a type in Why3 is either abstract for everybody, or the definition of the type fully visible to everybody. So there is no notion of scope for a type invariant, and type invariants are checked at every function boundary. We suspect that this is a bit too restrictive for realistic contracts. Our example, if the Wrap_Send function is inlined, should work in the same way in Why3. Similar to SPARK, the type invariant has to be attached to a single type, so one has to introduce a record type that holds all relevant data.

The Java modeling language JML (Leavens, 2019) has support for class invariants and ghost code, so the code shown in this paper should be easy to translate to JML and should work there, too. In addition, the JML language allows to specify an effect called “everything”, which is a convenient way to to say that a call may write “any” visible object. In SPARK and Why3, the user has to manually deduce the relevant set of objects, and annotate Send correctly.

This style of verification using type or class invariants could be simulated in a language without type invariants (such as Dafny) by repeating the invariant as appropriate in pre- and postconditions and intermediate assertions. But this would require a meta-argument to show that the reasoning is correct; also it would be very error-prone. An intermediate assertion, for example, could be omitted by accident, and the tool would not be able to detect the error.

Conclusion

Research in deductive verification has already tackled the issue of callbacks and reentrancy, but to our knowledge this research had never been applied to smart contracts. We have shown that the language feature of type invariants enables deductive verification of smart contracts even in the presence of callbacks, including reentrancy. This result improves upon existing research, that either excludes callbacks, or requires a meta-argument to remove effects. Our paper has used SPARK to illustrate the running example, but Why3 and JML have similar language features and could also support this style of reasoning. Our conclusion is that a language for formal verification of smart contracts should have support for type, object or class invariants to efficiently deal with callbacks and reentrancy issues.

References

Ethereum Foundation. (2019, 06 24). *Solidity*. Retrieved from https://solidity.readthedocs.io/en/develop/

Barnes, J. (2012). *Ada 2012 Rationale.* Retrieved from https://www.adacore.com/papers/ada-2012-rationale/

Barnett, M., DeLine, R., Fähndrich, M., Leino, K., & Schulte, W. (2004, June). Verification of object-oriented programs with invariants. *Journal of Object Technology, 3*(6).

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., & Gonthier, G. (2016). Formal Verification of Smart Contracts. *ACM Workshop on Programming Languages and Analysis for Security.* Vienna, Austria.

Charlet, A. (2018, October 4). *AdaCore Techdays - GNAT Pro Update.* Retrieved from https://www.adacore.com/uploads/page_content/presentations/TechDaysParis2018-2-GNAT-Pro-Update-Tech-Days-2018-Paris.pptx

Daian, P. (2016). *Analysis of the DAO exploit*. Retrieved from http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

Filliâtre, J.-C., & Paskevich, A. (2013). Why3 - Where Programs Meet Provers. *Proceedings of the 22nd European Symposium on Programming* (pp. 125-128). Springer, Lecture Notes in Computer Science.

Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzki, N., Sagiv, M., & Zohar, Y. (2017). Online detection of effectively callback free objects with applications to smart contracts. *Principles of Programming Languages.* Paris, France: ACM SIGPLAN.

Kalra, S., Goel, S., Dhawan, M., & Sharma, S. (2018). ZEUS: Analyzing Safety of Smart Contracts. *Network and Distributed Systems Security (NDSS).* San Diego, USA.

Kothapalli, A. (2017). *Solidify, An LLVM pass to compile LLVM IR into Solidity.* Albuquerque, NM (United States): 005355MLTPL00. Sandia National Lab.(SNL-NM).

Leavens, G. T. (2019). *JML Reference Manual*. Retrieved from http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html

Leino, K. (2010). Dafny: An Automatic Program Verifier for Functional Correctness. *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (pp. 348-370). Dakar, Senegal: Springer.

Leino, K., & Müller, P. (2005). Modular Verification of Static Class Invariants. *FM 2005: Formal Methods International Symposium of Formal Methods Europe*, (pp. 26-42). Newcastle, UK.

McCormick, W. J., & Chapin, P. C. (2015). *Building High Integrity Applications with SPARK.* Cambridge University Press.

Nakamoto, S. (2008). *Bitcoin: A peer-to-peer electronic cash system.* Retrieved from https://bitcoin.org/bitcoin.pdf

Nehai, Z., & Bobot, F. (2019). *Deductive Proof of Ethereum Smart Contracts Using Why3.* CEA DILS. Retrieved from https://hal.archives-ouvertes.fr/hal-02108987

Szabo, N. (1997). Formalizing and securing relationships on public networks. *First Monday 2, 9*.

Wood, G. (2014). *Ethereum: A secure decentralised generalised transaction ledger.* Retrieved from https://gavwood.com/paper.pdf

A Distributed Blockchain Model of Selfish Mining

Dennis Eijkel and Ansgar Fehnker

d.j.eijkel@student.utwente.nl and ansgar.fehnker@utwente.nl

University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands

Abstract. Bitcoin is still the most widely used cryptocurrency. A big part of Bitcoin's appeal is that it maintains a distributed ledger for transactions known as the blockchain. Miners receive a fee for every block of transactions that they mine, and should expect a reward proportional to the computational power they provide to the network. Eyal and Sirer introduced *seflish mining*, a strategy timing the publication of blocks to give them a significant edge in profits. This paper models the behavior of honest and selfish mining pools in Uppaal, and analyses properties of the mining process in the presence of network delay. This shows what effects selfish mining would have on the share of profits, but also on the number of orphaned blocks in the blockchain. This analysis allows us to compare those results to known results from literature and to real world data. This analysis shows that it is essential to take into account that there does not exist a single view of the blockchain.

Keywords: Bitcoin, Bitcoin mining, Selfish mining, Uppaal

1 Introduction

Bitcoin [3,11] is at the time of writing the most used cryptocurrency [5] by market capitalisation. Miners in the Bitcoin network are incentivised by the reward that they receive for validating new blocks of transactions. The aim is that every miner receives its fair share of said reward for the computational effort they perform for the network.

The Bitcoin protocol does not specify when miners must publish their newly found blocks. The most basic strategy is to publish them immediately after the miner finds them. This is referred to as the *honest* strategy.

Eyal and Sirer introduced a strategy for publishing newly found blocks called *selfish mining* [8]. It forces honest miners to waste computational power by waiting strategically and responding to what other miners in the network find and publish. Eyal and Sirer provide in [8] pseudo-code for selfish mining, along with a mathematical model of the forking behaviour of the blockchain, and an additional model for the rewards. They compute the expected rewards in the steady state, i.e. in the long run, depending on the share of the selfish pool in the computational power, and the share of races the selfish pool will win in case there are competing forks. For this, they computed a threshold for which selfish mining will increase the profit of the miner. Below this threshold, selfish mining will actually incur a penalty for the selfish miner.

This paper presents an UPPAAL-SMC model for selfish mining. It models a blockchain network as a network of nodes, each with their own copy of the blockchain. The model includes stochastic network delays, which means that on average it will take a while before new blocks are adopted by the network. These aspects are absent from the Eyal and Sirer models. UPPAAL-SMC can then analyse the behaviour of the network and the evolution of the blockchain over the simulation time – one day – and compare this with historical data obtained from the real blockchain. In particular, how selfish mining affects the number of expected forks, and how this is distinguished from the frequency of forks in the presence of selfish miners.

Chaudary et al. used UPPAAL in [7] to model majority attacks. Their paper focuses on blockchain forking and included a detailed model of the blockchain. In [9] the same authors present a simplified version of the model presented in this paper to analyse a particular type of majority attack, intended to enforce a new Bitcoin standard. UPPAAL was also used by Andrychowicz et al. to verify the security of Bitcoin contracts, and to repair several issues in the protocol [6].

Sapirshtein et al. mathematically investigate bounds for which selfish mining is profitable and optimize the original strategy [13]. They show that selfish mining can be optimized, such that the threshold above which the strategy is profitable is lower than described in the original paper [8]. Heilman et al. used Monte-Carlo simulation to investigate eclipse attacks and proposed countermeasures that will reduce the chances of such attacks to succeed [10]. Neudecker presents a full-scale simulation model of Bitcoin to study partition attacks [12].

The next section will describe *selfish mining* and its pseudo-code implementation. Section 3 describes the UPPAAL-SMC model, and Section 4 the results of the analysis. Section 5 will conclude with a discussion of future work.

2 Selfish Mining

2.1 Bitcoin Mining Process

Bitcoin is a distributed and decentralized cryptocurrency [3,11] with a shared ledger of transactions which is stored in an append-only chain of *blocks* called the *blockchain*. A block contains a group of transactions, the hash of the preceding block, and a nonce. Since the block also includes the hash of the preceding block it defines a chain of blocks.

Nodes in the peer-to-peer Bitcoin network run a process, known as *mining*, to validate blocks of transactions, as well as to induce an order on transactions. Validation entails finding random nonce such that the hash value of the block falls below a certain threshold. Finding such a nonce can be considered to be

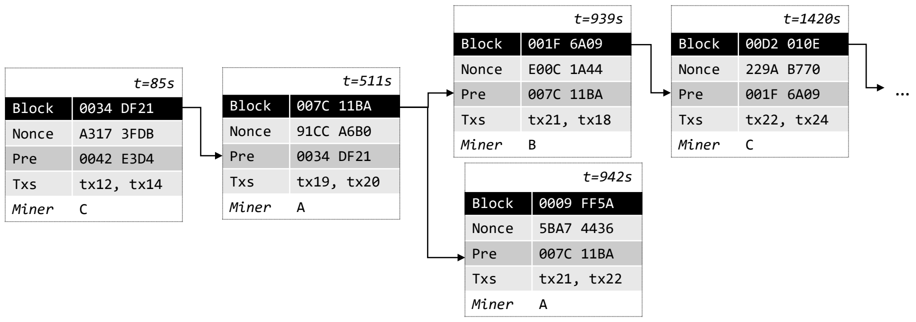


Fig. 1. Illustration of the Blockchain as hash-chain of blocks of transactions. For simplicity each block contains only two transactions.

a stochastic process with an exponential distribution, and is called the *proof-of-work challenge*. The threshold is regularly updated and agreed upon by the entire network such that a new block will be found on average every 10 minutes.

Figure 1 illustrates a blockchain. It starts with a block found by Miner C at `t=85s`, followed by a successor found by Miner A at `t=511s`. Due to the distributed nature of the network two pools may find a block at about the same time: in the example Miner B at `t=939s`, and Miner A at `t=942s`. If Miner A would have received the block of Miner B before it found its own, it would have abandoned its effort and switched to the `Block 001F 6A09`. The example assumes instead that Miner A found its own block first.

At this point, both blocks have been successfully mined as potential successors of `Block 007C 11BA`. Miners will continue with the block they receive first, and due to the distributed nature of the network, different pools may continue with mining different blocks, giving rise to so-called *forks*. It could take some time to resolve a fork and during that time, different views of the blockchain will exist. Blocks that fall outside of this longest chain are called *orphaned* blocks.

The race in Fig.1 is resolved as soon as the next block is found; here Block `001F 6A09`. Once this happens the protocol stipulates that the blocks in the longest chain become part of the authoritative blockchain. Only miners of blocks in the longest chain will receive the rewards attached to mining.

2.2 Selfish Mining Process

The Bitcoin protocol [3,11] does not specify when miners must publish their newly found blocks. The most basic strategy is to publish them immediately after they are mined. This is referred to as the *honest* strategy. Eyal and Sirer introduced a strategy for publishing newly found blocks called *selfish mining* [8].

Figure 2 illustrates one of the basic steps of selfish mining, intended to increase the number of forks. In this example, Miner C finds a block at `t=7s`. This block will be received by Miner A at `t=8s`, and by Miner B at `t=11s`[1]. All three

[1] Note, that in general, the network does not have access to a shared global time.

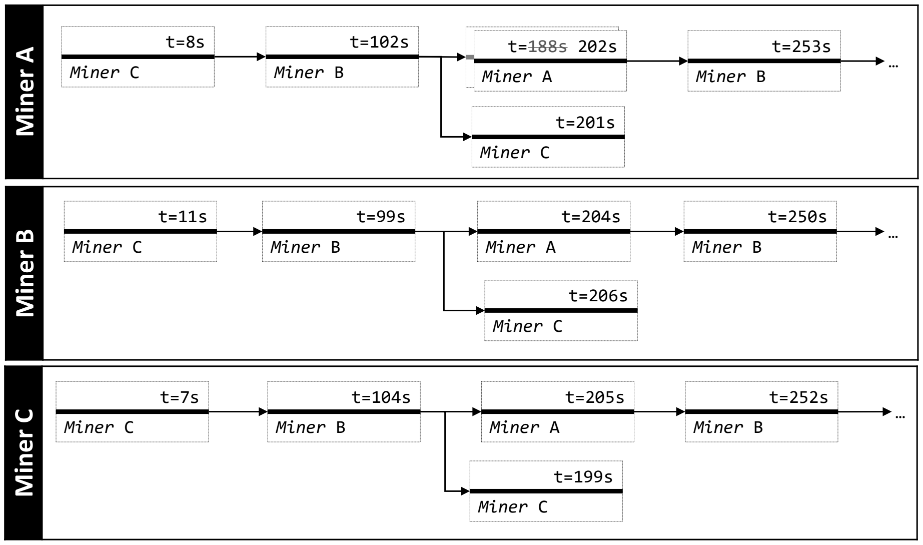


Fig. 2. Illustration of forks as races between different blockchains in a distributed network. Pool A is selfish miner, and postpones publication of a block found at `t=188s` until `t=202s`. This example omits for simplicity the hash values, nonces, and transactions.

miners will continue mining with this block. At `t=99s` Miner B finds a block and publishes it. It will be received by Miner A and C at `t=102s` and `t=104s`, respectively. Again all three miners will continue with this block. Up to this point, all miners employ honest mining.

Assume that Miner A employs the selfish strategy. If it finds a block at `t=188s`, it will not publish it immediately, but wait. If it receives a block by one of the other miners – in the example a block of Miner C at `t=201s` – it will publish its own block immediately, which intentionally creates a fork. The gamble is that its own block arrives at the others miners before the block of Miner C. In the example Miner B receives the block of Miner A before the block of Miner C, and thus continues mining the block of Miner A. If Miner B then finds a new block at time `t=250s` it will orphan the block Miner C found previously. Miner C's computational power from `t=104s` until `t=252s` – when it received the block of Miner B – was effectively wasted.

The question is if this can actually be beneficial for the selfish miner. In the example, Miner A forwent a certain reward for the block it found at `t=188s` to enter a race with pool C at `t=202s`. This looks superficially like a disadvantageous strategy. However, Figure 2 describes only one step of selfish mining, namely the step that intentionally introduces forks.

The following gives a full list of steps for the selfish miners. It assumes that the selfish miner always mines at the end of its own private chain.

1. **The selfish miner finds a block.**
    (a) **There is a fork, and both branches have length 1.** In this case, the selfish miner found a block to decide the race in its favour. The selfish miner appends the block to its private chain and publishes it. The selfish miner intends to orphan the single block in the public branch, and secure the rewards of its own branch.
    (b) **Otherwise.** The new block will be appended to its private chain, without publishing it. This includes cases where the private chain is two or more blocks ahead of the public chain.
2. **The selfish miner receives a block.** Provided that it actually increases the height of the public chain, the selfish miner will proceed as follows:
    (a) **If there is no fork.** This means the public and private chains are identical. The received block is appended to the public chain, and the public chain is adopted as the private chain. The other miner will receive the rewards.
    (b) **There is one unpublished block in the private branch.** The received block is appended to the public chain. The unpublished block is published. This is the scenario depicted in Figure 5.
    (c) **There are two unpublished blocks in the private branch.** The private chain is published. Since the public chain should still be one block behind, this would secure all rewards in the private branch for the selfish miner. After this, there is no fork.
    (d) **Otherwise.** This is the case when the selfish miner is more than two blocks in the lead. The selfish miner will publish the first unpublished block. While the private chain is at least two blocks ahead, the public branch and the portion of the private branch that has been published have the same height. To other miners, a race is ongoing, even though the selfish miner already has the blocks to decide the race in its favour.

To implement this strategy the selfish miner needs to maintain a record of the head of the public chain, of the head of the private chain, the head of the portion of the private chain that has been published, and the block where the private and public chain fork. It should be noted that the *public* chain is the local view that the selfish miner has of the blockchain. As discussed previously, in general, different miners may have different views.

Eyal and Sirer have shown that a miner using selfish mining will gain more rewards than would be proportional to their computational power, under the assumption that the other miners use the honest strategy. This result depends on the share α of computational power the selfish miner has in the network and the fraction γ of miners that adopt the block of the selfish miner in case of a fork. They discovered that selfish mining gives an increased reward if $(1-\gamma)/(3-2\gamma) < \alpha$. This means, for example, that if a quarter of the other nodes adopt the block of the selfish miner, i.e. $\gamma = 0.25$, then the selfish mining strategy will pay off if the network share satisfies $\alpha > 0.3$.

3 Uppaal Model

The Uppaal-SMC model consists of three templates: one for modelling the behaviour of an honest miner, one for a selfish miner, and one for modelling the propagation delay between miners. A fourth template is added to observe the blockchain, but this node does not take part in the protocol. This section will describe the important global variables and templates in detail.

Global variables and constants. The model includes two arrays of broadcast channels, `sendBlock[POOLS]` and `recvBlock[POOLS]`, for miners to send and receive blocks, where `POOLS` is the number of miners. A block is defined as a struct of the `height`, a bounded integer `BlockIndex`, and array `rewards[POOLS]`. If a miner with ID `id` mines a new block, it increments `height` and `rewards[id]`.

Global variable `syncBlock` is used as an auxiliary to copy blocks between processes. Important constants are integer `PDELAY` for the expected network delay, and integer array `POOL_RATES[POOLS]`, which contains for each miner the rate at which it finds blocks. The model uses as basic time unit 1 second; a rate of 1200 means that a miner finds on average one block every 1200 seconds.

Network links. The network link between any two miners is modelled as a one-place buffer with delay. For any pair of IDs `in` and `out`, the model will include one instance of the link template, depicted in Fig. 3. From the initial state it will synchronize on channel `sendBlock[in]` with Pool `in`, and copy the received block in global variable `syncBlock` to its local variable `blockBuffer`. It then enters the location to the right. In this location it will synchronize on channel `recvBlock[out]` with Miner `out` at a rate of 1 in `PDELAY` seconds. This transition will copy the value of the buffer to `syncBlock`. If it receives another block from Miner `in`, it will store that block in the buffer. Note, that the model will include for any pair of miners one link, i.e. for a network with 10 miners, 100 links, each with its own buffer.

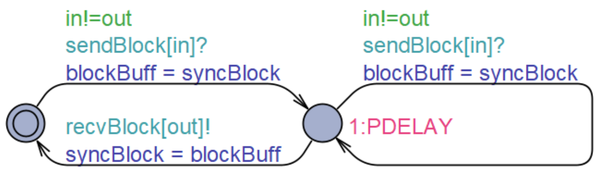


Fig. 3. Parameters of the link template are the `ID` of sender `in` and receiver `out`.

Honest mining. Figure 4 shows the template for an honest miner with ID `id`. It has a single location with two transitions. The first models successfully mining a block. It calls method `outputBlock` which increments the height of the head of its private chain and the rewards for itself. The other transition models receiving a block which calls method `updateBlock` which will adopt the new block if it improves on the height of the head of its private chain.

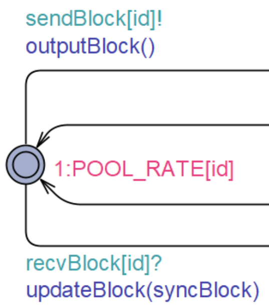


Fig. 4. The honest mining template has as parameter the `id` of the miner.

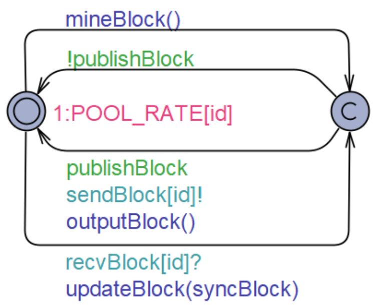


Fig. 5. The selfish mining template has as parameter the `id` of the miner.

Selfish mining. The selfish miner keeps a record of four blocks: the head of the private chain `privateBlock`, the head of the public chain `publicBlock`, the most recently published block `publishedBlock`, and the block where the public and the private chain fork, `forkBlock`. In addition, it uses a local Boolean `publishBlock` which encodes whether a block should be published.

The top-most edge models mining a block (case 1 on Page 5). It calls method `mineBlock()` at a rate of 1 in `POOL_RATE[id]` seconds. It decide whether to publish (part of) its private chain if it mines or receives a block.

The bottom-most edge models receiving a block (case 2 on Page 5). It synchronizes on channel `recvBlock[id]`, and calls `updateBlock` which decides whether to append it to the private chain, or whether to publish a part of the private chain. It sets Boolean `publishBlock`, depending on whether a block should be published, or not.

The committed location in the mining template in Fig. 5 completes the process. If `mineBlock()` or `updateBlock` set `publishBlock` to false, the selfish miner returns silently to the initial location. If it was set to true method `outputBlock` will copy the block that is meant to be published to `syncBlock`, but also to `publishedBlock` and `publicBlock`. The code for methods `mineBlock()` and `updateBlock` is given in Listing 1.

System composition. The analysis in Section 4 uses a model with 10 miners and 100 links. It considers the 6 sets of network shares, as given in Table 1. If the model includes a selfish miner it would be Miner A. Miner B has a share of 20% in all experiments to make the results comparable. A share of 20% would correspond to finding a block once every 3000 seconds, assuming a network rate of one block every 600 seconds. These rates are simplified but still largely similar to the distribution of hash rates in the real world [4].

Uppaal-SMC simulated each scenario 1000 times for one day of simulation time, i.e. for 86400 seconds. The simulation of one single scenario takes about 80 seconds on an Intel Core i5-5200 with 2 cores at 2.2GHz.

```
void mineBlock() {//case 1
    if (privateBlock.height == publicBlock.height &&
        privateBlock.height-forkBlock.height == 1) {//case 1.(a)
        privateBlock.height++;
        privateBlock.rewards[id]++;
        outputBuffer   = privateBlock;
        forkBlock      = privateBlock;
        publishBlock   = true;
    }
    else{                                          //case 1.(b)
        privateBlock.height++;
        privateBlock.rewards[id]++;
        publishBlock   = false;
    }
}

void updateBlock(Block newBlock) {                 //case 2
    if (newBlock.height>publicBlock.height) {
        if(newBlock.height>privateBlock.height){    //case 2.(a)
            privateBlock   = newBlock;
            forkBlock      = newBlock;
            publishedBlock = newBlock;
            publicBlock    = newBlock;
            publishBlock   = false;
        }
        else
        if(newBlock.height == privateBlock.height) {//case 2.(b)
            outputBuffer   = privateBlock;
            publishBlock   = true;
        }else                                      //case 2.(c)
        if(newBlock.height == privateBlock.height-1) {
            outputBuffer   = privateBlock;
            forkBlock      = privateBlock;
            publishBlock   = true;
        }
        else {                                     //case 2.(d)
            publishedBlock.height++;
            publishedBlock.rewards[id]++;
            outputBuffer   = publishedBlock;
            publishBlock   = true;
        }
    }
}
```

Listing 1. Essential methods of the selfish miner.

Scenario	A	B	C	D	E	F	G	H	I	J
#1	1%	20%	20%	15%	15%	10%	10%	5%	2%	2%
#2	10%	20%	20%	15%	15%	10%	5%	2%	2%	1%
#3	20%	20%	15%	15%	10%	10%	5%	2%	2%	1%
#4	30%	20%	15%	10%	10%	5%	5%	2%	2%	1%
#5	40%	20%	10%	10%	5%	5%	5%	2%	2%	1%
#6	50%	20%	10%	5%	5%	2%	2%	2%	2%	2%

Table 1. Network shares for different scenarios.

4 Analysis Results

This section will present for a 24h period the expected mining rewards and the expected number of orphaned blocks. The former allows a comparison with results by Eyal and Sirer, the latter with data obtained from the publicly available Bitcoin blockchain.

4.1 Mining Rewards

Fig. 6 depicts the height and rewards in different views of the blockchain. First is the number of *blocks mined* over the 24 hours period. It is around 144 blocks, as expected for a network that finds on average one block every 10 minutes.

Not all of these blocks will become part of the longest chain. Fig. 6 gives the blockchain height and the reward of the selfish and first honest miner, reward selfish and reward honest, respectively. Each of these three come in two versions depending on whether it is part of the private chain of the selfish miner, or the chain as known by the network.

These results show that as the network share of the selfish miner increases, it decreases the height of the blockchain, and increases the rewards for the selfish miner. For a miner with a 50% share the height is 89.4 and the reward 68.9, in the private blockchain of the selfish miner. In the blockchain of the first honest miner – Miner B in Table 1, who has a network share of 20% – the height is only 81.7, and the reward of the selfish miner is only 57.7. The difference is partly due to network delay, but mostly because the selfish miner has a buffer of 7.6 unpublished blocks in its private chain.

Fig. 7 translates these numbers to shares in the rewards. It also includes the nominal share these miners should achieve; the selfish miner proportionally to its network share, and the honest miner 20%. The results show that selfish mining becomes profitable once the network share of the selfish miner exceeds 30%.

To compare these to Eyal and Sirer's result, we need the probability that other miners adopt the block of the selfish miner above a competing block. It depends on two steps succeeding from the moment that the competing block is found. First the selfish miner has to receive the competing block before the

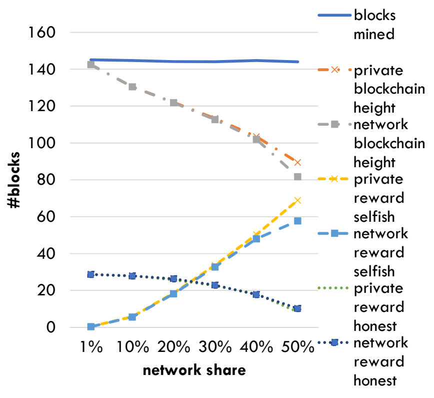


Fig. 6. Height and rewards of selfish and honest miner after 24 hours.

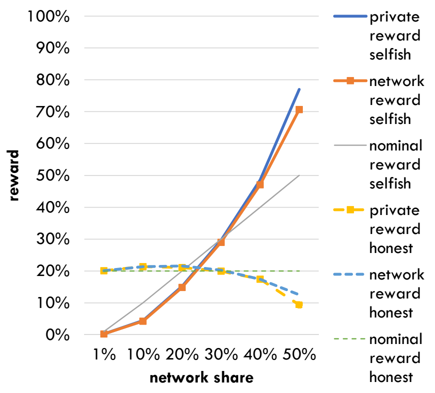


Fig. 7. Share of rewards for selfish and honest miner after 24 hours.

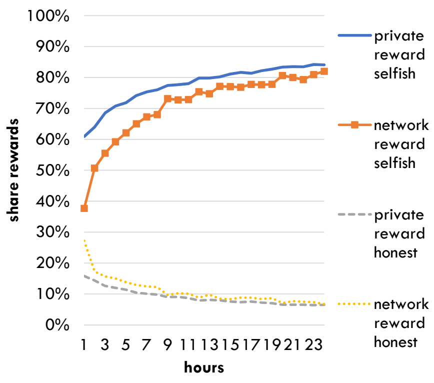


Fig. 8. Share of rewards per hour for the honest and selfish miner over 24 hours.

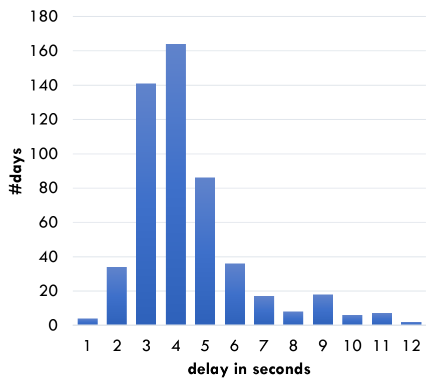


Fig. 9. Histogram of the propagation delays in the selected data set.

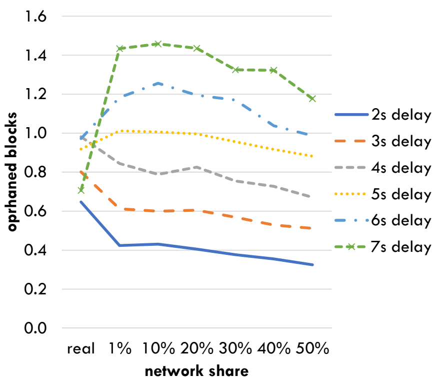


Fig. 10. Number of orphaned blocks in network w/o selfish miner after 24 hours.

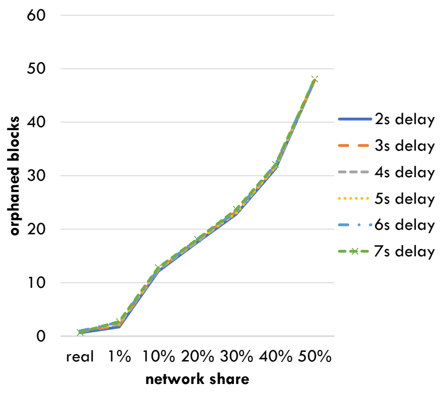


Fig. 11. Number of orphaned blocks in network with selfish miner after 24 hours.

other miners, (2) the block sent by the selfish miner in response has to arrive before the competing block. Given that in this model all delays use the same memoryless distribution both steps succeed with a 50% chance, giving an overall chance of 25%. Eyal and Sirer predict a threshold of 30% for this case, while in Fig. 7 the share of the 30% selfish miner is 29.6%.

Fig. 8 shows how this evolves over a 24 hour period for a selfish miner with a 50% network share. Initially, the selfish miner will appear to have a share that is below its 50% network share, as it is secretly mining blocks. As the day progresses its share will quickly exceed 80%, once it starts publishing blocks from its private chain.

All results of this section are based on a propagation delay of 4 seconds. For the results in this subsection, the propagation delay has little to no influence. The next subsection will discuss different propagation delays in more detail.

4.2 Orphaned Blocks

An essential aspect of selfish mining is to create forks such that other miners waste computational resources on blocks that are bound to be orphaned. To compare the models with data from the actually Bitcoin blockchain, we combined the data on orphaned blocks [2] with data on propagation times [1]. This gave 528 usable data points in the period from 18 March 2014 to 22 March 2017, i.e. days with both data on orphaned blocks and propagation times. Fig. 9 shows the distribution of days over different propagation times, rounded to the nearest integer second. This leaves us with a reasonable data set for propagation delays in the interval from 2 to 7 seconds.

Fig.10 shows the number of expected orphans if we have a network without any selfish miner. The figure includes, for reference, the number of orphans from the real data set, labelled *real*. The results show that as the delay increases, the number of orphans increases as well. With the exception of the data point for 7 seconds, the real data falls into the range given by the simulation.

This picture changes once we introduce a selfish miner as depicted in Fig. 11. Even a selfish miner with only a 1% network share leads to more orphans than for any scenario with only honest miners or the real data. This comparison suggests that there is no evidence in the real data of a prolonged presence of a selfish miner with a significant network share.

5 Discussion and Conclusion

In [8], Eyal and Sirer provide a pseudocode algorithm for selfish mining. The analysis uses a separate state transition model that captures the presence and length of a fork. Based on this model they manually derived state probabilities and expected rewards for each state. To validate the overall reward they use Monte Carlo Simulation. Their combination of models assumes a single view of the public chain where blocks are propagated instantly to provide estimates of the rewards a selfish miner can expect in the long run.

This paper presented a single unified modelling artefact. It also includes propagation delays, a block model with rewards, and a distributed blockchain. It does not separate the pseudo code from the transition probabilities, rewards, and the analysis of the evolution of the network over time. This allowed an automated analysis from the perspective of different participants, and compare these to the theoretical results by Eyal and Sirer, as well as to real-world data.

The analysis confirms that selfish mining becomes profitable for networks shares above 30%. The results of this paper show that the presence of a selfish miner may go undetected for the first few hours, but would be obvious after that. Future work would need to investigate how to identify a short-term attack on a blockchain. For this type of analysis, it is especially important to distinguish between the different views of the blockchain of different participants, as it is done in this paper.

All Uppaal-SMC models, simulation data and more detailed results will be available on `https://wwwhome.ewi.utwente.nl/~fehnkera/Q19`.

References

1. Bitcoinstats, network propagation times. `http://bitcoinstats.com/network/propagation/`. Accessed: 06-07-2019.
2. Bitcoin.info, number of orphaned blocks. `https://blockchain.info/charts/n-orphaned-blocks?timespan=all`, 2008. Accessed: 06-07-2019.
3. Bitcoin protocol rules. `https://en.bitcoin.it/wiki/Protocol_rules`, 2019. Accessed: 06-07-2019.
4. Blockchain.info, hashrate distribution. `https://blockchain.info/pools`, 2019. Accessed: 06-07-2019.
5. Coinmarketcap: Cryptocurrency market capitalizations. `https://coinmarketcap.com/`, 2019. Accessed: 06-07-2019.
6. Andrychowicz, Marcinand Dziembowski, S. M. D. M. Ł. Modeling bitcoin contracts by timed automata. In *Formal Modeling and Analysis of Timed Systems* (2014), M. Legay, Axeland Bozga, Ed., Springer.
7. Chaudhary, K., Fehnker, A., van de Pol, J., and Stoelinga, M. Modeling and verification of the bitcoin protocol. In *MARS 2015.* (2015), EPTCS.
8. Eyal, I., and Sirer, E. Majority is not enough: Bitcoin mining is vulnerable. In *FC 2014* (2014), LNCS 8437, pp. 436–454.
9. Fehnker, A., and Chaudhary, K. Twenty percent and a few days - optimising a bitcoin majority attack. In *NFM 2018* (2018), LNCS 10811, Springer.
10. Heilman, E., Kendler, A., Zohar, A., and Goldberg, S. Eclipse attacks on bitcoin's peer-to-peer network. In *SEC'15* (2015), USENIX Association.
11. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, 2008.
12. Neudecker, T., Andelfinger, P., and Hartenstein, H. A simulation model for analysis of attacks on the bitcoin peer-to-peer network. In *INM 2015* (2015).
13. Sapirshtein, A., Sompolinsky, Y., and Zohar, A. Optimal selfish mining strategies in bitcoin. In *FC 2016.* (2017), LNCS 9603, pp. 515–532.

Towards a Verified Model of the Algorand Consensus Protocol in Coq

Musab A. Alturki[1], Jing Chen[2], Victor Luchangco[2], Brandon Moore[1], Karl Palmskog[3], Lucas Peña[4], and Grigore Roşu[4]

[1] Runtime Verification, Inc., Urbana, IL, USA
{musab.alturki,brandon.moore}@runtimeverification.com
[2] Algorand, Inc., Boston, MA, USA
{jing,victor}@algorand.com
[3] The University of Texas at Austin, TX, USA
palmskog@acm.org
[4] University of Illinois at Urbana Champaign, IL, USA
{lpena7,grosu}@illinois.edu

Abstract. The Algorand blockchain is a secure and decentralized public ledger based on pure proof of stake rather than proof of work. At its core it is a novel consensus protocol with exactly one block certified in each round: that is, the protocol guarantees that the blockchain does not fork. In this paper, we report on our effort to model and formally verify the Algorand consensus protocol in the Coq proof assistant. Similar to previous consensus protocol verification efforts, we model the protocol as a state transition system and reason over reachable global states. However, in contrast to previous work, our model explicitly incorporates timing issues (e.g., timeouts and network delays) and adversarial actions, reflecting a more realistic environment faced by a public blockchain. Thus far, we have proved *asynchronous safety* of the protocol: two different blocks cannot be certified in the same round, even when the adversary has complete control of message delivery in the network. We believe that our model is sufficiently general and other relevant properties of the protocol such as liveness can be proved for the same model.

Keywords: Algorand · Byzantine consensus · blockchain · Coq.

1 Introduction

The Algorand blockchain is a scalable and permissionless public ledger for secure and decentralized digital currencies and transactions. To determine the next block, it uses a novel consensus protocol [1,3] based on pure proof of stake. In contrast to Bitcoin [6] and other blockchains based on proof of work, where safety is achieved by making it computationally expensive to add blocks, Algorand's consensus protocol is highly efficient and does not require solving cryptographic puzzles. Instead, it uses *cryptographic self-selection*, which allows each user to individually determine whether it is selected into the committees responsible for generating the next block. The self-selection is done randomly and independently by every participant, with probability proportional to its stake. Private communication channels are not needed, and the committees propagate their messages in public. They reach Byzantine consensus on the next block and certify it, so that all users learn what the next block is without any ambiguity. That is, rather than waiting for a long time so as to be sure that a block will not disappear from the longest chain like in Bitcoin, the Algorand blockchain does not fork, a certified block is immediately final and transactions contained in it can be relied upon right away. The Algorand blockchain guarantees fast generation of blocks as long as the underlying propagation network is not partitioned (i.e., as long as messages are delivered in a timely fashion). The Algorand consensus protocol, its core technology, and mathematical proofs of its safety and liveness properties are described in [3,1,2].

The focus of this work is to formally model and verify the Algorand consensus protocol using the Coq proof assistant. Automated formal verification of desired properties adds another level of assurance about its correctness, and developing a precise model to capture the protocol's runtime environment and the assumptions it depends on is interesting from a formal-methods perspective as well. For example, [11] proves state machine safety and linearizability for the Raft consensus protocol in a non-Byzantine setting, and [7] focuses on safety properties of blockchains and, using a largest-chain-based fork-choice rule and a clique network topology, proves eventual consistency for an abstract parameterized protocol. Similar to existing efforts, in this work we define a transition system relation on global protocol states and reason inductively over *traces* of states reachable via the relation from some initial state. As in previous efforts, we abstract away details on cryptographic primitives, modeling them as functions with the desired properties.

However, our goal and various aspects of the Algorand protocol presented new challenges. First, our goal is to verify the protocol's asynchronous safety under Byzantine faults. Thus, we explicitly allow arbitrary adversarial actions, such as corruption of users and replay of messages. Also, rather than relying on a particular network topology, we explicitly model global time progression and message delivery deadlines in the underlying propagation network. In particular, the Algorand protocol assumes that messages are delivered within given deadlines when the network is not partitioned, and that messages may be arbitrarily delayed and their delivery is fully controlled by the adversary when the network is partitioned. We have captured these aspects in our model. Moreover, as mentioned above, the Algorand protocol uses cryptographic self-selection to randomly select committees responsible for generating blocks. As mechanizing probabilistic analysis is still an open field in formal verification, instead of trying to mechanize randomized committee selection, we identify properties of the committees that are used to verify the correctness of the protocol without reference to the protocol itself. We then express these properties as axioms in our formal model. Pen-and-paper proofs that these properties hold (with overwhelming probability) can be found in [3,1].

It is worth pointing out that our approach is based on reasoning about *global* states and allows an adversary to arbitrarily coordinate actions among corrupted users. This is different from [8], which formally verifies the PBFT protocol under arbitrary local actions. Finally, [10] uses distributed separation logic for consensus protocol verification in Coq with non-Byzantine failures. Using this approach to verify protocols under Byzantine faults is an interesting avenue of future work.

Thus far, we have proved in Coq the *asynchronous safety* property of the protocol: namely, two different blocks can never be both certified in the same round, even when the adversary has complete control of message delivery in the network. We believe that our model is sufficiently general and other relevant properties of the protocol such as liveness can be proved for the same model.

2 The Algorand Consensus Protocol

In this section, we give a brief overview of the Algorand consensus protocol with details salient to our formal model. More details can be found in [3,5,1].

All users participating in the protocol have unique identifiers (public keys). The protocol proceeds in *rounds* and each user learns a *certified* block for each round. Rounds are asynchronous: each user individually starts a new round whenever it learns a certified block for its current round.

Each round consists of one or more *periods*, which are different attempts to generate a certified block. Each period consists of several *steps*, in which users propose blocks and then vote to certify a proposal. Specifically, each user waits a fixed amount of time (determined by network parameters) to receive proposals, and then votes to support the proposal with the best *credential* as described below; these votes are called *soft-votes*. If it receives a quorum of soft-votes, it then votes to certify the block; these votes are called *cert-votes*. A user considers a block certified if it receives a quorum of cert-votes. If a user doesn't receive a quorum of cert-votes within a certain amount of time, it votes to begin a new period; these votes are called *next-votes*. A next-vote may be for a proposal, if the user received a quorum of soft-votes for it, or it may be *open*. A user begins a new period when it receives a quorum of next-votes from the same step for the same proposal or all being open; and repeats the next-vote logic otherwise.[5]

Committees. For scalability, not all users send their messages in every step. Instead, a committee is randomly selected for each step via a technique called *cryptographic self-selection*: each user independently determines whether it is in the committee using a verifiable random function (VRF). Only users in the committee send messages for that step, along with a *credential* generated by the VRF to prove they are selected. Credentials are totally ordered, and the ones accompanying the proposals are used to determine which proposal to support.

[5] The actual logic for next-votes is more complex, but roughly speaking the next-votes are classified as either for proposals or open.

Network. Users communicate by propagating messages over the network. Message delivery is asynchronous and may be out-of-order, but with upper bounds on delivery times. However, messages may not be delivered within these bounds if the network is *partitioned*.

Adversary. The adversary can corrupt any user and control and coordinate corrupted users' actions: for example, to resend old messages, send any message for future steps of the adversary's choice, and decide when and to whom the messages are sent by them. The adversary also controls when messages are delivered between honest users within the bounds described above, and fully controls message delivery when the network is partitioned. The adversary cannot, however, control more than 1/3 of the total stake participating in the consensus protocol.

3 Model

Our model of the protocol in the Coq proof assistant is in the form of a transition system, encoded as an inductive binary relation on global states. The transition relation is parameterized on finite types of user identifiers (`UserId`) and values (`Value`); the latter abstractly represents blocks and block hashes.

User and Global State. We represent both the user state and global state as Coq records. For brevity, we omit a few components of the user state in this paper and only show some key ones, such as the Boolean indicating whether a user is corrupt, the local time, round, period, step, and blocks and cert-votes that have been observed. The global state has the global time, user states and messages via finite maps [4], and a Boolean indicating whether the network is partitioned.

```
Record UState := mkUState {
 corrupt: bool; timer: R;
 round: ℕ; period: ℕ; step: ℕ;
 blocks: ℕ → seq Value;
 certvotes: ℕ → ℕ → seq Vote;
 (* ... omitted ... *)
}.
```

```
Record GState := mkGState {
 network_partition: bool;
 now: R;
 users: {fmap UserId → UState};
 msgs: {fmap UserId → {mset R * Msg}};
 msg_history: {mset Msg};
}.
```

State Transition System. The transition relation on global states `g` and `g'`, written `g ⇝ g'`, is defined in the usual way via inductive rules. For example, the rule for adversary message replay is as follows:

```
step_replay_msg : ∀ (pre:GState) uid (ustate_key : uid ∈ pre.(users)) msg,
  ¬ pre.(users).[ustate_key].(corrupt) → msg ∈ pre.(msg_history) →
  pre ⇝ replay_msg_result pre uid msg
```

Here, `replay_msg_result` is a function that builds a global state where `msg` is broadcasted. We call a sequence of global states a *trace* if it is nonempty and `g ⇝ g'` holds whenever `g` and `g'` are adjacent in the sequence.

Assumptions. To enable expressing relevant properties about our transition relation, we add assumptions about committees and quorums. This includes a function `committee` that determines self-selected committees, which we use to express properties of overlapping user quorums, as in the following statement, which says that for any two sets (quorums) of users of size at least `tau`, that are both subsets of the committee for the given round-period-step triple, there is an honest user for the step who belongs to both quorums:

```
Definition quorum_honest_overlap_statement (tau:ℕ) :=
 ∀ (trace:seq GState) (r p s:ℕ) (q1 q2:{fset UserId}),
  q1 ⊆ committee r p s → #|q1| ≥ tau →
  q2 ⊆ committee r p s → #|q2| ≥ tau →
  ∃ (honest_voter : UserId), honest_voter ∈ q1 ∧ honest_voter ∈ q2 ∧
   honest_during_step (r,p,s) honest_voter trace.
```

Similarly, we capture that a block was certified in a period as follows:

```
Definition certified_in_period (trace:seq GState) (tau r p:ℕ) (v:Value) :=
  ∃ (certvote_quorum:{fset UserId}),
  certvote_quorum ⊆ committee r p 3 ∧ #|certvote_quorum| ≥ tau ∧
  ∀ (voter:UserId), voter ∈ certvote_quorum →
   certvoted_in_path trace voter r p v.
```

This property is true for a trace if there exists a large-enough quorum of users selected for cert-voting who actually sent their votes along that trace for the given period (via `certvoted_in_path`, which we omit here). This is without loss of generality since a corrupted user who did not send its cert-vote can be simulated by a corrupted user who sent its vote but the message is received by nobody.

4 Asynchronous Safety

The analysis of the protocol in the computational model permits forking, albeit with negligible probability [1,3]. In contrast, we specify and prove formally in the *symbolic* model with idealized cryptographic primitives that at most one block is certified in a round, even in the face of adversary control over message delivery and corruption of users. We call this property *asynchronous safety*:

```
Theorem asynchronous_safety : ∀ (g0:GState) (trace:seq GState) (r:ℕ),
  state_before_round r g0 → is_trace g0 trace →
  ∀ (p1:ℕ) (v1:Value), certified_in_period trace r p1 v1 →
  ∀ (p2:ℕ) (v2:Value), certified_in_period trace r p2 v2 →
  v1 = v2.
```

Here, the first precondition `state_before_round r g0` states that no user has taken any actions in round `r` in the initial global state `g0`, and the second precondition `is_trace g0 trace` states that `trace` follows $\leadsto$ and starts in `g0`.

Note that it is possible to end up with block certifications from multiple periods of a round. Specifically, during a network partition, which allows the adversary to delay messages, this can happen if cert-vote messages are delayed enough for some users to advance past the period where the first certification was produced. However, these multiple certifications will all be for the same block.

Proof Outline. The proof of asynchronous safety proceeds by case-splitting on whether the certifications are from the same period or different periods. For the first and the easier case, `p1 = p2`, we use quorum hypotheses to establish that there is an honest user that contributed a cert-vote to both certifications. Then, we conclude by applying the lemma `no_2_certvotes_in_p`, which establishes that an honest user cert-votes at most once in a period (proved by exhaustive analysis of possible transitions by an honest node):

```
Lemma no_2_certvotes_in_p : ∀ (g0:GState) (trace:seq GState) uid (r p:ℕ),
 is_trace g0 trace →
 ∀ idx1 v1, certvoted_in_path_at idx1 trace uid r p v1 →
   user_honest_at idx1 trace uid →
 ∀ idx2 v2, certvoted_in_path_at idx2 trace uid r p v2 →
   user_honest_at idx2 trace uid → idx1 = idx2 ∧ v1 = v2.
```

The second case (`p1` ≠ `p2`) is proved with the help of proving an invariant. This invariant first holds in the period that produces the first certification — say, `p1` for `v1`— and keeps holding for all later periods of the same round. The invariant is that no step of the period produces a quorum of open next-votes, and any quorum of value next-votes must be for `v1`.

5 Conclusion

We developed a model in Coq of the Algorand consensus protocol and outlined the specification and formal proof of its asynchronous safety. The model and the proof open up many possibilities for further formal verification of the protocol, most directly of *liveness* properties. In total, our Coq development [9] contains around 2000 specification lines and 4000 lines of proof scripts.

References

1. Algorand blockchain features (2019), `https://github.com/algorandfoundation/specs/blob/master/overview/Algorand_v1_spec-2.pdf`
2. Chen, J., Gorbunov, S., Micali, S., Vlachos, G.: Algorand Agreement: Super fast and partition resilient Byzantine agreement. Cryptology ePrint Archive, Report 2018/377 (2018), `https://eprint.iacr.org/2018/377`
3. Chen, J., Micali, S.: Algorand: A secure and efficient distributed ledger. Theoretical Computer Science **777**, 155–183 (2019)
4. Cohen, C.: finmap (2019), `https://github.com/math-comp/finmap`
5. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: SOSP. pp. 51–68 (2017)
6. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
7. Pîrlea, G., Sergey, I.: Mechanising blockchain consensus. In: CPP. pp. 78–90 (2018)
8. Rahli, V., Vukotic, I., Völp, M., Esteves-Verissimo, P.: Velisarios: Byzantine fault-tolerant protocols powered by Coq. In: ESOP. pp. 619–650 (2018)
9. Runtime Verification, Inc.: Algorand verification (2019), `https://github.com/runtimeverification/algorand-verification`
10. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. PACMPL **2**(POPL), 28:1–28:30 (2018)
11. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.: Planning for change in a formal verification of the Raft consensus protocol. In: CPP. pp. 154–165 (2016)

Smart Contracts: Application Scenarios for Deductive Program Verification

Bernhard Beckert, Jonas Schiffl, and Mattias Ulbrich

Karlsruhe Institute of Technology, Karlsruhe, Germany
`bernhard.beckert@kit.edu, jonas.schiffl@kit.edu, mattias.ulbrich@kit.edu`

Abstract. Smart contracts are programs that run on a distributed ledger platform. They usually manage resources representing valuable assets. Moreover, their source code is visible to potential attackers, they are distributed, and bugs are hard to fix. Thus, they are susceptible to attacks exploiting programming errors. Their vulnerability makes a rigorous formal analysis of the functional correctness of smart contracts highly desirable.
In this short paper, we show that the architecture of smart contract platforms offers a computation model for smart contracts that yields itself naturally to deductive program verification. We discuss different classes of correctness properties of distributed ledger applications, and show that design-by-contract verification tools are suitable to prove these properties. We present experiments where we apply the KeY verification tool to smart contracts in the Hyperledger Fabric framework which are implemented in Java and specified using the Java Modeling Language.

1 Introduction

Smart contracts are programs that work in conjunction with a distributed ledger. They automatically manage resources on that ledger. Multiple distributed ledger platforms supporting smart contracts have been developed, most prominently the public Ethereum blockchain. Smart contracts manage resources representing virtual or real-world assets. Their source code is visible to potential attackers. Therefore, they are susceptible to attacks exploiting errors in the program source code. Furthermore, smart contracts cannot be easily changed after deployment. They need to be correct upon deployment, and formal methods should be used for ensuring their correctness [3].

In this paper, we describe the computational model of smart contracts, which makes them an ideal target for deductive program verification. We discuss different notions of smart contract correctness, and the implications for formal verification.

We focus on the Hyperledger Fabric [4] architecture. Fabric is a framework for the operation of private, permission-based distributed ledger networks. Smart contracts in Fabric can currently be written in Go, Java, and Javascript. While our concrete verification efforts target Fabric smart contracts written in Java, much of the concepts can be generalized to other programming languages, and also to other smart contract platforms.

The KeY system [1], which we used for experiments, is a deductive program verification tool for verifying Java programs w.r.t. a formal specification. KeY follows the principle of design-by-contract, i.e., system properties are broken down into method specifications called *contracts* that must be individually proven correct. Specifications for KeY are written in the Java Modeling Language [7], the de-facto standard language for formal specification of Java programs. For verification, KeY uses a deductive component operating on a sequent calculus for JavaDL, a program logic for Java.

In Section 2, we describe an abstract computational model for applications in a distributed ledger architecture. In Section 3, we discuss different notions and classes of smart contract correctness w.r.t. that model. Then, in Section 4, we describe how properties from these classes can be verified in the KeY tool. Finally, we draw some conclusions and discuss future work in Section 5.

2 Distributed Ledger Infrastructure and the Computational Model

Smart contract platforms are complex systems. Their functionality is spread across several layers and components. Some components are by necessity part of every smart contract platform, other components are unique to certain types of smart contract systems.

The correct behavior of a smart contract depends on all components of the distributed ledger architecture. This includes: the implementation of the blockchain data structure, which ensures that the shared history cannot be changed; the consensus and ordering algorithms for creating a single view of the system state; the cryptography modules for chain integrity and the public key infrastructure; and the network layer, which ensures correct distribution of transaction requests and new blocks.

If all these components work correctly, they provide an abstract computational model for the execution of smart contract applications in a distributed ledger system. This computational model can be described as follows: A distributed ledger platform behaves like a (non-distributed) single-core machine which takes requests (in the form of function calls) from clients. The execution of a request (a transaction) is atomic and sequential. The machine's storage is a key-value database in which serialized objects are stored at unique addresses. The storage can only be modified through client requests. The overall state of the storage is determined entirely by the order in which requests are taken. No assumptions can be made about the relationship between the order of requests made by the clients and the actual order of execution. However, it can be assumed that every request is eventually executed. All requests are recorded, even if they do not modify the state or are malformed.

3 Correctness of Smart Contracts

In the previous section, we have described the abstraction provided by smart contract architectures: It behaves like a single-core machine operating on a database storage and taking requests from clients. In this section, we discuss how this abstraction is useful for applying program verification techniques and tools. We give an overview of different classes of smart contract correctness properties and characterize the requirements and challenges for formal analysis that each class entails. The properties are roughly ordered by the effort required to prove them. Existing approaches to verification of smart contracts are given as examples for each class.

3.1 Generic Properties

Generic properties are independent of the concrete smart contract application and its functionality, i.e., there is not need to write property specifications for individual contracts. Typical examples of generic properties are termination for all inputs, absence of exceptions (such as null-pointer dereference), and absence of type errors.

Program properties such as termination are undecidable in general, and proofs may be non-trivial and require heavy-weight verification tools. Nevertheless, many generic properties can be validated by syntactical methods like type checking or simple static analysis. They are less precise than program verification and produce false alarms in case of doubt, but are still very useful in practice. Especially in the context of Ethereum, there is a wide variety of static analysis tools, e.g. [8,9], that can show the absence of known anti-patterns or vulnerabilities, like the notorious reentrance vulnerability, or inaccessible funds. For Hyperledger Fabric, there exists a tool which statically checks a smart contract for anti-patterns like non-determinism or local state.[1]

3.2 Specific Correctness Properties of Single Transactions

Correctness of a smart contract applications cannot be captured by generic properties alone: There has to be some formal specification which expresses the expected resp. required behavior of a program. Smart contract functions, which are atomic and deterministic in our computational model, are the basic modules of smart contracts (much like methods are basic components of programs), and therefore also the basic targets for correctness verification. The specification of a function consists of a precondition, which states what conditions the caller of the function has to satisfy, and a postcondition expressing what conditions are guaranteed to hold after the transaction (i.e., the function execution).

In case of smart contracts, the precondition should generally be empty because no assumptions about the state of the ledger should be necessary for correctness; furthermore, the values of the call parameters could be chosen by a malicious agent, and correctness properties need to hold for all possible input values.

Examples of specific properties of a single transaction include functional correctness statements (e.g., “the specified amount is deducted from the account if sufficient funds are available, otherwise the account remains unchanged”) and statements about which locations on the ledger a transaction is allowed to modify.

An approach to verification of single transaction correctness using the Why3 verification platform has been proposed [6]; our own approach using the KeY tool is discussed in Section 4.

3.3 Correctness of Distributed Ledger Applications

While transactions are equivalent to individual program functions, a *distributed ledger application* (DLA) is equivalent to a reactive program whose functions can be called by external agents. Informally, a DLA is a part of a smart contract network concerned with one specific task, like running an auction or providing a bank service. More precisely, a DLA is the set of all transactions that can

[1] https://chaincode.chainsecurity.com/

affect a given set of storage locations (including transactions that cannot access a storage location but are used in the calculation of the values being written).

While correctness of the component transactions is a necessary pre-requisite for the correctness of the DLA, there are properties which inherently are properties of transaction traces. They cannot be readily expressed as correctness properties of single transactions. To break them down into a set of single-transaction properties is a non-trivial process. Examples for this class of properties include invariants (e.g., "the overall amount of funds stays the same" for a banking application) and liveness properties giving the guarantee that some condition will eventually be fulfilled. Complex properties of this kind typically are expressed in temporal logic.

4 Verification of Smart Contracts Using the KeY Tool

In this section, we discuss verification of smart contract correctness using the KeY tool. The abstract computational model devised in Section 2 is an excellent fit for KeY because, in this setting, a distributed ledger application can be viewed as the equivalent of a Java program where single transactions correspond to Java methods. Thus, the KeY tool, which is designed for verifying Java programs, can be utilized for DLA verification, requiring only minor adaptations. These adaptations mostly concern the nature of the storage, since KeY operates on a heap with object references, while the distributed ledger application's storage is a database of serialized objects. Furthermore, due to the unknown order of execution and the fact that different agents operate within the shared program, there cannot be any assumptions as to the contents of the storage or order of transaction execution.

As an example, we consider a Hyperledger Fabric smart contract that implements functionality for simple auctions. It consists of three public methods, which allow clients to (1) start auctions for items they want to auction off, (2) bid for existing auctions, and (3) close an auction (an action that requires administrative credentials):

```
void createAuction(List<Item> items, int minBid, int endTime) { ... }
void bid(int auctionID, int bid) { ... }
void closeAuction(int AuctionID) { ... }
```

4.1 Generic Properties

The KeY tool can be used to verify any generic property. As a heavy-weight verification tool, it is particularly useful for properties that cannot be handled by light-weight tools resp. that require KeY's higher precision to avoid too many false alarms. Examples are program termination and the absence of exceptional behavior (the Java Modeling Language keyword `normal_behavior` can be used to specify that a method terminates without exception).

While constructing proofs for such properties is a non-trivial task in general, typical smart contracts are compact and lack complex control flows. In such settings, proofs of termination and absence of exceptions can be expected to be found automatically by KeY, requiring none or minimal auxiliary specifications.

4.2 Chaincode Transaction Correctness

Verification of Hyperledger Fabric chaincode functions, if written in Java, is possible in KeY. The difference between a normal Java program and our computational model is in the storage: While Java programs operate on a heap, a Fabric Smart Contract operates on an (abstracted) key-value database storing serialized objects. In our experiments, this difference was addressed by an extension of the KeY tool, including an axiomatisation of the read/write interface of the Fabric ledger, a model of the ledger on a logical level, and the introduction of abstract data types for each type of object that is managed by the smart contract [5].

In the auction example, one might want to specify the `closeAuction()` method as follows:

```
/*@ ensures read(ID) != null ==> read(ID).closed;
  @ ensures (\forall Item i \in read(ID).items;
  @               i.owner_id == read(ID).highestBidderID);
  @ modifies read(ID);
  @*/
void closeAuction(int ID) { ... }
```

This JML specification is somewhat simplified for readability; the `read` function is an abstraction for accessing the ledger, i.e., reading and deserializing the object at the given location. The specification states that, if the auction object at `ID` is not null, then after execution of the `closeAuction()` method the `closed` flag must be correctly set; furthermore all items in the auction must belong to the highest bidder (as indicated by the `owner_id` attribute). The `modifies` clause states that only the object at the location specified by `ID` may be changed by the transaction, ensuring that no unexpected side effects are possible. This method contract can be loaded and proven using our smart-contract version of KeY; the logical rules necessary for handling the data types stored on the ledger (in this case, auctions, items, and participants) are created automatically. The proof requires some user interaction, since the new rules have not yet been included in the automation mechanism of the prover.

There exists a comparable approach for using KeY to verify Ethereum smart contracts [2].

4.3 Correctness of Distributed Ledger Applications

More complex properties can be reasoned about in KeY using class invariants, two-state invariants, and counters, thereby reducing complex properties of transaction traces (including temporal logic properties) to KeY's method-modular approach. For example, the specification of the auction application could state that, as long an the auction is open, the items that are offered still belong to the auctioneer:

```
//@ invariant (\forall Auction a; !a.closed ==>
                    (\forall item i \in a.items;
                         i.owner_id == a.auctioneer_id));
```

If every bidder has to deposit the funds for their bid in the auction, the specification could state that as long as the auction remains open, the sum of the funds in the auction remains the same or increases, but never decreases. This can be expressed with a history constraint:

```
//@ constraint (\forall Auction a; !a.closed ==>
                     \old(a.funds) <= a.funds);
```

Though this constraint can easily be expressed in the Java Modeling Language, proving in KeY that a smart contract conforms to this specification is currently infeasible due to the large amount of user interactions that is necessary to close the proof, and due to the inefficiencies of our current approach regarding the handling of reading from and writing to the ledger.

5 Conclusion and Future Work

We have outlined the setting in which deductive program verification of distributed ledger applications takes place and shown that the KeY verification tool is suitable to prove different classes of correctness properties which are interesting in smart contract platforms.

The extensions to KeY which enable verification of Hyperledger Fabric smart contracts are still in a prototypical state. Further improvements are necessary to improve scalability and enable proofs of more complex properties.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book: From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (Dec 2016)
2. Ahrendt, W., Bubel, R., Ellul, J., Pace, G.J., Pardo, R., Rebiscoul, V., Schneider, G.: Verification of smart contract business logic. In: 8th IPM International Conference on Fundamentals of Software Engineering (FSEN). To appear. (2019)
3. Ahrendt, W., Pace, G.J., Schneider, G.: Smart Contracts: A Killer Application for Deductive Source Code Verification. In: Müller, P., Schaefer, I. (eds.) Principled Software Development: Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday, pp. 1–18. Springer International Publishing (2018)
4. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S.W., Yellick, J.: Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In: Proceedings of the Thirteenth EuroSys Conference. pp. 30:1–30:15. EuroSys '18, ACM (2018), http://doi.acm.org/10.1145/3190508.3190538
5. Beckert, B., Herda, M., Kirsten, M., Schiffl, J.: Formal Specification and Verification of Hyperledger Fabric Chaincode. In: Bai, G., Biswas, K. (eds.) 3rd Symposium on Distributed Ledger Technology (SDLT-2018) co-located with ICFEM 2018: the 20th International Conference on Formal Engineering Methods (Nov 2018), https://symposium-dlt.org/
6. Bhargavan, K., Swamy, N., Zanella-Béguelin, S., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T.: Formal Verification of Smart Contracts: Short Paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS'16. pp. 91–96. ACM Press, Vienna, Austria (2016), http://dl.acm.org/citation.cfm?doid=2993600.2993611
7. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML Reference Manual (May 31, 2013), draft Revision 2344
8. Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 653–663. ACSAC '18, ACM, New York, NY, USA (2018), http://doi.acm.org/10.1145/3274694.3274743, event-place: San Juan, PR, USA
9. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82. ACM (2018)

Formal specification of a security framework for smart contracts

Mikhail Mandrykin[1], Jake O'Shannessy[2], Jacob Payne[2], and Ilya Shchepetkov[1]

[1] ISP RAS, Moscow, Russia
{mandrykin,shchepetkov}@ispras.ru
[2] Daohub, San Francisco, USA
joshannessy@gmail.com, jacob@daohub.io

Abstract. As smart contracts are growing in size and complexity, it becomes harder and harder to ensure their correctness and security. Due to the lack of isolation mechanisms a single mistake or vulnerability in the code can bring the whole system down, and due to this smart contract upgrades can be especially dangerous. Traditional ways to ensure the security of a smart contract, including DSLs, auditing and static analysis, are used before the code is deployed to the blockchain, and thus offer no protection after the deployment. After each upgrade the whole code need to be verified again, which is a difficult and time-consuming process that is prone to errors. To address these issues a security protocol and framework for smart contracts called Cap9 was developed. It provides developers the ability to perform upgrades in a secure and robust manner, and improves isolation and transparency through the use of a low level capability-based security model. We have used Isabelle/HOL to develop a formal specification of the Cap9 framework and prove its consistency. The paper presents a refinement-based approach that we used to create the specification, as well as discussion of some encountered difficulties during this process.

Keywords: formal specification · smart contracts · isabelle · security.

1 Introduction

Ethereum [6] is a global blockchain platform for decentralised applications with a built-in Turing-complete programming language. This language is used to create smart contracts — automated general-purpose programs that have access to the state of the blockchain, can store persistent data and exchange transactions with other contracts and users. Such contracts have a number of use-cases in different areas: finance, insurance, intellectual property, internet of things, voting, and others.

However, creating a *reliable* and *secure* smart contract can be extremely challenging. Ethereum guarantees that the code of a smart contract would be executed precisely as it is written through the use of a consensus protocol, which resolves potential conflicts between the nodes in the blockchain network. It prevents malicious nodes from disrupting and changing the execution process, but do not protect from the flaws and mistakes in the code itself. And due to the lack of any other control on the execution of the code any uncaught mistake can potentially compromise not only the contract itself, but also other contracts that are interacting with it and expect a certain behavior from it.

Such flaws can be turned into vulnerabilities and cause a great harm, and there are many examples of such vulnerabilities and attacks that exploit them [1]. Developers can ensure the security of a contract using auditing, various static analysis tools [15, 11], domain-specific languages [5, 7], or formal verification [2]. These are excellent tools and methods that can significantly improve the quality of the code. But they are not so effective during the upgrades, which is a common process for almost every sufficiently sophisticated smart contract. Upgrades are necessary because it is the only way to fix a bug that was missed during the verification process. However, they can also introduce their own bugs, so after each upgrade the code needs to be verified again, which may cost a lot of time and effort.

These issues are addressed by the Cap9 framework [4]. It provides means to isolate contracts from each other and restrict them from doing dangerous state-changing actions unsupervised, thus greatly reducing risks of upgrades and consequences of uncaught mistakes. Cap9 achieves this by using a low level capability-based security model, which allows to explicitly define what can or can not be done by any particular contract. Once defined, such capabilities, or permissions, are visible to anyone and can be easily understood and independently checked, thus increasing transparency of the system.

In order to be trusted, the Cap9 framework itself needs to be formally verified. The specification of the framework must be formalised and proved, in order to show that it is consistent and satisfies the stated properties. Then the implementation, which is a smart contract itself, must be proved to be compliant with its specification. In this paper we are focusing only on the first part — on developing and proving a formal specification of the Cap9 framework using the Isabelle/HOL theorem prover [17] The paper presents a refinement-based approach that we used to create the specification, and evaluates the chosen formal method by describing encountered difficulties during this process.

The following section outlines the features and capabilities of the Cap9 framework. Section 3 presents the Isabelle/HOL specification, as well as the difficulties we have encountered and the refinement process we used to develop it. Related work is reviewed in Section 4. The last section concludes the paper and considers future work.

2 Cap9 Framework

The Cap9 framework achieves isolation by interposing itself between the smart contracts that are running on top of it and potentially dangerous actions that they can perform, including calling other smart contracts, writing to the storage and creating new contracts. Such actions can be performed only using special "System Call" interface provided by the framework. Via this interface it has complete control over what contracts can and cannot do. Each time a system call is executed Cap9 conducts various runtime security checks to ensure that a calling contract indeed has necessary rights to perform a requested action. It works similar to how operating system kernels manage accesses of programs to the hardware (like memory and CPU) and protect programs from unauthorised accesses to each other.

In order to ensure that a contract correctly uses the system call interface and does not perform any actions bypassing the framework its source code needs to be verified. Cap9 does it on-chain and it checks that the source code does not contain any forbidden instructions, like ones allowing to make state changes, make an external call, or self destruct. Once the code is verified the corresponding contract can be registered in the framework as a *procedure* and thus access its features.

There are system calls available to securely perform the following actions:

– Register new procedure in the framework;
– Delete or registered procedure;
– Internally call a registered procedure;
– Write data to the storage;
– Append log record with given topics;
– Externally call another Ethereum address and/or send Ether;
– Mark a procedure as an *entry* procedure — a one that would handle all the incoming external calls to this contract system or organisation.

As a typical smart contract, Cap9 has access to the storage — a persistent 256 x 256 bits key-value store. A small part of it is restricted and can be used only by the framework itself. It has a strict format and is used to store the list of registered procedures, as well as procedure data, addresses of entry and current procedures and the Ethereum address of the deployed framework itself. This part is called the *kernel storage*. The rest of the storage is open to use by any registered procedure either directly (in case of read) or through a dedicated system call (in case of write).

Traditional kernels have a lot of abstraction layers between programs and hardware. Unlike them, Cap9 exposes all the underlying Ethereum mechanisms directly to the contracts, with only a thin permission layer between them. This layer implements a capability-based access control, according to which in order to execute a system call a procedure must posses a *capability* with an explicit permission. Such capability has a strict format, which is different for each available type of system calls.

Capabilities can be used to restrict components of a smart contract system and thus to implement the principle of least privilege. They can even be used as base primitives to create a custom high-level security policy model to better fit a particular use case. Such policy would be simple to analyze and understand, but able to limit possible damage from bugs in the code or various malicious actions (including replacing the code of a contract via the upgrade mechanism).

Cap9 is compatible with both EVM and Ewasm applications.

3 Formal Specification

The main goal of formalizing the interface specification of the Cap9 security framework was to ensure internal consistency and completeness of its description as well as to provide a reliable reference for all of its implementations. The reference should eventually serve as an intermediate between the users and the developers of any Cap9 implementation ensuring full compatibility of all further system uses and implementations. The source specification itself is formulated as a detailed textual description of the system interface [19], which is language-agnostic and relies on the binary interfaces of the underlying virtual machine. Thus all the data mentioned in the specification is given an explicit concrete bit-level representation, which is intended to be shared by all system users and implementations.

3.1 Consolidation of low-level representation with high-level semantics

One of the immediately arising challenges of formally verifying a system with very explicit specifications on concrete data representation is efficiently establishing a correspondence between this representation and the corresponding intended semantics, which is used for actual reasoning about the system and therefore for the actual proof.

A particular example in our case is the representation and the semantics of capability subsets. Each capability of every procedure in the system logically corresponds to a set of admissible values for some parameter configuration, such as kernel storage address (for writing to the storage), Ethereum address and amount of gas for external procedure call, log message with several special topic fields etc. Each such set is composed of a (not necessarily disjoint) union of a number of subsets, which in their turn directly correspond to some fixed representations. A subset of writable addresses, for example, is represented as a pair of the starting address and the length of a continuous range of admissible addresses. Thus the entire write capability of any kernel procedure is a union of such continuous address ranges.

But it's important to note that while on one hand we clearly need to state the set semantics of the write capability (as a generally arbitrary set of addresses), in particular this is especially convenient semantics to be used for proofs of generic capability properties, such as transitivity; on the other hand, however, we have a clearly indicated format of the corresponding capability representation stated in the system specification, which is not a set, but a range of storage cells holding the bit-wise representations of the starting addresses and lengths of the corresponding ranges.

If we stick with the specified representation, we will be unable to efficiently use many powerful automated reasoning tools provided with Isabelle/HOL, such as the classical reasoner and the simplifier readily pre-configured for the set operations. However, if we just use the set interpretation, the specification on the concrete representation will be notoriously hard to express. Hence we likely need several different formalizations of a notion of capability on several levels of abstraction. We actually used three representations: the concrete bit-wise representation, the more abstract representation with the length of the range expressed as natural number (and with an additional invariant), and finally the set representation. By using separate representations we ended up with small simple proofs for both generic capability properties and their concrete representations.

3.2 Correspondence relation vs. representation function

Eventually we decided to employ the same refinement approach with several formalizations for the entire specification, thus obtaining two representations of the whole system: the structured high-level representation with additional type invariants and the low-level representation as the mapping from 32-byte addresses to 32-byte values, i.e. the state of the kernel storage. However, using separate representations raises a problem of efficiently establishing the correspondence between them. Initially we tried a more general approach based on the correspondence relation. Yet to properly transfer properties of the high-level representation to the low-level one, the relation should enjoy at least two properties: injectivity and non-empty image of every singleton:

lemma *rel_injective*: "$[\![s \Vdash \sigma_1;\ s \Vdash \sigma_2]\!] \Longrightarrow \sigma_1 = \sigma_2$"
lemma *non_empty_singleton*: "$\exists\ s.\ s \Vdash \sigma$"

Here $\Vdash$ stands for the correspondence relation, σ — for the high-level representation and s — for the concrete one. We noticed that proving the second lemma essentially requires defining a function mapping an abstract representation to the corresponding concrete one. Thus this approach results in significant redundancy in a sense that both the function defined for the sake of proving the second lemma and the correspondence relation itself repeat essentially the same constraints on the low-level representation. For a very simple example consider:

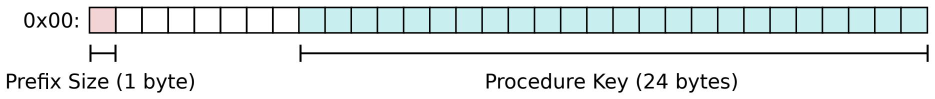
definition *models* :: "$(word32 \Rightarrow word32) \Rightarrow kernel \Rightarrow bool$" ("$\_ \Vdash \_$") **where**
  "$s \Vdash \sigma \equiv unat\ (s\ (addr\ Nprocs)) = nprocs\ \sigma$"
definition "$witness\ \sigma\ a \equiv case\ addr^{-1}\ a\ of\ Nprocs \Rightarrow of\_nat\ (nprocs\ \sigma)$"

Here not only we need to repeatedly state the relationship between the value of kernel storage at address *addr Nprocs* and the number of procedures registered in the system (*nprocs* σ) twice, but we also potentially have to define the address encoding and decoding functions (*addr* and $addr^{-1}$) separately and to prove the lemma about their correspondence. We discuss our approach to address encoding in the following section and here only emphasize the redundancy arising from the approach based on the correspondence relation.

At the same time, the major reason for introducing the correspondence relation instead of using a function is an inherent ambiguity of the encoding of the high-level representation into the low-level one. However, after carefully revisiting the initial specification of the system we noticed that the ambiguity of representation in our system actually arises only from the unused storage memory rather than from the presence of any truly distinct ways of representing the same state. But this particular kind of ambiguity can be efficiently expressed using a representation function with an additional parameter — i. e. the state of the unused memory.

Let's illustrate our formalization approach that is based on representation functions on the example of Procedure Call capability. The specification of this capability is as follows:

The capability format for the Call Procedure system call defines a range of procedure keys what the capability allows one to call. This is defined as a base procedure key b and a prefix s. Given this capability, a procedure may call any procedure where the first s bits of the key of that procedure are the same as the first s bits of procedure key b.



Here the unused space is left blank. Beforehand we strive to make the actual formulation of the arising injectivity lemma as simple as possible by eliminating premises of the lemma and turning them into type invariants. So we introduce the following definitions:

typedef *prefix_size* = "$\{n :: nat.\ n \le LENGTH(key)\}$"
definition "$prefix\_size\_rep\ s \equiv of\_nat\ \lfloor s \rfloor :: byte$" **for** s :: *prefix_size*
type_synonym *prefixed_capability* = "$prefix\_size \times key$"
definition — set interpretation of single write capability
  "$set\_of\_pref\_cap\ sk \equiv let\ (s,\ k) = sk\ in$
  $\{k' :: key.\ take\ \lfloor s \rfloor\ (to\_bl\ k') = take\ \lfloor s \rfloor\ (to\_bl\ k)\}$"
    for sk :: *prefixed_capability*
adhoc_overloading *rep prefix_size_rep* — *prefix_size_rep* is now denoted as $\lfloor\cdot\rfloor$
definition — low-level (storage) representation of single write capability
  "$pref\_cap\_rep\ sk\ r \equiv let\ (s,\ k) = sk\ in$
  $\lfloor s \rfloor\ _1\Diamond\ k\ OR\ r \restriction \{LENGTH(key)..{<}LENGTH(word32) - LENGTH(byte)\}$"
    for sk :: *prefixed_capability*

Here the parameter r represents some arbitrary memory state being overwritten by the representation of the capability. The binary representation of r is truncated (by bit-wise conjunction with a mask) to fill the range of unused bits before combining it with the zero-padded representation. The value of unused memory r is propagated across all representation functions in a composable way, so all low-level representations are formalized with plain single-valued functions. This approach not only allows simple transfer of all high-level properties to the low-level representation, but also avoids the need in explicit definitions of the corresponding inverse (decoding) functions. A single definition is enough to reuse the encoding functions (along with their injectivity proofs) for the specifications of operations that require decoding of representations:

definition
    "$maybe\_inv\ f\ y \equiv if\ y \in range\ f\ then\ Some\ (the\_inv\ f\ y)\ else\ None$"

Since we don't verify the actual implementation of the decoding functions, this implicit definition is sufficient and greatly simplifies proofs.

3.3 Disjointness of addresses

Another problem arising from detailed low-level specifications of memory layout, such as the layout of the kernel storage, is the problem of reasoning about non-intersecting memory areas. While in the context of program verification there are such well-known approaches to reasoning about disjoint memory footprints as separation logic [18] and dynamic frames [12], in our context of formalizing the specification (rather than the implementation) of the system these approaches turned out to be both too abstract and too heavyweight. Too abstract since in separation logic the particular concrete layout of the memory footprints is left entirely abstract, while we needed to formalize the actual mapping of the data structures to the mostly fixed address ranges they should occupy. Too heavyweight since to represent the encoding of the whole kernel state with either separation logic or dynamic frames we would need to use some additional means to set up the embedding of the corresponding reasoning mechanism into plain HOL, while not having any real need in verifying code involving updates to the system state. In our approach we simply treated kernel addresses as semantic entities with some ascribed low-level representations (concrete values). Then following our general use of representation functions we defined the representation of addresses and its inverse. The inverse than can be directly used to specify the storage layout and prove the injectivity of the overall encoding with minimal effort. Here's an illustrative example:

```
typedef offset = "{ n :: nat. n < 2 ^ LENGTH(byte)}"
        morphisms off_rep off
datatype address = Nprocs | Curr_proc | Proc_heap offset
definition "addr_rep a ≡ case a of
        Nprocs          ⇒ 0x0000
      | Curr_proc       ⇒ 0x0100
      | Proc_heap offs ⇒ 0x02 OR of_nat (off_rep offs)"
definition "addr_inv ≡ maybe_inv addr_rep"
definition "encode σ r a ≡ case addr_inv a of
          Some a'          ⇒ case a' of
              Nprocs          ⇒ of_nat (nprocs σ) OR (r a) ↾ ...
            | Curr_proc       ⇒ of_nat (curr_proc σ) OR (r a) ↾ ...
            | Proc_heap offs ⇒ encode_heap σ offs r
        | None                 ⇒ r a"
```

Also note the filler of the unused memory r being passed over in a top-down manner starting from the outermost representation function.

Now we move from the problems arising from the detailed low-level specification of our target system to some more general issues of formalization and formal proofs within the Isabelle/HOL framework that we encountered during verification.

3.4 General Isabelle/HOL limitations

Bit-vector concatenation An example of a minor, though noticeable limitations of the simple Hindley-Milner type system employed within the Isabelle/HOL framework is its inability to express type-level sum (and other simple arithmetic operations), while still being able to express type-level numbers. For an illustration of the issue consider the following definition of bit-vector concatenation function from the HOL-Word library that comprises an extensive Isabelle/HOL formalization of fixed-size bit-vectors, corresponding operations and their various properties:

```
definition word_cat :: "'a::len0 word ⇒ 'b::len0 word ⇒ 'c::len0 word" ...
```

The annotation of the form *'a::len0* constrains the type parameter *'a* to belong to the *len0* type class, which has the corresponding associated operation *LENGTH('a)* returning a natural number. Thus we essentially gen type-level numbers that can be injected into terms as natural numbers with the use of the *LENGTH* operation. However, as we can see in the definition of *word_cat*, the result of this function has the type *'c::len0* that is generally unrelated to the parameter types *'a* and *'b*. This has two basically unavoidable, but undesirable consequences:

- Since there is no way of further constraining the resulting parameter type *'c::len0*, the function *word_cat* is forced to be partial. Generally, there is nothing particularly special about handling of partial functions within the Isabelle/HOL framework, but their presence has at least one undesirable consequence for formalization of system interface specifications, which we discuss further in this section.
- Since the resulting type parameter *'c::len0* cannot be automatically inferred from the arguments of *word_cat*, if has to be explicitly specified. Normally, this doesn't lead to a significant type annotation burden since the parameter can be propagated by type inference from some term with a known type. But in case of consecutive (nested or chained) *word_cat* applications, the inner type parameters become essentially inaccessible for further type propagation or inference and have to be specified explicitly e.g.

  ```
  definition "entry_proc_addr ≡ word_cat
      (word_cat
        (word_cat (k_prefix :: 32 word) (0x04 :: byte) :: 40 word)
        (0 :: 192 word) :: 232 word)
      (0x000000 :: 24 word) :: 256 word"
  ```

  This can be slightly mitigated by introducing some ad-hoc monomorphic notation for hexidecimal numbers (e.g. syntactically reconstructing the type annotation from the length of the input hexidecimal representation), but this approach still quickly becomes unwieldy in practice, especially in the context of the great available variety of Ethereum bit-vector types with various lengths.

First we propose a relatively simple remedy for the second problem. We actually used our own definition of a concatenation function with a fixed result type (the largest needed length of 256 bits) and parameter types of arbitrary length that is ignored. Instead we provided the necessary length of the second argument as an additional explicit parameter. Thus the whole issue of dealing with lengths was shifted from the type to the term level eliminating the need in any type-level representations altogether. This resulted in more approachable definitions e.g.

definition "*entry_proc_addr* $\equiv$
  (k_prefix :: 32 word) $\bowtie_{224}$ 0x04 $\bowtie_{216}$ (0 :: 192 word) $\bowtie_{24}$ 0x000000"

Here $\cdot \bowtie . \cdot$ denotes our concatenation function. In our opinion in the lack of dependent types [3] or other expressive capabilities of the type system the use of logical (term-level) constraints may be often preferable to some limited meta-logical (e.g. type-level) extensions such as the use of type classes.

Now we move to the second problem.

Partiality The presence of partial functions in the specification of an interface of the system has a subtle undesirable property — unpredictability stemming from the undefined results returned by the partial functions. Consider the following very typical and general preservation lemma:

lemma *preservation*: "$I\ s \Longrightarrow I\ (op\ s\ a)$"

Here I is an invariant of the system and op is an operation on the system with an argument a. Let's imagine an example instance of this kind of lemma: Let s be a natural number, $I\ s$ be the predicate $s > 0$ and op correspond to the operation $s \leftarrow s + s \operatorname{div} a$. Looking at the general statement of the lemma, a rather natural interpretation of such a preservation property would be that any application of the operation op to the system is "safe" as it preserves its invariant. However in our particular example it's obvious that even though the application of op with $a = 0$ provably preserves the invariant, it actually has entirely unpredictable consequences for the system. So specifications of operations on the system involving partial functions may considerably mislead the reader of the specification while remaining perfectly correct form the purely logical perspective. If the formal specification is to serve as a formal documentation on the system this fact may significantly undermine the value of applying the formal methodology for that purpose. Fortunately, there are various ways to strengthen the specification to exclude such unintuitive definitions. For our specification we additionally proved the following injectivity-like lemmas for every operation:

lemma *injectivity_like*: "$op\ s\ a = op\ s\ b \Longrightarrow a \sim b$"

Here $\sim$ denotes some notion of equivalence for arguments of the operation in a sense that equivalent arguments produce equivalent results. In case the operation op actually involves some non-determinism, the formulation of the lemma should be adjusted accordingly, thus making this non-determinism explicitly exposed for the reader. The proof of such a lemma is enough to exclude any hidden non-determinism, since for any non-trivial equivalence relation $\sim$ $(\exists a'.\ a' \nsim a)$ if the *op* has non-deterministic result on a, *op* a may be arbitrarily chosen to be equal to *op* a' and the relation $a \sim a'$ then cannot be established.

Dependent products Another limitation arising from the lack of dependent types or other expressive type system features is inability to directly express dependent products i.e. types of the form $\prod_{x::'a} f(x)$, where f is a type-level function on the value x of some type $'a$. A typical example of a situation, where this seems very natural is a list of pairs of the form "*capability_type* $\times$ *capability_representation*" or triples of the form "*capability_type* $\times$ *capability_encoding_function* $\times$ *capability_decoding_function*". Here the right members of the tuples should have appropriate types depending on the corresponding type of the capability e.g. if the value of the first member is "*Write*", than the type of the second member should be "*write_capability*" or "*write_capability* $\Rightarrow$ *byte list*". Such types cannot be directly expressed within the Isabelle/HOL framework, although many modern functional programming languages are capable of that due to the availability of generalized abstract data types (GADTs). We used type invariants in such cases:

```
definition "wf_cap (tc :: capability_type × capability) ≡  case tc of
   (Delete, Delete_cap _) ⇒ True
 | (Write,  Write_cap _)  ⇒ True
 | _                      ⇒ False"
typedef capability_pair = "Collect wf_cap"
```

Finally it's important to note an essential benefit of a logical framework with a very limited type system, which is its amenability to automation using existing readily available tools such as saturation-based provers (E-priver, Vampire, Metis) and SMT solvers (Z3, CVC4). In our experience their use within the Isabelle framework lead to great advantages ultimately outweighing all the limitations mentioned above.

4 Related Work

There are many examples of using formal methods for developing specifications of various systems. Isabelle/HOL was used to prove functional correctness of the seL4 operating system microkernel [13], providing a proof chain from the high-level abstract specification of the kernel, down to the executable machine code. The B-method was applied to create formal models of various safety-critical railway systems [14]. A dedicated specification language for defining the high-level abstract models was introduced in [20].

On the other hand, verification of smart contracts is almost exclusively concentrated on the contract implementation, omitting the separate formalisation of their specification. It is a valid approach if the specification is simple enough, which is not the case for the Cap9 framework.

There are several examples of formalisation of the Ethereum virtual machine: using the K framework [9], the Lem language [10], F* [8], which can serve as a basis for formal verification of the contract code. Why3 platform for deductive program verification was recently applied for writing and verifying smart contracts [16].

5 Conclusion and Future Work

We have developed a formal specification[3] of the Cap9 framework using the Isabelle/HOL theorem prover and proved its internal consistency. To create it we have employed a refinement approach based on representation functions, which allowed us to efficiently use powerful automated reasoning tools provided with Isabelle. We have found Isabelle/HOL to be suitable for developing specifications of smart contracts, although some minor issues were identified and outlined.

The next step is formal verification of the Ewasm implementation of the Cap9 framework for its compliance with the Isabelle/HOL specification, which may require developing some additional tools. Other possible direction is to develop and verify a higher level permission system that is based on the Cap9 primitives.

References

1. Atzei, N., Bartoletti, M., Cimoli, T.: A Survey of Attacks on Ethereum Smart Contracts SoK. In: Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204. pp. 164–186. Springer-Verlag New York, Inc., New York, NY, USA (2017). https://doi.org/10.1007/978-3-662-54455-6_8
2. Bhargavan, K., Swamy, N., Zanella-Bguelin, S., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T.: Formal Verification of Smart Contracts: Short Paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS'16. pp. 91–96. ACM Press, Vienna, Austria (2016). https://doi.org/10.1145/2993600.2993611
3. Bove, A., Dybjer, P.: Dependent Types at Work. In: Bove, A., Barbosa, L.S., Pardo, A., Pinto, J.S. (eds.) Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures, pp. 57–99. Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03153-3_2
4. Cap9 white paper. https://cap9.io/docs/Whitepaper.pdf, last accessed 2 Jul 2019
5. The Ergo language for smart legal contracts. https://www.accordproject.org/projects/ergo, last accessed 2 Jul 2019
6. Ethereum white paper. https://github.com/ethereum/wiki/wiki/White-Paper, last accessed 2 Jul 2019
7. Frantz, C.K., Nowostawski, M.: From Institutions to Code: Towards Automated Generation of Smart Contracts. In: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W). pp. 210–215 (Sep 2016). https://doi.org/10.1109/FAS-W.2016.53

[3] The specification is publicly available at https://github.com/Daohub-io/cap9-spec

8. Grishchenko, I., Maffei, M., Schneidewind, C.: A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In: Bauer, L., Ksters, R. (eds.) Principles of Security and Trust. pp. 243–269. Lecture Notes in Computer Science, Springer International Publishing (2018)
9. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). pp. 204–217. IEEE, Oxford (Jul 2018). https://doi.org/10.1109/CSF.2018.00022
10. Hirai, Y.: Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) Financial Cryptography and Data Security. pp. 520–535. Lecture Notes in Computer Science, Springer International Publishing (2017)
11. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: Analyzing Safety of Smart Contracts. In: Proceedings 2018 Network and Distributed System Security Symposium. Internet Society, San Diego, CA (2018). https://doi.org/10.14722/ndss.2018.23082
12. Kassios, I.T.: Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 268–283. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2006)
13. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems **32**(1), 1–70 (Feb 2014). https://doi.org/10.1145/2560537
14. Lecomte, T., Servat, T., Pouzancre, G.: Formal Methods in Safety-Critical Railway Systems. In: 10th Brasilian Symposium on Formal Methods. p. 10 (2007)
15. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making Smart Contracts Smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. CCS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978309
16. Nehai, Z., Bobot, F.: Deductive Proof of Ethereum Smart Contracts Using Why3. Research Report, CEA DILS (Apr 2019), https://hal.archives-ouvertes.fr/hal-02108987
17. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg (2002), https://www.springer.com/gp/book/9783540433767
18. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74 (Jul 2002). https://doi.org/10.1109/LICS.2002.1029817
19. Specification of the Cap9 framework. https://github.com/Daohub-io/cap9/blob/master/docs/spec/Cap9Spec.pdf, last accessed 2 Jul 2019
20. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z.: A Practical Verification Framework for Preemptive OS Kernels. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification, vol. 9780, pp. 59–79. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_4

Verifying Smart Contracts with Cubicle

Sylvain Conchon[1,2], Alexandrina Korneva[1], and Fatiha Zaïdi[1]

[1] LRI (CNRS & Univ. Paris-Sud), Université Paris-Saclay, F-91405 Orsay
[2] Inria, Université Paris-Saclay, F-91120 Palaiseau

Abstract. Smart contracts are programs that constitute the main ingredients of decentralized applications (DApps) running on top of blockchain networks like Ethereum, EOS or Tezos. From a computing point of view, smart contracts are reminiscent to distributed objects used through several entry points by an unknown number of *accounts*. Like any (concurrent) program, smart contracts may have bugs. But contrary to traditional applications, once deployed on a blockchain, DApps can neither be removed nor modified. Unfortunately, but not surprisingly, it has recently been revealed that a large number of smart contracts contain critical security vulnerabilities.
To help DApps programmers, we propose in this paper to model smart contracts into the declarative input language of Cubicle, a model checker for parameterized systems based on SMT. In our approach, smart contract code, as well as the transactional model of the blockchain, are encoded as a state machine on which safety properties of interest are encoded and verified. We show the success of our technique through the simple yet prime example of an auction. This preliminary result is very promising and lays the foundations for a complete and automatized framework for the design and certification of smart contracts.

Keywords: Blockchain, Smart-contracts, Model Checking, MCMT

1 Introduction

The number of decentralized applications (DApps) running on top of blockchain networks is growing very fast. According to [3], there are now more than 2,700 DApps available on the Ethereum and EOS platforms which generate 1.6 million transactions per day for a volume of 19 million USD. These DApps interact with the blockchain through around 40,000 smart contracts.

A smart contract is a stateful program stored in the blockchain with which a user (human or computer) can interact. From a legal perspective, a smart contract is an agreement whose execution is automated. As such, its revocation or modification is not always possible and worse than that, what the code of a smart contract does is *the law*... no matter what it may end up doing. Unfortunately, like any program, smart contracts may have bugs. Given the potential financial risks, finding these bugs before the origination of the contracts in the blockchain is an important challenge, both from economic and scientific points of view.

Various formal methods have been used to verify smart contracts. In [5], the authors present a shallow embedding of Solidity within F*, a programming language aimed at verification. Other similar approaches are based on deductive verification platforms like Why3 [11, 13]. Interactive proof assistants (*e.g.* Isabelle/HOL or Coq) have also been used for modeling and proving properties about Ethereum and Tezos smart contracts [4, 1].

A common thread here is the use of general-purpose frameworks based on sequential modeling languages. However, smart contracts can be considered state machines [2, 10] whose execution model, according to [14], is closer to that of a concurrent programming language rather than a sequential one. In this context, the use of model checking techniques becomes highly appropriate [12, 6].

An important aspect of blockchains is that they are completely open. As a consequence, smart contracts are state machines that need to be conceived for an unknown (and potentially varying) number of users. This parameterized side of blockchains has not previously been taken into account.

In this paper, we propose to model smart contracts into the declarative input language of Cubicle, a model checker for parameterized systems based on Satisfiability Modulo Theories (SMT) techniques. In our approach, smart contract code, as well as the transactional model of the blockchain, are encoded as a state machine on which safety properties of interest are encoded and verified. Our contributions are as follows:

– A two-layer framework for smart contract verification in Cubicle (Section 3). The first layer is a model of the blockchain transaction mechanism. The second layer models the smart contract itself.
– A description of how to express smart contract properties as Cubicle safety properties using both ghost variables and model instrumentation (Section 4)
– A way of interpreting Cubicle error traces as part of the smart contract development cycle (Section 5)

Throughout the paper, we illustrate our framework with a simple auction contract written in Solidity (Section 2).

2 A Motivating Example

Our example contract is based on an open auction example given in Solidity's official documentation. We will be focusing on the functions and variable declarations in Figure 1.

Once the SimpleAuction contract is created and the auction has begun, any person (client) can bid money to win. The contract has three key variables (lines 3-5):

– highestBidder, the client with the current highest bid (winner);
– highestBid, the bid amount of the above client;
– pendingReturns, a map of clients to their bids, used to return money.

In order to bid, a client needs to call the bid function on line 8. The requirements are that (i) the auction still be open and (ii) the client's bid be bigger than highestBid. If these conditions are fulfilled, pendingReturns for the old winner is set to the winner's old winning amount, and highestBidder and highestBid are set to the client (the new winner).
If a client wishes to withdraw their bid, they need to call the withdraw function on line 18. This function has no particular requirements. It simply checks that the client requesting their money has something in pendingReturns. If that is true, pendingReturns is set to zero and then the money is sent to the client via send on line 22. If send fails then pendingReturns reverts back to the original sum and withdraw returns false to indicate that the action failed. If everything worked, the function returns true.

```
 1 contract SimpleAuction {
 2     //...
 3     address public highestBidder;
 4     uint public highestBid;
 5     mapping(address => uint) pendingReturns;
 6     bool ended;
 7
 8     function bid() public payable {
 9         require(now <= auctionEndTime,
10                     "Auction already ended." );
11         require(msg.value > highestBid,
12                     "There already is a higher bid.");
13         if (highestBid != 0) {
14             pendingReturns[highestBidder] = highestBid;
15         }
16         highestBidder = msg.sender;
17         highestBid = msg.value;
18         emit HighestBidIncreased(msg.sender, msg.value);
19     }
20    function withdraw() public returns (bool) {
21         uint amount = pendingReturns[msg.sender];
22         if (amount > 0) {
23             pendingReturns[msg.sender] = 0;
24             if (!msg.sender.send(amount)) {
25                 pendingReturns[msg.sender] = amount;
26                 return false;
27             }
28         }
29         return true;
30     }
31     //...
```

Fig. 1: Auction contract in Solidity

When a client calls the SimpleAuction contract, he needs to be sure that certain properties hold. These properties can be simple enough that you can see them in the code, or they can be more complex to the point where you can't prove them through the code alone. Consider two properties:

(a) "Each new winning bid is superior to the old winning bid"
(b) "I do not lose money"

Property (a) is easy to see in the code as it's the requirement on line 10 of Figure 1. Property (b) on the other hand, while extremely important, is not obvious. A client needs to be sure that he is not going to lose his money due to the *Code is law* element, meaning that anything that happens cannot be changed or undone. So if, for whatever reason, a client does lose money through the SimpleAuction contract, no one can do anything about it. Which is why being sure of a property like "I do not lose money" is crucial. To do that, we propose to model the smart contract using the model checker Cubicle. This requires not only a model of the smart contract itself, but also of the underlying blockchain semantics.

3 Modeling a smart contract

Anything done by a smart contract can be traced back to its blockchain. If highestBidder is modified, that means that somewhere in the blockchain, there is a transaction that called bid with a sufficiently large sum. Not being able to trace an action back to the blockchain implies a problem. Therefore, modeling a smart contract requires an accompanying model of the blockchain. We do this with the help of Cubicle, briefly introduced in the next subsection.

3.1 Cubicle

Cubicle is an SMT-based model checker for parameterized transition systems. For a more in-depth and thorough explanation, we refer the reader to [7, 8]. For the purposes of this work, we will focus on a quick overview of the necessary aspects of Cubicle.

Cubicle input programs represent transition systems described by: (1) a set of type, variable and array declarations; (2) a formula for the initial states; and (3) a set of guarded commands (transitions).

Type, variable and array declarations. Cubicle has several built-in data types, among which are integers (int), booleans (bool), and process identifiers (proc). Additionally, the user can define enumerations. For instance, the code

```
type location = L1 | L2 | L3
var W : location
var X : int
array Z[proc] : bool
```

defines a type location with three constructors (L1, L2, and L3), two global variables W and X of types location and int, respectively, and a proc-indexed array Z. The type proc is a key ingredient here as it is used to parameterize the system: given a process identifier i, the value Z[i] represents somehow the *local* variable Z of i.

Initial states. The content of a system state is fully characterized by the value of its global variables and arrays. The initial states are defined by an **init** formula given as a universal conjunction of literals. For example, the following declaration

```
init (i) { Z[i] = False && W = L1 }
```

should be read as : "initially, for all process i, Z[i] is equal to False and W contains L1". (Note that the content of variable X is unspecified, and can thus contain any value)

Transitions. The execution of a parameterized system is defined by a set of guard/action transitions. It consists of an infinite loop which non-deterministically triggers at each iteration a transition whose guard is true and whose action is to update state variables. Each transition can take one or several process identifiers as arguments. A guard is a conjunction of literals (equations, disequations or inequations) and an action is a set of variable assignments or array updates. For instance, the following transition

```
transition tr_1 (i)
requires { Z[i] = False }
{ W := L2;
  X := 1; }
```

should be read as follows : "if there exists a process i such that Z[i] equals False, then atomically assign W to L2 and X to 1".

Unsafe states. The safety properties to be verified are expressed in their negated form and characterize unsafe states. They are given by existentially-quantified formulas. For instance, the following unsafe formula

```
unsafe (i) { Z[i] = False && X = 1 }
```

should be read as follows : "a state is unsafe if there exists a process i such that Z[i] is equal to False and X equals 1".

Error traces. All of the above allows Cubicle to verify a model. If it finds a way to reach an unsafe state, an error trace is printed, such as the following

```
Error trace: Init -> t2(#1) -> t3(#3) -> unsafe[1]
```

This lets the user check which sequence of transitions led to the unsafe state. A number preceded by # is a process identifier. This means that t2(#1) stands for process 1 activating that transition. If you have multiple unsafe states declared, the index next to **unsafe** lets you know which one was reached.

3.2 Blockchain model

To model the blockchain we first need to model the elements that will constitute transactions seen in the blockchain.

```
type call = Bid | Withdraw | Send | Finish | None

var Cmd : call
var Value : int
var Sender : proc
var Recv : proc
```

The constructors of type call represent calls to smart contract entry points. Bid and Withdraw correspond to functions bid() and withdraw(). Finish corresponds to a function to close the auction (not pictured in Figure 1). Send is used to represent transactions to the sender (*e.g.* line 24 of Figure 1) while None means *absence of transactions*. The elements of a transaction are defined by four variables:

- Cmd, the calls to an entry point;
- Value, the amount of money attached to a transaction;
- Sender, who calls the contract;
- Recv; the receiver, used in the case of Withdraw, where the contract calls a client.

Once the elements of a transaction are declared, the next step is to model the transaction mechanism of the blockchain. For that, we define three transitions to simulate transactions to the three smart contract entry points.

```
transition call_bid(i)
requires { Cmd = None }
{
  Cmd := Bid;
  Value := Rand.Int();
  Sender := i;
}

transition call_withdraw(i)
requires { Cmd = None }
{
  Cmd := Withdraw;
  Sender := i;
}

transition call_finish(i)
requires { Cmd = None }
{
  Cmd := Finish;
  Sender := i;
}
```

Each transaction has a parameter i which represents the client who called the corresponding entry point (the sender in Solidity). The only requirement indicates that the contract can't be doing something else simultaneously (Cmd = None). The effects of these transitions are simple: Cmd is set to the corresponding constructor (Bid, Withdraw, or Finish, respectively) and the variable Sender is

assigned to i. In call_bid, the variable Value is set to a (positive) random integer corresponding to the amount bid by i.
Once the blockchain has been modeled, we can move on to modeling the contract itself.

3.3 Smart contract model

To model the actual contract, we need to model its variables and its functions.

```
var Bidder : proc
var Bidding : int
var End_auction : bool
var Owner : proc
array PR[proc] : int
```

Bidder and Bidding correspond to highestBidder and highestBid, while End_auction corresponds to ended. The variable Owner is the person who started the auction, meaning created the contract. The array PR stands for the pendingReturns map in Figure 1. It is worth noting that both variables are finite but unbounded data structures.
The functions bid and withdraw are modeled as Cubicle transitions. These transitions serve as entry points for our contract.

```
transition bid(i)
requires { End_auction = False &&
           Cmd = Bid && i = Sender && i <> Bidder &&
           PR[i] = 0 && Value > Bidding }
{
  Bidding := Value;
  Bidder := i;
  PR[Bidder] := Bidding;
  Cmd := None;
}

transition withdraw(i)
requires { Cmd = Withdraw && i = Sender && PR[i] > 0 }
{ PR[i] := 0;
  Cmd := Send;
  Value := PR[i];
  Recv := i;
}
```

Transition bid is called by one process, i, who has to be the Sender, but not the current Bidder. The other requirements should be read as follows:

End_auction = False: the auction is open
Cmd = Bid: the transaction in the blockchain is Bid
PR[i] = 0: the new bidder hasn't previously bid
Value > Bidding: the new bid is bigger than the old winning bid

The effects are simple, Bidder and Bidding are set to the new values, PR for the old winner who has now been outbid is set to his old bid value, and Cmd is reset to None to indicate that the contract is no longer occupied.
Similarly, the requirements of transition Withdraw are the following:

Cmd = Withdraw: the transaction in the blockchain is a call to withdraw
i = Sender: the process i is the one that called the function
PR[i] $>$ 0: the person previously bid and was outbid by someone

The effects of transition withdraw are slightly different since withdraw goes on to send money to whoever called the method. The receiver is now set to i (Recv := i), and the value that will accompany the transaction is set to the amount of money to be returned (Value := PR[i]). The pending return PR[i] is reset globally for client i (PR[i] = 0) and Cmd is set to Send, to indicate that the contract is calling the client's method. The transition which Send corresponds to can be seen below:

```
transition value(i)
requires { Cmd = Send && Recv = i }
{ Cmd := None; }
```

This transition checks that Send was in fact called (Cmd = Send), as well as the fact that the receiver is the currently active process (Recv = i). It then resets Cmd to None to free the contract.

4 Defining and Verifying Properties

Recall that we want to be sure of certain properties:

(a) "Each new winning bid is superior to the old winning bid"
(b) "I do not lose money"

Once defined in an informal manner, the properties need to be converted into safety properties. This is not always straightforward and might require additional information. This is done via a two-step process consisting of (i) defining extra logical formulas (*ghost* variables) and (ii) instrumenting the model with these formulas.

4.1 Ghost variables and Model instrumentation

Ghost variables, introduced below, will neither appear in the original Solidity contract, nor will they impact the Cubicle model outside of property verification.

```
array Out[proc] : int
array In[proc] : int
var Old_Bidding : int
```

The variables Out and In are for property (b). In is an array storing how much each client (aka process) bids, and Out stores how much they get back if/when they call withdraw. Old_Bidding tracks the old highest bid for property (a). The code below is the instrumented model. Transition withdraw has been omitted since it is not instrumented.

```
transition bid(i)
requires { End_auction = False &&
           Cmd = Bid && i = Sender && i <> Bidder &&
           PR[i] = 0 && Value > Bidding}
{
  Bidding := Value;
  Bidder := i;
  PR[Bidder] := Bidding;
  Cmd := None;
  Old_Bidding := Bidding;
  In[i] := In[i] + Value;
}

transition value(i)
requires { Cmd = Send && Recv = i }
{ Cmd := None;
  Out[i] := Out[i] + Value;
}
```

The *ghost* variables appear only in the action parts of the transitions. The bid transition utilizes both Old_Bidding and In. It updates Bidding to set a new highest bid value. To keep track of what the old value was, Old_Bidding is set to Bidding's value. In is updated for the new bidder with their bid value. The transition value is instrumented instead of withdraw, since the most important action, that is giving the client back their money, happens during the value transition. It uses Out to keep track of how much money has been returned.

The ghost variables are also part of the initial state declaration.

```
init (i) { End_auction = False && Bidding = 0 && Cmd = None
           && PR[i] = 0 && In[i] = 0 && Out[i] = 0
           && Old_Bidding = 0 }
```

That is to say, the auction hasn't ended, there is no winning bid, the contract isn't doing anything, and no one has bid and subsequently withdrawn money.

4.2 Defining properties

Once the code is instrumented, we can introduce the safety properties we want Cubicle to check.

Property (a): New bids are higher.

The first property is "*Each new winning bid is superior to the old winning bid*". This property can be easily defined by the following unsafe formula which uses only the ghost variables Old_Bidding and Bidding.

```
unsafe () { Old_Bidding > Bidding }
```

Checking property (a) with the above formula simply means declaring Old_bidding being superior to Bidding as unsafe, but only if the model was correctly instrumented with these variables.

Property (b) : Do I lose money ?

Defining this property is less obvious. While ghost variables have been introduced to keep track of money exchanges between users and the contract, another problem is the lack of precision of the sentence. When should we check that a user did not lose money? At the end of the auction? If so, when do we consider the auction to *really* be over?

We will make these issues more concrete in the next section. In particular, we shall explain how we arrived at the following formulation of property (b):

```
unsafe (i) { End_auction = True && i <> Bidder && PR[i] = 0
             && Cmd = None && Out[i] < In[i] }
```

5 Interpreting Cubicle Error Traces

As stated previously, the tricky property is "*I do not lose money*". The logical implication of "I do not lose money" is that if the auction is over, (End_auction = True), then your Out isn't less than your In.

```
unsafe (i) { End_auction = True && Out[i] < In[i] }
```

Except Cubicle prints the following error trace:

```
Error trace: Init -> call_bid(#1) -> bid(#1) ->
             call_finish(#1) -> finish_auction() -> unsafe
UNSAFE !
```

Upon further inspection, it becomes obvious why this state is reached. This is true for every client, even the winner, who technically does *lose* money, so to speak. We modify our unsafe state to the following by adding that the process cannot be the winner (Bidder <> i).

```
unsafe (i) { End_auction = True && i <> Bidder &&
             Out[i] < In[i] }
```

However, Cubicle still says

```
Error trace: Init -> call_bid(#1) -> bid(#1) ->
             call_bid(#2) ->  bid(#2) ->
             call_finish(#1) -> finish_auction() -> unsafe
UNSAFE !
```

as what's missing is checking whether or not a client withdrew their bid. We incorporate that check below.

```
unsafe (i) { End_auction = True && i <> Bidder &&
             PR[i] = 0 && Out[i] < In[i] }
```

but Cubicle can still reach that state:

```
Error trace: Init -> call_bid(#1) -> bid(#1) ->
             call_bid(#2) -> bid(#2) ->
             call_finish(#1) -> finish_auction() ->
             call_withdraw(#1) -> withdraw(#1) -> unsafe
UNSAFE !
```

Once the smart contract has completely finished every action associated with a function (i.e. transition), it resets Cmd to None, which we haven't checked for. We add that to our unsafe state.

```
unsafe (i) { End_auction = True && i <> Bidder && PR[i] = 0
             && Cmd = None && Out[i] < In[i] }
```

The above is the correct implementation of "I do not lose money." This time Cubicle replies **Safe**

But error traces aren't necessarily the result of an incorrectly written unsafe state. Looking back at Figure 1, there is an equals sign on line 12. This essentially implies that you cannot bid multiple times in a row without calling withdraw between every bid. This is why PR[i] = 0 needs to be in the requirements of transition bid. Removing this literal (or putting it in a comment) like in the transition below

```
transition bid(i)
requires { End_auction = False &&
           Cmd = Bid && i = Sender && i <> Bidder
           (*&& PR[i] = 0*) && Value > Bidding}
...
```

makes Cubicle reach the correctly-implemented "I do not lose money" unsafe state with the following trace:

```
Unsafe trace: call_bid(#1) -> bid(#1) -> call_bid(#3) ->
              bid(#3) -> call_bid(#1) -> bid(#1) ->
              call_bid(#2) -> bid(#2) -> call_finish(#2) ->
              finish_auction() -> call_withdraw(#1)->
              withdraw(#1) -> value(#1) -> unsafe
UNSAFE !
```

The scenario represented by this trace is not immediately evident as it brings into play 3 clients (processes) and 13 transitions.

6 Conclusion & Future Work

In this paper we proposed a two-layer framework for smart contract verification with the model checker Cubicle. This method implements a model of the smart contract itself and the blockchain transaction mechanism behind it. Our method introduces a way of verifying various types of functional properties linked to a smart contract as Cubicle safety properties. Since this is done through ghost variables and model instrumentation, it has no impact on the original smart contract code, meaning it is independent of any particular smart contract language, and is therefore generalizable and usable for multiple smart contract languages. We also describe a way of interpreting potential error traces generated by Cubicle, and how they can aid in the development of a smart contract. An immediate line of future work is to automate this stepwise process. We need to define an abstract high-level language to express the properties to be checked by Cubicle. From this language, the ghost variables will be automatically generated to instrument the Cubicle code. Furthermore, we would also like to consider automatic translation of Solidity or Michelson code to Cubicle.

References

1. Mi-cho-coq: Formalisation of the michelson language using the coq proof assistant. `https://gitlab.com/nomadic-labs/mi-cho-coq`.
2. Solidity Common Patterns. `https://solidity.readthedocs.io/en/v0.5.10/common-patterns.html#state-machine`.
3. State of DApps website. `https://www.stateofthedapps.com/stats`.
4. S. Amani, M. Bégel, M. Bortin, and M. Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77. ACM, 2018.
5. K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. 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. ACM, 2016.
6. G. Bigi, A. Bracciali, G. Meacci, and E. Tuosto. Validation of decentralised smart contracts through game theory and formal methods. In *Programming Languages with Applications to Biology and Security*, pages 142–161. Springer, 2015.
7. S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems: Tool paper. In *CAV*, CAV'12, pages 718–724, Berlin, Heidelberg, 2012. Springer-Verlag.
8. S. Conchon, A. Mebsout, and F. Zaïdi. Vérification de systèmes paramétrés avec Cubicle. In *JFLA*, Aussois, France, Feb. 2013.
9. A. Das, S. Balzer, J. Hoffmann, and F. Pfenning. Resource-aware session types for digital contracts. *arXiv preprint arXiv:1902.06056*, 2019.
10. A. Mavridou and A. Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. *arXiv preprint arXiv:1711.09327*, 2017.
11. Z. Nehai and F. Bobot. Deductive proof of ethereum smart contracts using why3. *arXiv preprint arXiv:1904.11281*, 2019.
12. Z. Nehai, P.-Y. Piriou, and F. Daumas. Model-checking of smart contracts. In *IEEE International Conference on Blockchain*, pages 980–987, 2018.
13. C. Reitwiessner. Formal verification for solidity contracts.
14. I. Sergey and A. Hobor. A concurrent perspective on smart contracts. In *International Conference on Financial Cryptography and Data Security*, pages 478–493. Springer, 2017.

Smart Contract Interactions in Coq

Jakob Botsch Nielsen and Bas Spitters

Concordium Blockchain Research Center, Computer Science, Aarhus University

Abstract. We present a model/executable specification of smart contract execution in Coq. Our formalization allows for inter-contract communication and generalizes existing work by allowing modelling of both depth-first execution blockchains (like Ethereum) and breadth-first execution blockchains (like Tezos). We represent smart contracts programs in Coq's functional language Gallina, enabling easier reasoning about functional correctness of concrete contracts than other approaches. In particular we develop a Congress contract in this style. This contract – a simplified version of the infamous DAO – is interesting because of its very dynamic communication pattern with other contracts. We give a high-level partial specification of the Congress's behavior, related to reentrancy, and prove that the Congress satisfies it for all possible smart contract execution orders.

1 Introduction

Since Ethereum, blockchains make a clear separation between the consensus layer and the execution of smart contracts. In Ethereum's Solidity language contracts can arbitrarily call into other contracts as regular function calls. Modern blockchains further separate the top layer in an execution layer and a contract layer. The execution layer schedules the calls between the contracts and the contract layer executes individual programs. The choice of execution order differs between blockchains. For example, in Ethereum the execution is done in a synchronous (or depth first) order: a call completes fully before the parent continues, and the parent is able to observe its result. Tezos uses the breadth first order.

We provide[1] a model/executable specification of the execution and contract layer of a third generation blockchain in the Coq proof assistant. We use Coq's embedded functional language Gallina to model contracts and the execution layer. This language can be extracted to certified programs in for example Haskell or Ocaml. Coq's expressive logic also allows us to write concise proofs. The consensus protocol provides a consistent global state which we treat abstractly in our formalization.

We work with an account-based model. We could also model the UTxO model by converting a list of UTxO transactions to a list of account transactions [Zah18]. Like that work, we do not model the cryptographic aspects, only the accounting aspects: the transactions and contract calls.

[1] `https://gitlab.au.dk/concordium/smart-contract-interactions/tree/v1.0`

This text is organized as follows: Section 2 describes the implementation of the execution layer in Coq. In Section 3 we provide a simple principled specification for the Congress. By using such specifications one avoids having to deal with reentrancy bugs in a post-hoc way. Section 4 discusses related work. Section 5 concludes.

2 Implementation

2.1 Basic assumptions

Our goal is to model a realistic blockchain with smart contracts. To do so we will require this blockchain to supply some basic operations that are to be used both by smart contracts and when specifying our semantics. Our most basic assumptions are captured as a typeclass:

```
Class ChainBase :=
  { Address : Type;
    address_countable :> Countable Address;
    address_is_contract : Address → bool;
    ... }.
```

Specifically we require a countable `Address` type with a clear separation between addresses belonging to contracts and to users. While this separation is not provided in Ethereum its omission has led to exploits before[2] and we thus view it as realistic that future blockchains allow this. Other blockchains commonly provide this by using some specific format for contract addresses, for example, Bitcoin marks addresses with associated scripts using so-called pay-to-script-hash addresses which always start with 3.

Generally all semantics and smart contracts will be abstracted over an instance of this type, so in the following sections we will assume we are given such an instance.

2.2 Smart Contracts

We will consider a functional smart contract language. Instead of modelling the language as an abstract syntax tree in Coq, as in [AS19], we model individual smart contracts as records with (Coq) functions.

Local state. It is not immediately clear how to represent smart contracts by functions. For one, smart contracts have local state that they should be able to access and update during execution. In Solidity, the language typically used in Ethereum, this state is mutable and can be changed at any point in time. It is possible to accomplish something similar in pure languages, for example by using a state monad which allows state to be updated at any point during a

[2] See for instance `https://www.reddit.com/r/ethereum/comments/916xni/how_to_pwn_fomo3d_a_beginners_guide/`

function's execution, but we do not take this approach. Instead we use a more traditional functional approach where the contract takes as input its current state, and returns a single new, updated state.

However, different contracts will typically have different types of states. A crowdfunding contract may wish to store a map of backers in its state while an auction contract would store information about ongoing auctions. To facilitate this polymorphism we use an intermediate storage type called `SerializedValue`. We define conversions between `SerializedValue` and primitive types like booleans and integers plus derived types like pairs, sums and lists. Generally this allows conversion from and to `SerializedValue` to be handled implicitly and mostly transparently to the user.

Inter-contract communication. In addition to local state we also need some way to handle inter-contract communication. In Solidity contracts can arbitrarily call into other contracts as regular function calls. This would once again be possible with a monadic style, for example by the use of a promise monad where the contract would ask to be resumed after a call to another contract had finished. To ease reasoning we choose a simpler approach where contracts return actions that indicate how they would like to interact with the blockchain, allowing transfers, contract calls and contract deployments only at the end of execution. The blockchain will then be responsible for scheduling these actions in its execution layer.

With this design we get a clear separation between contracts and their interaction with the chain. That such separations are important has been realized before, for instance in the design of Michelson and Scilla [SKH18a]. Indeed, a "tail-call" approach like this forces the programmer to update the contract's internal state before making calls to other contracts, mitigating by construction reentrancy issues such as the infamous DAO exploit.

Thus, contracts will take their local state and some data allowing them to query the blockchain. As a result they then optionally return the new state and some operations (such as calls to other contract) allowing inter-contract communication. Overall, this design is very similar to the Tezos blockchain where contracts are written in Michelson which follows a similar approach.

The Ethereum model may be compared to object-oriented programming. Our model is similar to the actor model, as contracts do not read or write the state of another contract directly, but instead communicate via messages. One finds similar models in Liquidity and in Scilla, which is based on IO-automata.

The contract's view. Smart contracts are typically allowed to query various data about the blockchain during execution, such as the current block height. Normally this is provided as special instructions. For instance, this is the case in EVM bytecode used for Ethereum. Since we use a shallow embedding we will instead pass this as an additional argument to the contract. In our framework, we give contracts the following view of the blockchain:

```
Definition Amount := Z.
```

```
Record Chain :=
  { chain_height : nat;
    current_slot : nat;
    finalized_height : nat;
    account_balance : Address → Amount; }.
```

We allow contracts to access basic details about the blockchain, like the current chain height, slot number and the finalized height. The slot number is meant to be used to track the progression of time; in each slot, a block can be created, but it does not have to be. The finalized height allows contracts to track the current status of the finalization layer available in for example the Concordium blockchain [MMNT19]. This height is different from the chain height in that it guarantees that blocks before it will not be changed. We finally also allow the contract to access balances of accounts, as is common from other blockchains.

The contract. The final piece of information provided to contracts when they are executed is information about the call. Overall, we thus represent contracts using the following types:

```
Record ContractCallContext :=
  { ctx_from : Address;
    ctx_contract_address : Address;
    ctx_amount : Amount; }.
Inductive ActionBody :=
  | act_transfer (to : Address) (amount : Amount)
  | act_call (to : Address) (amount : Amount)
             (msg : SerializedValue)
  | act_deploy (amount : Amount) (c : WeakContract)
               (setup : SerializedValue)
with WeakContract :=
  | build_weak_contract
      (init : Chain → ContractCallContext →
         SerializedValue (* setup *) →
         option SerializedValue)
      (receive : Chain → ContractCallContext →
         SerializedValue (* state *) →
         option SerializedValue (* message *) →
         option (SerializedValue * list ActionBody)).
```

Here the `ContractCallContext` type represents information that is common to when the contract executed due to deployment or due to receiving a message. It contains the source address (`ctx_from`), the contract's own address (`ctx_contract_address`) and the amount of money transferred (`ctx_amount`). The `ActionBody` type represents operations that interact with the chain. It allows for simple messageless transfers (`act_transfer`), calls with messages (`act_call`), and deployment of new contracts (`act_deploy`). These do not contain a source address to model that while contracts can interact with the blockchain, they do not get to specify the source (which is their own address) when they do so. Finally, a contract is two functions. The `init` function is used when a contract is deployed to set up its initial state, while the `receive` function will be used for transfers and calls with messages afterwards. They both return option types, allowing the contract to signal invalid calls or deployments. The `receive` function additionally returns a list of `ActionBody` that it wants to be performed in the chain after, as we described above. Later, we will also use a representation where there *is* a source address; we call this type `Action`:

```
Record Action :=
  { act_from : Address;
    act_body : ActionBody; }.
```

This type might resemble what is normally called a transaction, but we make a distinction between the two. An `Action` is an unevaluated operation that, when executed by an implementation, affects the blockchain's state. Particularly, compared to a transaction it is underrepresented in that `act_deploy` does not contain the address of the contract to be deployed. This models that it is the implementation that picks the address of a newly deployed contract, not the contract making the deployment. We will later describe our `ActionEvaluation` type which captures more in depth the choices made by the implementation while executing an action.

The functions of contracts may seem peculiar in that they are typed using `SerializedValue` parameters. This is also the reason for the name `WeakContract`. Generally this makes specifying semantics simpler, since the semantics can deal with contracts in a generic way. However, for users of the framework writing concrete contracts this form of "string-typing" makes things harder. For this reason we provide a dual notion of a *strong* contract, which is a polymorphic version of contracts generalized over the setup, state and message types. Users of the framework will only need to be aware of this notion of contract, which does not contain references to `SerializedValue` at all.

One could also imagine an alternative representation using a dependent record of setup, state and message types plus functions over those types. However, such a representation makes it nearly impossible for contracts to interact with other contracts since they will somehow need to prove that the messages they are sending are of the types stored in this record. In particular this is difficult when the blockchain has no knowledge about individual contracts and only works generically with them.

2.3 Semantics

Next we wish to specify the semantics of the execution layer.

Environments. The `Chain` type given above is merely the contract's view of the blockchain and does not store enough information to allow the blockchain to run actions. More specifically we need to be able to look up information about currently deployed contracts like their functions and state. We augment the `Chain` type with this information and call it an `Environment`:

```
Record Environment :=
  { env_chain :> Chain;
    env_contracts : Address → option WeakContract;
    env_contract_states :
      Address → option SerializedValue; }.
```

It is not hard to define functions that allow us to make updates to environments. For instance, inserting a new contract is done by creating a new function that checks if the address matches and otherwise uses the old map. In other words we use simple linear maps in the semantics. In similar ways we can update the rest of the fields of the `Environment` record.

Evaluation of actions. When contracts return actions the execution layer will somehow need to evaluate the effects of these actions. We define this as a "proof-relevant" relation `ActionEvaluation` in Coq:

```
ActionEvaluation : Environment → Action →
  Environment → list Action → Type
```

This relation captures the requirements and effects of executing the action in the environment. It is "proof-relevant", meaning that it can be inspected, which is useful since actions by themselves are underspecified. For example, a contract can return an action that deploys a new contract; in this case we leave it up to the implementation to pick an appropriate address for the new contract. However, when reasoning about action evaluation it is useful to know which address a contract was deployed to and this information can be retrieved by inspecting the evaluation.

We define the relation by three cases: one for transfers of money, one for deployment of new contracts, and one for calls to existing contracts. To exemplify this relation we give its formal details for the simple transfer case below:

```
| eval_transfer :
    forall {pre : Environment}
           {act : Action}
           {new_env : Environment}
           (from to : Address)
           (amount : Amount),
      amount ≤ account_balance pre from →
      address_is_contract to = false →
      act_from act = from →
      act_body act = act_transfer to amount →
      EnvironmentEquiv
        new_env
        (transfer_balance from to amount pre) →
      ActionEvaluation pre act new_env []
```

In this case the sender must have enough money and the recipient cannot be a contract. When this is the case a transfer action and the old environment evaluate to the new environment where the `account_balance` has been updated appropriately. Finally, such a transfer does not result in more actions to execute since it is not associated with execution of contracts. Note that we close the evaluation relation under extensional equality (`EnvironmentEquiv`).

We denote this relation by the notation $\langle \sigma, a \rangle \Downarrow (\sigma', l)$. The intuitive understanding of this notation is that evaluating the action a in environment σ results in a new environment σ' and new actions to execute l.

Chain traces. The `Environment` type captures enough information to evaluate actions. We further augment this type to keep track of the queue of actions to execute. In languages like Simplicity [O'C17] this data is encoded implicitly in the call stack, but since interactions with the blockchain are explicit in our framework we keep track of it explicitly in the `ChainState` type.

```
Record ChainState :=
  { chain_state_env :> Environment;
    chain_state_queue : list Action; }.
```

We are now ready to define what it means for the chain to take a step. Formally, this is defined as a "proof-relevant" relation `ChainStep`:

```
ChainStep : ChainState → ChainState → Type
```

We denote this relation with the notation $(\sigma, l) \to (\sigma', l')$, meaning that we can step from the environment σ and list of actions l to the environment σ' and list of actions l'. We give this relation as simplified inference rules below.

$$\textsc{step-block}\ \frac{b \text{ valid in } \sigma \qquad \textit{acts} \text{ from users}}{(\sigma, \texttt{[]}) \to (\texttt{add\_block}\ b\ \sigma, \textit{acts})} \qquad \textsc{step-action}\ \frac{\langle \sigma, a \rangle \Downarrow (\sigma', l)}{(\sigma, a :: l') \to (\sigma', l \mathbin{++} l')} \qquad \textsc{step-permute}\ \frac{\text{Perm}(l, l')}{(\sigma, l) \to (\sigma, l')}$$

The STEP-BLOCK rule allows the addition of a new block with associated actions. This is the only way to add new actions into a trace when the queue is empty. We require that the block information (b in the rule) is valid in the current environment (the b valid in σ premise), meaning that it needs to satisfy some well-formedness conditions. For example, if the chain currently has height n, the next block added needs to have height $n + 1$. There are other well-formedness conditions on other fields, such as the finalized height, but we omit them here for brevity. Another condition is that all added actions must come from users (the *acts* from users premise). This models the real world where transactions added in blocks are "root transactions" from users, and carrying out these transactions might cause contracts to generate new transactions. In our model this condition is crucial to ensure that transfers from contracts can happen only due to execution of their associated code. When the premises are met we update information about the current block (such as the current height and the balance of the creator, signified by the `add_block` function) and update that the queue now contains the actions that were added.

The STEP-ACTION rule allows the evaluation of the action in the beginning of the queue, replacing it with the resulting new actions to execute. This new list (l in the rule) is concatenated at the beginning, corresponding to using the queue as a stack. This results in a depth-first execution order of actions. The STEP-PERMUTE rule allows an implementation to use a different order of reduction by permuting the queue at any time. For example, it is possible to obtain a breadth-first order of execution by permuting the queue so that newly added events are in the back. In this case the queue will be used like an actual FIFO queue.

Building upon steps we can further define *traces* as the proof-relevant reflexive transitive closure of the step relation. In other words, this is a sequence of steps where each step starts in the state that the previous step ended in. Intuitively the existence of a trace between two states means that there is a semantically correct way to go between those states. If we let ε denote the empty environment with no queue this allows us to define a concept of *reachability*. Formally we say a state (σ, l) is *reachable* if there exists a trace starting in ε and ending in (σ, l). In Coq we define this as

```
Definition reachable (state : ChainState) : Prop :=
  inhabited (ChainTrace empty_state state).
```

Generally, only reachable states are interesting to consider and most proofs are by induction over the trace to a reachable state.

2.4 Building blockchains

We connect our semantics to an executable definition of a blockchain with a typeclass in Coq:

```
Class ChainBuilderType := {
    builder_type : Type;
    builder_initial : builder_type;
    builder_env : builder_type → Environment;
    builder_add_block
      (builder : builder_type)
      (header : BlockHeader)
      (actions : list Action) :
      option builder_type;
    builder_trace (builder : builder_type) :
      ChainTrace empty_state
      (build_chain_state (builder_env builder) []);}.
```

A chain builder is a dependent record consisting of an implementation type (`builder_type`) and several fields using this type. It must provide an initial builder, which typically would be an empty chain, or a chain containing just a genesis block. It must also be convertible to an environment allowing to query various information about the state. Furthermore, it must define a function that allows addition of new blocks. Finally, the implementation needs to be able to give a trace showing that the current environment is reachable with no more actions left in the queue to execute. This trace captures a definition of soundness, since it means that the state of such a chain builder will always be reachable[3].

Instantiations. A priori it is not a guarantee that the semantics we have defined are reasonable. More formally it is possible that `ChainBuilderType` is uninhabited which makes proving properties based on it uninteresting. Thus, as a sanity check, we implement two instances of this typeclass. Both of our implementations are based on finite maps from the std++ library used by Iris [JKJ+18] and are thus relatively efficient compared to the linear maps used to specify the semantics. The difference in the implementations lies in their execution model: one implementation uses a depth-first execution order, while the other uses a breadth-first execution order. The former execution model is similar to the EVM while the latter is similar to Tezos.

These implementations are useful as sanity checks but they also serve other useful purposes in the framework. Since they are executable they can be used to test concrete contracts that have been written in Coq. This involves writing the contracts and executing them using Coq's `Compute` vernacular. In addition, they can also be used to give counter-examples to properties. In the next section we will introduce the *Congress* contract, and we have used the depth-first implementation of our semantics to formally show that this contract with a small change is vulnerable to reentrancy.

3 Case: Congress – a simplified DAO

In this section we will present a case study of implementing and partially specifying a complex contract in our framework.

3.1 The Congress contract

Wang [Wan18] gives a list of eight interesting Ethereum contracts. One of these is the so-called *Congress* in which members of the contract vote on *proposals*. Proposals contain transactions that, if the proposal succeeds, are sent out by the Congress. These transactions are typically monetary amounts sent out to some address, but they can also be arbitrary calls to any other contract.

We pick the Congress contract because of its complex dynamic interaction pattern with the blockchain and because of its similarity to the infamous DAO contract that was deployed on the Ethereum blockchain and which was eventually hacked by a clever attacker exploiting reentrancy in the EVM.

The Congress can be seen as the core of the DAO contract, with the DAO implementing various additional mechanisms on top of voting for proposals. For example, proposals can be seen as investments into other projects, and the DAO contract kept track of the voters on each proposal to be able to pay back rewards to these people in case the project turned out successful.

[3] We do not currently include a notion of completeness. For instance, it is possible to define a trivial chain builder that just ignores the blocks and actions to be added.

We implement the logic of the Congress in roughly 150 lines of Gallina code. The type of messages accepted by the Congress can be thought of as its interface since this is how actors on the blockchain can interact with it. For the Congress we define the following messages:

```
Inductive Msg :=
  | transfer_ownership : Address → Msg
  | change_rules : Rules → Msg
  | add_member : Address → Msg
  | remove_member : Address → Msg
  | create_proposal : list CongressAction → Msg
  | vote_for_proposal : ProposalId → Msg
  | vote_against_proposal : ProposalId → Msg
  | retract_vote : ProposalId → Msg
  | finish_proposal : ProposalId → Msg.
```

The Congress has an owner who is responsible for managing the rules of the congress and the member list. By default, we set this to be the creator of the congress. The owner can transfer his ownership away with the `transfer_ownership` message. For example, it is possible to make the Congress its own owner, in which case all rule changes and modifications to the member list must happen through proposals (essentially making the Congress a democracy).

Anyone can use the `create_proposal` and `finish_proposal` messages. We allow proposals to contain any number of actions to send out, though we restrict the actions to only transfers and contract calls (i.e. no contract deployments). This restriction is necessary because this would require the state of the Congress to contain the contracts to deploy. Since contracts are functions in our shallow embedding this would require storing higher order state which we do not allow in the framework. This is a downside to the shallow embedding – with a deep embedding like [AS19], the code could be stored as an AST or bytes.

While proposals can be finished by anyone they must first have been debated for some period specified in the rules of the congress. During this period, members of the congress have the ability to vote for or against the proposal. After the debating period is over the proposal can be finished and the Congress will remove it from its internal storage and send out its actions in case it passed. The conditions for passing are once again specified in the rules, which contain values such as the margin of yes-votes required.

3.2 A partial specification

The vulnerability of the DAO was in reward payout code in which a specially crafted contract could reenter the DAO causing it to perform actions an unintended number of times. Specifically, the attacker was able to propose a so-called *split* and have the original DAO transfer a disproportionate amount of money to a new DAO contract under his control. The Congress does not contain similar code, but the same kind of bug would be possible in code responsible for carrying out proposals.

Previous research has focused on defining this kind of reentrancy formally which we could also define and prove in our framework. Such (hyper-)properties are interesting, but they also rely heavily on the benefit of hindsight and their statements are complex and hard to understand. Instead we would like to come up with a natural specification pertaining to the Congress that a programmer could reasonably have come up with, even without knowledge of reentrancy. Our goal with this is to apply the framework in a very concrete setting.

The specification we give is based on the following observation: any transaction sent out by the congress should correspond to an action that was previously created with a `create_proposal` message. This is a temporal property because it says something about the past whenever an outgoing transaction is observed. Temporal logic is not natively supported by Coq, so this would require some work. Therefore we prefer a similar but simpler property: the number of actions in previous `create_proposal` messages is always greater than or equal to the total number of transactions the congress has sent out. This is not a full specification of the behavior of the Congress but proving this property can help increase trust that the congress is not vulnerable to reentrancy. With such a proof, any bug exploiting the Congress in a similar way to the DAO would somehow require a new proposal to be created for each time the exploit was carried out. In particular, the result would not have been provable in the original DAO contract because of the reentrancy exploit. Our main result about the congress is a formal proof that this always holds after adding a block:

```
Corollary congress_txs_after_block
          {ChainBuilder : ChainBuilderType}
          prev creator header acts new :
  builder_add_block prev creator header acts = Some new →
  forall addr,
    env_contracts new addr =
    Some (Congress.contract : WeakContract) →
    length (outgoing_txs (builder_trace new) addr) ≤
    num_acts_created_in_proposals
      (incoming_txs (builder_trace new) addr).
```

This result states that, after adding a block, any address at which a Congress contract is deployed satisfies the property previously described. Here the function `num_acts_created_in_proposals` looks at all previous `create_proposal` messages and sums the number of actions in them. The `incoming_txs` and `outgoing_txs` functions are general functions that finds transactions (evaluation of actions) in a trace. In this sense the property treats the contract as a black box, stating only things about the transactions that has been observed on the blockchain.

We prove this property by generalizing it and proving something stronger. Specifically, instead of stating the invariant over just the transactions and proposals we state it over the following data:

– The internal state of the contract; more specifically, the current number of actions in proposals stored in the internal state.

– The number of transactions sent out by the Congress, as before.

- The number of actions *in the queue* where the Congress is the source.
- The number of actions created in proposals, as before.

The key observations being that

1. When a proposal is created, the number of actions created in proposals goes up, but so does the number of actions in the internal state of the Congress.
2. When a proposal is finished, the number of actions in the internal state goes down, but the number of actions in the queue goes up accordingly (assuming the proposal was voted for). In other words, actions "move" from the Congress's internal state to the queue.
3. When an outgoing transaction appears on the chain it is because an action moved out of the queue.

Especially observation 3 is interesting. It allows us to connect the evaluation of a contract in the past to its resulting transactions on the chain, even though these steps can be separated by many unrelated steps in the trace.

The proof of the stronger statement is straightforward by inducting over the trace and showing that it always holds. When deploying the Congress we need to establish the invariant which boils down to proving functional correctness of the `init` function and the usage of some results that hold for contracts which have just been deployed (for instance, such contracts have not made any outgoing transactions). On calls to the Congress the invariant needs to be reestablished, which boils down to proving functional correctness of the `receive` function. Crucially, we can reestablish the invariant because the implementation of the Congress clears out proposals from its state *before* the actions in the proposal are evaluated (the DAO was vulnerable because it neglected to do this on splits). Once we have established this stronger statement the result easily follows as a direct corollary.

4 Related work

Both Simplicity [O'C17] and Scilla [SKH18a] are smart contract languages with an embedding in Coq. Temporal properties of several smart contracts have been verified in Scilla [SKH18b], although our congress contract is more complex than the contracts described in that paper. We are unaware of an implementation of such a contract in Scilla. Scilla, as an intermediate language which includes both a functional part and contract calls, uses a CPS translation to ensure that every call to another contract is done as the last instruction. In our model, the high-level language and the execution layer are strictly separated.

The formalization of the EVM in F* [GMS18] can be extracted and used to run EVM tests to show that it is a faithful model of the EVM. However, they do not prove properties of any concrete contracts. Instead they consider classes of bugs in smart contracts and try to define general properties that prevent these. One of these properties, call integrity, is motivated by the DAO and attempts to capture reentrancy. Intuitively a contract satisfies call integrity if the calls it makes cannot be affected by code of other contracts. VerX [PDT$^+$19] uses temporal logic and model checking to check a similar property. Such statements are not hard to state in our framework given Coq's expressive logic, and it seems this would be an appropriate property to verify for the Congress. Unfortunately even a correct Congress does not satisfy this property, since it is possible for called contracts to finish proposals which can cause the Congress to perform calls. This property could potentially be proven in a version of the Congress that only allowed proposals to be finished by humans, and not by contracts.

5 Conclusion and future work

We have formalized the execution model of blockchains in Coq and used our formalization to prove formally a result about a concrete contract. Our formalization of blockchain semantics is flexible in that it accounts both for depth-first and breadth-first execution order, generalizing existing blockchains and previous work, while remaining expressive enough to allow us to prove results about complex contracts. We showed for a Congress – a simplified version of the DAO, which still has a complex dynamic interaction pattern – that it will never send out more transactions than have been created in proposals. This is a natural property that aids in increasing trust that this contract is not vulnerable to reentrancy like the DAO.

More smart contracts are available in Wang's PhD thesis [Wan18] and specifying these to gain experience with using the framework will help uncover how the framework itself should be improved. In this area it is also interesting to consider more automatic methods to make proving more productive. For example, temporal logics like LTL or CTL can be useful to specify properties on traces and model checking these can be automated; see e.g. [PDT+19].

Finally, while our current framework is inspired by and generalizes existing blockchains, there is still more work to be done to get closer to practical implementations. Gas is notoriously difficult to deal with in our shallow embedding because tracking costs of operations can not be done automatically, but monadic approaches have been used for similar purposes before [MFN+18]. To deal with this problem we plan to connect our shallow embedding with a deep embedding of the language Oak as described in [AS19], which will also allow proving properties about Oak contracts in our framework. In the other direction it is interesting to consider extraction of our contracts into other languages like Liquidity, Oak or Solidity. This is more directly applicable to current practice.

Acknowledgements We would like to thank the Oak team for stimulating discussions.

References

[AS19] Danil Annenkov and Bas Spitters. Deep and shallow embeddings in Coq. *TYPES*, 2019.

[GMS18] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *PoST*, pages 243–269. Springer, 2018.

[JKJ⁺18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[MFN⁺18] Jay McCarthy, Burke Fetscher, Max S New, Daniel Feltey, and Robert Bruce Findler. A Coq library for internal verification of running-times. *Science of Computer Programming*, 164:49–65, 2018.

[MMNT19] Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. Afgjort – a semi-synchronous finality layer for blockchains. Cryptology ePrint 2019/504, 2019. `https://eprint.iacr.org/2019/504`.

[O'C17] Russell O'Connor. Simplicity: A new language for blockchains. *CoRR/1711.03028*, 2017.

[PDT⁺19] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. *Security and Privacy 2020*, 2019.

[SKH18a] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *arXiv:1801.00687*, 2018.

[SKH18b] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Temporal properties of smart contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 323–338. Springer, 2018.

[Wan18] Peng Wang. *Type System for Resource Bounds with Type-Preserving Compilation*. PhD thesis, MIT, 2018.

[Zah18] Joachim Zahnentferner. Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies. Cryptology ePrint 2018/262, 2018. `https://eprint.iacr.org/2018/262`.

Solidity 0.5: when typed does not mean type safe

Silvia Crafa[1] and Matteo Di Pirro[2]

[1] University of Padova, Italy crafa@math.unipd.it
[2] Kynetics, Italy matteo.dipirro@kynetics.com

Abstract. The recent release of Solidity 0.5 introduced a new type to prevent Ether transfers to smart contracts that are not supposed to receive money. Unfortunately, the compiler fails in enforcing the guarantees this type intended to convey, hence the type soundness of Solidity 0.5 is no better than that of Solidity 0.4. In this paper we discuss a paradigmatic example showing that vulnerable Solidity patterns based on potentially unsafe callback expressions are still unchecked. We also point out a solution that strongly relies on formal methods to support a type-safer smart contracts programming discipline, while being retro-compatible with legacy Solidity code.

Keywords: type soundness · smart contracts · address type

1 Introduction

Over the last few years the execution of smart contracts on the blockchain has emerged as a form of distributed programming of a global computer. Anyone can deploy a global service, encoded as a smart contract, that can be used by mutually untrusted parties to "safely" interact with no need of a central authority. Therefore it is of paramount importance that the intended interaction provided by the service is "correctly" implemented by the code of the corresponding contract. Indeed, while the term *contract* is generally used to refer to an interaction that is intended to be enforced by law, a smart contract on the blockchain is intended to be *automatically enforced*: the law is embodied by the code to be executed (see the TheDAO affair [1]).

Formal methods have a long tradition of successes in dealing with the subtle mismatches between program specification and code implementation, and they can be helpful also in the new context of smart contracts. Here we focus on Solidity, the most widely used programming language in Ethereum's ecosystem, and on formal methods that provide support for a safer programming discipline by acting directly at the programming language level. In particular, since Solidity is a statically typed language, we foster the use of types as a tool to shape and substantiate the programmer's reasoning. However, static typing conveys an effective programming discipline only if type constraints are actually enforced by the compiler. In other terms, there is a gap between the definition of types in a language and their type-safe usage. We show below that this is precisely the case of the last release of Solidity 0.5. Indeed, the newly-introduced type `address payable` is intended to prevent Ether transfers to smart contracts that are not supposed to receive money, but the compiler fails to enforce such semantics. In other words, the type soundness of Solidity 0.5 is no better than that of the previous release.

Formal methods and the theory of typed languages show the way to bridge that gap and develop a statically typed language that is also type-safe. In particular, since Solidity contracts are reminiscent of class-based objects in distributed Object-Oriented Languages, it is worth to study how the rich and well-known theory of OOLs can be reused and adapted to smart contracts programming.

In a previous work we defined the Featherweight Solidity typed calculus ([3]), which formalizes the core of the Solidity language and the basic type soundness provided by its compiler (both versions 0.4 and 0.5). In that work we also proposed a refined typing that enjoys a stronger soundness property, but remains retro-compatible with legacy Solidity code. That typing ensures safer accesses to contracts through their address; hence it statically prevents a general class of runtime errors. We show here that the unsafe usage of the `address payable` type can be statically captured by the refined type system put forward in [3]. Therefore, it represents a solution to the soundness issue of Solidity 0.5 and supports an effective smart contract programming discipline using the compiler as a convenient building tool.

2 The problem

As in class-based Object-Oriented Languages, the declaration of a Solidity contract C defines a contract type C. However, instances of such a contract are often referred to by the Solidity code through expressions of type `address`, that essentially represent an untyped way to access them. Such expressions must then be cast to the type C in order to call the functions provided by the contract C. Casting an untyped pointer is notoriously a very flexible but subtle feature requiring programmers to precisely know what pointers refer to. Solidity's compiler provides no help here: neither static or dynamic checks are performed on cast expressions, and a dynamic error is raised only when calling a function (or accessing a state variable) that is not provided by the underlying contract.

Two features of Solidity make this problem pervasive in the code of smart contracts. First of all, in Ethereum the instances of smart contracts deployed on the blockchain can only be accessed through their public address. Secondly, contract functions make extensive use of their implicit variables `this` and `msg.sender`, that are dynamically bound to the *contract instance* being executed and the *address* of the caller contract, respectively. Therefore, while the callee is referred to through a typed pointer (as in OOLs), the caller is referred to through an untyped one. Hence, even though usual method recursion is type-safe, all the callback expressions undergo potentially unsafe usages. Indeed, besides the dangerous casts described above, a typical Solidity pattern consists in calling `msg.sender.transfer`(n) to send n Ether from the balance of the callee to that of the caller. However, such a transfer implicitly calls the *fallback* function of the contract referred to by `msg.sender`, thus raising a dynamic error if such function has not been defined by that contract.

To mitigate this problem, the last release of Solidity (i.e. version 0.5 [2]) distinguishes two types, `address` and `address payable`, where the second one denotes addresses pointing to contracts that declare the *fallback* function. Ideally, by using the new type `address payable`, Solidity 0.5 intends to statically prevent at least the unsafe money transfers, that are actually the most common form of the dynamic errors described above. It is worth to observe that these errors, that in OOLs are known as *message-not-understood*, are particularly harmful in the context of the blockchain. Indeed, in Ethereum the occurrence of a dynamic error causes the initial transaction to be interrupted and rolled-back (the so-called *revert*). This makes the account that issued that transaction lose the money it paid to the miner node and possibly leads to Ether indefinitely locked into a contract's balance. Hence, there is a pressing requirement to issue a transaction only if it can be statically guaranteed that it will not evolve to a revert.

Unfortunately, Solidity 0.5 fails to prevent unsafe money transfers at compile-time. As a matter of fact, no type check is enforced by the compiler to ensure that a variable of type `address payable` is substituted with the address of a contract that actually provides a *fallback* function. The problem can be detected with a careful read of the documentation[1], which states:

> *It might very well be that you do not need to care about the distinction between address and address payable and just use address everywhere. For example, if you use the withdraw pattern you most likely do not have to change your code because transfer is only used on msg.sender instead of stored addresses and msg.sender is an address payable.*
> *[...] Address literals can be implicitly converted to address payable.*
> *[...] In external function signatures address is used for both the address and the address payable type.*

Concretely, the counterexample in Figure 1 shows that the implicit variable `msg.sender` is assumed to be of type `address payable`, but no check is performed on the type of the actual caller's address. More precisely, the expression `msg.sender.transfer`(10) in the body of the function of the contract `Test` (line 26) correctly compiles, and so does the call of this function from the contract `WithoutFallback` (line 11). However, issuing a transaction that invokes the function `callUnsafeContract` of `WithoutFallback` results in a revert as that contract cannot receive money back from the contract `Test`. The same problem occurs if the functions are marked public or private instead of external. Furthermore, in order for the contract `WithoutFallback` to refer to a deployed instance of the contract `Test`, its constructor can only accept a parameter of address type and then cast it to the expected contract type (line 7). Even if nothing ensures that the actual parameter refers to an instance of `Test`, the cast expression correctly compiles and correctly executes, postponing the dynamic check to the

[1] https://solidity.readthedocs.io/en/v0.5.9/050-breaking-changes.html

```
1  pragma solidity >=0.5.0 <0.7.0;
2
3  contract WithoutFallback {
4    Test _test;
5
6    constructor (address _unsafeAddress) payable public {
7        _test = Test(_unsafeAddress);
8    }
9
10   function callUnsafeContract() external {
11       _test.foo();
12   }
13
14   function testUnsafeCast() external {
15      address _addr = address(_test);
16      //_addr.transfer(10); // DOES NOT COMPILE
17      address payable _payAddr = address(uint160(_addr));
18      _payAddr.transfer(10);
19     }
20 }
21
22 contract Test {
23   constructor () payable public {}
24
25   function foo() external {
26       msg.sender.transfer(10);
27   }
28 }
```

Fig. 1. Counterexample to the type safety of Solidity 0.5

moment where the `_test` reference is actually used (line 11). The constructor's parameter `_unsafeAddress` could also be of type `address payable`. In this case, one might expect the compiler to check that, when casting a payable address to a contract type, the target type of the cast (i.e. `Test`) at least defines a *fallback* function. Again, this is not true. No check is performed, either at compile-time or at run-time, to ensure that `Test` respects the constraints that `address payable` is supposed to impose.

The example also shows (in function `testUnsafeCast`) that the `transfer` primitive can be correctly used only on addresses of static type `address payable`, but the type constraint can be circumvented by resorting to an intermediate cast to the type `uint160`, as explicitly stated by the official documentation. Clearly, the expression `_payAddr.transfer(10)` at line 18 dynamically raises an error since there is no *fallback* function in the `Test` contract.

We tested the code in Figure 1 with Remix, the online Ethereum IDE, using the version *0.5.9+commit.e560f70d* of the Solidity compiler.

Money transfers that dynamically lead to errors were possible since the first release of Solidity, so the new version has not introduced a new problem. On the other hand, the addition of the new type `address payable` to capture the (addresses of) contracts that can "safely" receive Ether, generates into programmers the expectation that "safely" means type-safely, that is the compiler will check it. In fact nothing has actually changed w.r.t. version 0.4: the new type essentially provides only a refined documentation about addresses, but programmers have certainly more confounded expectations.

3 The solution

The typed theory of programming languages allows to identify a type preservation issue in Solidity 0.5's type system, confirmed by the code in Figure 1, and also offers a solution. In a previous work ([3]) we developed a precise formalization of the core of the Solidity language and its type system. We resorted to a formalization style that is reminiscent of the well known Featherweight Java language [4], highlighting the similarities between the notions of object and smart contract. Along with a precise definition of the basic type-soundness provided by the Solidity compiler, we proposed a refined type system that enjoys a stronger soundness property. In particular, that typing solves the type preservation problem pointed out here. Furthermore, the solution put forward in [3] is general enough to statically prevent not just unsafe calls to a non existent *fallback* function, but all the *message-not-understood* errors arising from unsafe casts from addresses to contract types.

The key idea is twofold. First, the type `address` is refined with type information about the contract it refers to. That is, $\texttt{address}\langle C\rangle$ is the type of the addresses of instances of the contract C, or of a contract that inherits form C. In particular, assuming a dummy contract $\texttt{Top}_{\texttt{fb}}$ that only contains a *fallback* function with an empty body, the type $\texttt{address}\langle\texttt{Top}_{\texttt{fb}}\rangle$ has the same meaning of Solidity 0.5's `address payable`. Indeed, it is the (super-)type of the addresses of every contract that can safely accept money transfers.[2]

The second idea is to enrich functions' signatures with the maximum type allowed for the caller, so that functions can only be invoked by contracts with an expected (super-)type. Adding a type constraint for the caller in function signatures is essential to safely type the implicit `msg.sender` parameter, thus to guarantee type preservation. The compiler can then statically check potentially unsafe callback expressions, such as $\texttt{msg.sender.transfer}(n)$ or `C(msg.sender).foo()`, that reduce to a revert if `msg.sender` is bound to the address of a contract that has no *fallback* function or does not have type C, respectively.

The counterexample in Figure 1 can be fixed by choosing a suitable refined signature for the `foo` function of the contract `Test`. As the only requirement for

[2] In [3] we proposed the keyword `payableaddress` as a syntactic sugar for the type $\texttt{address}\langle\texttt{Top}_{\texttt{fb}}\rangle$, since at the time of writing we were not aware of Solidity 0.5.

the caller is to provide a *fallback* function, it is sufficient to amend the function's code as follows:

```
function foo() <Top_fb> external {
   msg.sender.transfer(10);
}
```

In the body of the function, the variable `msg.sender` is then assumed to have type `address`⟨$\texttt{Top}_{\texttt{fb}}$⟩, hence the call to `transfer` is now well typed. On the other hand, the compiler prevents the unsafe money transfer by identifying a type error in the function call at line 11, since the caller's type, `WithoutFallback`, is not a subtype of $\texttt{Top}_{\texttt{fb}}$. As a further example, the following function, whose refined signature specifies the expected (super-)type of the caller, could be safely added to the `Test` contract:

```
function boo() <WithoutFallback> external {
   WithoutFallback(msg.sender).testUnsafeCast();
}
```

To simplify the notation, and in line with the Solidity programming style, in [3] we proposed a syntactic sugar based on a new function marker, `payback`, for functions whose caller must simply provide a *fallback* function (which is the most common case). In this way the `foo` function inside the `Test` contract would simply become as follows:

```
function foo() payback external {
   msg.sender.transfer(10);
}
```

Similarly, the standard function signature with no annotation could correspond to assuming the (super-)type `address`⟨`Top`⟩, that is no constraint for the caller.

Further details about the formalization of this idea, its type-soundness, and its retro-compatibility with Solidity contracts already deployed on the blockchain can be found in [3]. We just observe here that, despite the usage of the convenient `payback` marker, to take advantage of the full power of the refined typing the major effort required to Solidity programmers is to annotate their functions with the required (super-)type of the caller. Such a requirement might be verbose, but it actually supports a safer programming discipline, where types mirror the programmer's reasoning and the compiler can be effectively used as a convenient building tool.

References

1. E.J. Spode. The great cryptocurrency heist. *Aeon.* February 2017. Available at https://aeon.co/essays/trust-the-inside-story-of-the-rise-and-fall-of-ethereum
2. Solidity. https://solidity.readthedocs.io/en/develop/index.html. Release 0.5.9.
3. S. Crafa, M. Di Pirro, and E. Zucca. Is Solidity solid enough?. In *3rd Workshop on Trusted Smart Contracts*, 2019. Pre-workshop proceedings available at http://fc19.ifca.ai/wtsc/SoliditySolid.pdf
4. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

Towards a Smart Contract Verification Framework in Coq *

Danil Annenkov and Bas Spitters

Aarhus University

Abstract

We propose a novel way of embedding functional smart contract languages into the Coq proof assistant using meta-programming techniques. Our framework allows for developing the meta-theory of smart contract languages using the deep embedding and provides a convenient way for reasoning about concrete contracts using the shallow embedding. The proposed approach allows to make a connection between the two embeddings in a form of a soundness theorem. As an instance of our approach we develop an embedding of the Oak smart contract language in Coq and verify several important properties of a crowdfunding contract. The developed techniques are applicable to all functional smart contract languages.

1 Introduction

The concept of blockchain-based smart contracts has evolved in several ways since its appearance. Starting from the restricted and non Turing complete Bitcoin script[1] designed to validate transactions, the idea of smart contracts expanded to fully featured languages like Solidity running on the Ethereum Virtual Machine (EVM).[2] Recent research on the smart contract verification discovered the presence of multiple vulnerabilities in many smart contract written in Solidity [3, 6]. Several times the issues in smart contract implementations resulted in huge financial losses (for example, the DAO contract and the Parity multi-sig wallet on Ethereum). The setup for smart contracts is quite unique: once deployed, they cannot be changed and any small mistake in the contract logic may lead to serious financial consequences. This shows not only the importance of formal verification of smart contracts, but also the importance of principled programming language design. Next generation smart contract languages tends to employ the functional programming paradigm. A number of blockchain implementations have already adopted certain variations of functional languages as an underlying smart contract language. These languages range from minimalistic and low-level (Simplicity [5], Michelson[3]) to fully-featured OCaml- and Haskell-like languages (Liquidity [2], Plutus[4]). There is a very good reason for this tendency. Statically typed functional programming languages can rule out many mistakes. Moreover, due to the absence (or more precise control) of side effects programs in functional languages behave as mathematical functions that makes reasoning about them easier. However, one cannot hope to perform only stateless computations: the state is inherent for blockchains. One way to approach this is to limit the ways of changing the state. While Solidity allows arbitrary state modifications at any point of execution, many modern smart contract languages represent smart contract execution as a function from a current state to a new state. This functional nature of modern smart contract languages makes them well-suited for formal reasoning.

*This work is supported by the Concordium Blockchain Research Center, Aarhus University, Denmark.

[1] Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf

[2] Ethereum's white paper: https://github.com/ethereum/wiki/wiki/White-Paper

[3] https://www.michelson-lang.com/

[4] https://cardanodocs.com/technical/plutus/introduction/

The Ethereum Virtual Machine and the Solidity smart contract language remains one of the most used platforms for writing smart contacts. Due to the permissiveness of the underlying execution model and complexity of the language verification in this setting is quite challenging. On the other hand, many new generation languages such as Oak,[5] Liquidity and Scilla, offer a different execution model and a type system allowing to rule out many errors by means of type checking. Of course, many important properties are not possible to capture even with powerful type systems of functional smart contract languages. For that reason, to provide even higher guarantees, such as functional correctness, one has to resort to stronger type systems/logics for reasoning about programs and employ deductive verification techniques. Among various tools for that purpose proof assistants provide a versatile solution for that problem.

Proof assistants, or interactive theorem provers are tools that allow for stating and proving theorems by interacting with users. Proof assistants often offer some degree of proof automation by implementing decision and semi-decision procedures, or interacting with automated theorem provers (SAT and SMT solvers). Some proof assistants allow for writing user-defined automation scripts, or write extensions using a plug-in system. This is especially important, since many problems in the verification of programming languages are undecidable and providing users with a convenient way of interactive proving while retaining a possibility to do automatic reasoning makes proof assistants very flexible tools for verification of smart contracts.

Existing formalisations of functional smart contract languages in proof assistants focus either on meta-theory[6] or on verification of particular smart contracts translated by hand to Coq [7]. None of these developments combine deep and shallow embeddings in one framework or provide an automatic way of converting smart contracts to Coq programs. We are making a step towards this direction by allowing for deep and shallow embeddings to coexist and interact in Coq.

The contributions of this paper are the following: (1) we develop an approach allowing for developing in one framework the meta-theory of smart contract languages and convenient reasoning about concrete contracts; (2) we combine deep and shallow embedings using the metaprogramming facilities of the MetaCoq plug-in [1]; (3) as an instance of our approach we define the syntax and semantics of the Oak language (the deep embedding) and the corresponding translation of Oak programs into Coq functions (the shallow embedding); (4) we prove properties of a crowdfunding contract given as a deep embedding (abstract syntax tree) of an Oak program. We discuss details of our approach in Section 2 and provide an example of a crowdfunding contract in Section 3.

2 Our approach

There are various ways of reasoning about properties of a functional programming language in a proof assistant. First, let us split the properties in two groups: meta-theoretical properties (properties of a language itself) and properties of programs written in the language. Since we are focused on functional smart contract languages and many proof assistants come with a built-in functional language, it is reasonable to assume that we can reuse the programming language of a proof assistant to express smart contracts and reason about their properties. A somewhat similar approach is taken by the authors of the hs-to-coq library [8], which translates total Haskell programs to Coq by means of source-to-source transformation. Unfortunately, in

[5] The Oak language is an ML-style functional smart contract language with Elm-like syntax. Oak is currently under development at the Concordium foundation.

[6] Michelson meta-theory: `https://gitlab.com/nomadic-labs/mi-cho-coq/`
Plutus core meta-theory: `https://github.com/input-output-hk/plutus/tree/master/metatheory`
Simplicity meta-theory: `https://github.com/ElementsProject/simplicity/tree/master/Coq`.

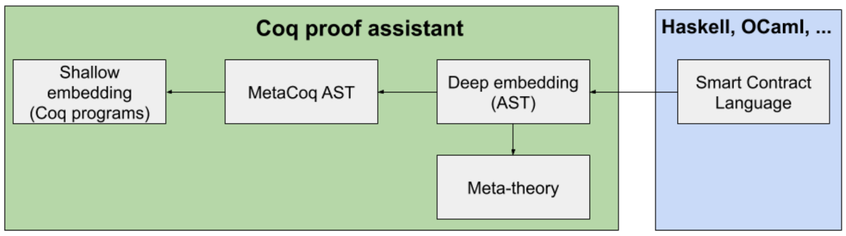


Figure 1: The structure of the framework

this case it is impossible to reason about the correctness of the translation.

We would like to have two representations of functional programs within the same framework: a deep embedding in the form of an abstract syntax tree (AST), and a shallow embedding as a Coq function. While the deep embedding is suitable for meta-theoretical reasoning, the shallow embedding is convenient for proving properties of concrete programs. We use the meta-programming facilities of the MetaCoq plug-in [1] to connect the two ways of reasoning about functional programs.

The overview of the structure of the framework is given in Figure 1. As opposed to source-to-source translations in the style hs-to-coq[8] and coq-of-ocaml[7] we would like for all the non-trivial transformations to happen in Coq. This makes it possible to reason within Coq about the translation and formalize the required meta-theory for the language. That is, we start with an AST of a program in a smart contract language implemented in Haskell, OCaml or some other language, then we generate an AST represented using the constructors of the corresponding inductive type in Coq (deep embedding) by printing the desugared AST of the program. By printing we mean a recursive procedure of converting the AST into a string consisting of the constructors of our Coq representation. The main idea is that this procedure should be as simple as possible and does not involve any non-trivial manipulations, since it will be a part of a trusted code base. If any non-trivial transformations are required, they should happen within the Coq implementation.

MetaCoq allows us to convert an AST represented as an inductive type into a Coq term. Thus, starting with the syntax of a program in our functional language, through a series of transformations we produce a MetaCoq AST, which is then interpreted into a program in Coq's Gallina language (shallow embedding). The transformations include conversion from the named to the nameless representation (if required) and translation into the MetaCoq AST. The deep embedding also serves as input for developing meta-theory of the smart contract language.

As an instance of our approach we develop an embedding of the Oak smart contract language to Coq.[8] The semantics of Oak is given as a definitional interpreter. This gives us an executable semantics for the language. The interpreter is implemented in an environment-passing style and works both with named and nameless representations of variables. To be able to interpret general fixpoints we evaluate fixpoints applications in the environment extended with the closure corresponding to the recursive call. Due to the potential non-termination, we define our interpreter using a *fuel idiom*: by structural recursion on an additional argument (a natural number).

Since the development of the meta-theory of Coq itself is one of the aims of the MetaCoq we can use this development to show that the semantics of our functional language agrees with its

[7] The coq-of-ocaml github page: `https://github.com/clarus/coq-of-ocaml`
[8] Our Coq development `https://github.com/annenkov/FMBC19-artefact/`, including examples from Section 3

```
(* Defining AST using customised notations      *)

(* Brackets [\ \] delimit the scope of global  *)
(* definitions and [| |] the scope of programs  *)

Definition state_syn : global_dec :=
  [\ record State :=
     { balance : Money ;
       donations : Map ;
       owner : Money ;
       deadline : Nat ;
       goal : Money } \].

Make Inductive (trans_global_dec state_syn ).

Definition action_syn : global_dec :=
  [\ data Action :=
         Transfer : Address → Money → Action
       | Empty : Action ; \].

Make Inductive (trans_global_dec action_syn ).

Definition result_syn : global_dec :=
  [\ data Result :=
       Res : State → Action → Result
     | Error : Result ; \].

Make Inductive (trans_global_dec result_syn ).

Definition msg_syn : global_dec :=
  [\ data Msg :=
       Donate : Msg
     | GetFunds : Msg
     | Claim : Msg ; \].

Make Inductive (trans_global_dec msg_syn ).
```

```
Definition crowdfunding : expr :=
[| \c : Ctx ⇒ \s : State ⇒ \m : Msg ⇒
     let bal : Money := balance s in
     let now : Nat := cur_time c in
     let tx_amount : Money := amount c in
     let sender : Address := ctx_from c in
     let own : Address := owner s in
     let accs : Map := donations s in
     case m : Msg return Result of
     | GetFunds →
       if (own == sender) && (deadline s < now)
           && (goal s ≤ bal ) then
       Res (mkState 0 accs own (deadline s) (goal s))
           (Transfer bal sender )
      else Error : Result
    | Donate → if now ≤ deadline s then
      (case (mfind accs sender ) : Maybe return Result of
       | Just v →
        let newmap : Map :=
           madd sender (v + tx_amount ) accs in
         Res (mkState (tx_amount + bal) newmap own
             (deadline s) (goal s)) Empty
       | Nothing →
         let newmap : Map := madd sender tx_amount accs in
         Res (mkState (tx_amount + bal) newmap own
             (deadline s) (goal s)) Empty )
       else Error : Result
    | Claim → if (deadline s < now) && (bal < goal s) then
       (case (mfind accs sender ) : Maybe return Result of
        | Just v → let newmap : Map := madd sender 0 accs in
          Res (mkState (bal − v) newmap own (deadline s)
              (goal s)) (Transfer v sender )
        | Nothing → Error)
        else Error : Result |].

Make Definition entry :=
     Eval compute in (expr_to_term (indexify crowdfunding )).
```

Figure 2: The crowdfunding contract

translation to MetaCoq (on terminating programs) and our interpreter is sound with respect to the embedding. We compare the results of evaluation of Oak expressions with the weak head call-by-value evaluation relation of MetaCoq up to appropriate conversion of values. Currently, the full formalisation of this proof is under development. Being able to relate the semantics of Oak to the semantics of Coq through Coq's meta-theory formalisation gives stronger guarantees that our shallow embedding reflects the actual behaviour of Oak programs. The described approach provides a more principled way of embedding functional language, in contrast to the source-to-source based approaches.

3 The crowdfunding contract

As an example of our approach we consider verification of some properties of a crowdfunding contract (Figure 2). Such a contract allows arbitrary users to donate money within a deadline. If the crowdfunding goal is reached, the owner can withdraw the total amount from the account after the deadline has passed. Also, users can withdraw their donations after the deadline if the goal has not been reached. This is a standard example of a contract and it appears in a number of publications related to smart contract verification.

We extensively use a new feature of Coq called "custom entries" to provide a convenient notation for our deep embedding.[9] The program texts in Figure 2 written inside the special brackets [\ ... \] and [| ... |] are parsed according to the custom notation rules. For example, without using notations the definition of `action_syn` looks as follows:

```
gdInd Action 0 [("Transfer", [(nAnon, tyInd "nat"); (nAnon, tyInd "nat")]);("Empty", [])] false.
```

This AST otherwise would be printed directly from the smart contract AST by a simple procedure (as we outlined in Section 2). We start with defining the required data structures such

[9] Custom entries are available in Coq 8.10

as `State`, `Action`, `Result` and `Msg` meaning contract state, resulting contract actions, the type of results (equivalent to the `option` type of Coq) and messages accepted by this contract. We pre-generate string constants for corresponding names of inductive types, constructors, etc. using the MetaCoq template monad.[10] This allows for more readable presentation using our notation mechanism. Currently, we use the `nat` type of Coq to represent account addresses and currency. Eventually, these types will be replaced with corresponding formalisations of these primitive types.

The `trans_global_dec` : `global_dec` → `mutual_inductive_entry` function takes the syntax of the data type declarations and produces an element of `mutual_inductive_entry` — a MetaCoq representation for inductive types. For each of our deeply embedded data type definitions we produce corresponding definitions of inductive types in Coq by using the `Make Inductive` command of MetaCoq that "unquotes" given instances of the `mutual_inductive_entry` type. Similar notation mechanism is used to write programs using the deep embedding. The definition of `crowdfunding` represents a syntax of the crowdfunding contract. We translate the crowdfunding contract's AST into a MetaCoq AST using the `expr_to_term` : `global_env` → `expr` → `term` function. Here, `global_env` is a global environment containing declarations of inductive types used in the function definition, `expr` is a type of Oak expressions, and `term` is a type of MetaCoq terms. Before translating the Oak AST we apply the `indexify` function that converts named variables into De Bruijn indices. The result of these transformations is unquoted with the `Make Definition` command. The corresponding function has the following type `entry` : `ctx` → `State_coq` → `Msg_coq` → `Result_coq`, where `ctx` is a call context containing current block time, transferred amount, sender's address and other information available for inspection during the contract call. The type names with the "coq" postfix correspond to the unquoted data types from the Figure 2.

The `entry` function corresponds to a transition from the current state of the contract to the new state. That allows for proving functional correctness properties using pre- and post-conditions. Similarly to [7], we a prove number of properties of the contract using the shallow embedding. Specifically, we proved the following properties: the contract does no leak funds; the donations can be paid back to the backers if the goal is not reached within a deadline; donations are recorded correctly in the contract's state. Moreover, in our Coq development, we show how one can verify library code for Oak by proving Oak functions equivalent to the corresponding functions from the standard library of Coq. In particular, we provide an example of such a procedure for certain functions on finite maps.

4 Related work

In this work we focus on modern smart contract languages based on a functional programming paradigm. In many cases various small errors in smart contracts can be ruled out by the type systems of these languages. Capturing more serious errors requires employing such techniques as deductive verification (for verification of concrete contracts) and formalisation of meta-theory (to ensure soundness of type systems). Works related to formalisation of such languages are mentioned in Section 1 and include languages like Plutus, Michelson, Liquidity, Scilla and Simplicity.

[10] The template monad is a part of the MetaCoq infrastructure. It allows for interacting with Coq's global environment: reading data about existing definitions, adding new definitions, quoting/unquoting definitions, etc.

5 Conclusion and future work

We have presented a work-in-progress on the smart contract verification framework. An important feature of our approach is the ability to both develop a meta-theory of a smart contract language and to conveniently reason about smart contracts. One can prove soundness theorems relating meta-theory of the smart contract language with the embedding. Such an option is usually not available for source-to-source translations. We applied our approach to the development of an embedding of the Oak smart contract language and provided an example of verification of a crowdfunding contract starting from the contract's AST. However, the approach is quite general and applies to other functional smart contract languages.

As future work, we would like to provide integration with Oak-language infrastructure allowing for a convenient translation of Oak programs to Coq. Since our framework is not focused on one particular smart contract language, we also consider benchmarking our development by developing "backends" for translation of other languages (e.g. Liquidity, Simplicity). Currently, our framework allows for proving functional correctness of contracts corresponding to one "step" from the current state to the new state. To be able to reason about the chain of contract calls one needs an execution model to be formalised in Coq as well. We plan to connect our development to the ongoing work on formalising such and execution model for the Oak programming language[4].

Extending the formalisation of the Oak language meta-theory is also among our goals for the framework. An important bit of Oak's meta-theory is the cost semantics allowing for reasoning about "gas". We would like to give a cost semantics for the deep embedding and explore how it can be extended on the shallow embedding.

References

[1] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards Certified Meta-Programming with Typed Template-Coq. In *ITP18*, volume 10895 of *LNCS*, pages 20–39, 2018.

[2] Çagdas Bozman, Mohamed Iguernlala, Michael Laporte, Fabrice Le Fessant, and Alain Mebsout. Liquidity: OCaml pour la Blockchain. In *JFLA18*, 2018.

[3] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS16*, pages 254–269, 2016.

[4] Jakob Botsch Nielsen and Bas Spitters. Smart Contract Interactions in Coq. Submitted to FMBC19.

[5] Russell O'Connor. Simplicity: A New Language for Blockchains. PLAS17, pages 107–120. ACM, 2017.

[6] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In *Financial Cryptography and Data Security*, pages 478–493, 2017.

[7] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR*, abs/1801.00687, 2018.

[8] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total Haskell is Reasonable Coq. CPP18, pages 14–27. ACM, 2018.