A model of Solidity-style smart contracts in the theorem prover Agda

1st Fahad Alhabardi
Department of Computer Science
Swansea University
Swansea, United Kingdom
fahadalhabardi@gmail.com

2nd Anton Setzer
Department of Computer Science
Swansea University
Swansea, United Kingdom
a.g.setzer@swansea.ac.uk

***Abstract*—The use of smart contracts is transforming traditional industry and business practices. It enables the automatic enforcement of contractual terms without the need for a trusted third party. Smart contracts can automate a variety of transactions on Blockchain. Despite their numerous benefits, some challenges, such as security vulnerabilities, still need to be addressed before smart contracts can be widely adopted.**

This paper introduces two models of smart contracts – one simple and one more complex – using the interactive theorem prover Agda. This is a step towards converting the previous work of verifying Bitcoin smart contracts using weakest preconditions [1], [2] to Ethereum's Solidity-style [3] smart contracts. Since Ethereum's contracts are object-oriented, this model is substantially more complex than Bitcoin's. We provide models supporting simple and complex executions, the calling of other contracts, and functions referring to addresses and messages. Furthermore, these models also support transferring money to other contracts and updating specific contracts, and the more complex model includes gas cost and pure functions.

***Index Terms*—Ethereum, smart contracts, Agda proof assistant, cryptocurrency, theorem prover, Solidity, Model of solidity-style, transaction, blockchain**

I. Introduction

Among the most popular emerging technologies at the moment is blockchain. Its fundamental benefit is that blockchain technology allows transactions without relying on other authorities such as banks. It also offers data integrity, built-in authenticity, and user transparency. Blockchain has many applications, such as cryptocurrencies and smart contracts.

A smart contract is an application on the blockchain initially suggested by Nick Szabo in 1990 [4]. A smart contract is a program that is automatically executed when the agreement conditions between the parties involved as recorded on the blockchain are fulfilled. By coding their terms, smart contracts automate agreements. When all conditions are met, the code enforces the agreement automatically, removing the need for a third party. This eliminates fraud, mistakes, and processing time. When smart contracts are kept on a blockchain, trust is assured since the blockchain forbids any changes or tampering with the smart contract's conditions [5], provided the blockchain is not changed by e.g. a 51% attack. Smart contracts and blockchain technology have the potential to speed transactions while also making them transparent and secure without third parties [6].

The simplest example of a smart contract on the blockchain decentralised network is the buying and selling of products and services: Buyers deposit money on the blockchain for sellers. The funds are not paid until the buyer signs again after receiving the goods. Customers are reimbursed if items are late [7].

Smart contract codes are immutable [8] when deployed on the blockchain network. The only way to amend the clauses of an ongoing smart contract or to withdraw it is by using functions already provided by the original contract. Thus, the developers must ensure and verify the security of the code before publishing it on the blockchain in order to avoid any errors. Errors in smart contract programs can result in massive losses, as exemplified by the case of the DAO. The DAO was a decentralised autonomous organisation whose contracts were manipulated by cyber criminals once the fund's market value had reached US$ 150 million.

In order to avoid any potential risk that may be related to the use of smart contracts such as errors in the codes of smart contracts or vulnerability to hacking, one needs to verify the correctness of smart contracts. This needs to be done before deploying them on the blockchain network. There are two ways of achieving this [9]: formal verification and execution of test cases. Formal verification techniques use mathematical approaches to prove program correctness. In the context of smart contracts this can be done by building a formal smart contract model and showing the smart contracts in question are correct. In contrast, the execution of test cases runs the code in order to ensure that for valid inputs, execution terminates and produces correct outputs, while checking as well for possible weaknesses or security flaws. As an example of an erroneous code in smart contracts, consider a smart contract which is intended to transfer money from one particular account to another, but because of a coding error, results in the money being moved to an incorrect account. If this code can be invoked by a transaction there might be no way to reverse it. This could have serious consequences for the parties involved in the contract.

Smart contracts can be written in many different languages, for instance, high-level languages such as Solidity [3] or Liquidity [10], intermediate representations such as Simplicity [11], and low-level languages such as SCRIPT [12] or

Ethereum Virtual Machine bytecode (EVM) [13].

In this paper, we present the first step towards verifying the smart contracts in Ethereum using weakest preconditions. Weakest preconditions give a precise meaning to a contract by determining the exact minimal conditions required for the execution of a contract to end up in a given state, and therefore state the security of contracts. We have applied this approach with coauthors to Bitcoin [1], [2], and it is the goal of the current project to extend this approach to Solidity-style contracts. In this paper, we take a first step towards this project by building a model for Solidity-style contracts in the interactive theorem prover Agda [14] – we plan to carry out the verification of smart contracts in a future next step using this model. Because of the object-oriented features of Solidity, this is a more complex endeavour than the simple model for Bitcoin Script.

The advantage of using Agda is that it can be used as well as a functional programming language based on dependent types. Agda allows to develop programs, reason about them, and verify them by using the same language, avoiding translation errors from one language to another.

We build two kinds of models: a simple and a complex one. The simple model supports basic instructions such as transferring funds, calling other contracts, updating particular contracts, looking up the current address, the calling address and returning the balance of a specific contract. The simple model does not support the gas cost. In contrast, the complex model supports all of the features mentioned for the simple model, and adds gas cost, supports complex instructions, and deals with pure functions.

The rest of this paper is organised as follows: We introduce and evaluate related work in Sect. II. In Sect. III, we give an overview of Agda proof assistant and Ethereum. Then, we develop the simple and the complex models with examples for the Solidity-style smart contracts in Sect. IV. Finally, we conclude with a conclusion and future work in Sect. V.

Git repository. This work has been developed and formalised in the proof assistant Agda. All displayed Agda code in this paper has been generated from type-checked Agda code. The source code is available at [15].

II. Related work

This section gives a survey of articles that use a theorem prover to verify smart contracts. Then we provide some methods that may be used to verify smart contracts, such as model checking and symbolic execution. Following that, we discuss several articles on translating smart contract code into languages used for program verification. Furthermore, we present some tools that can be used to verify and analyse smart contracts. In addition, we provide some projects that use a novel language to verify smart contracts. Our previous papers [1], [2] contained a more extensive literature review.

Verification of smart contracts using theorem proving. Several projects used theorem provers to verify smart contracts. Nielsen et al. [16] proposed a model and executable specification for the execution of smart contracts in the proof assistant Coq [17], [18]. Then they used their formalisation to enable inter-contract communication and generalise existing accomplished work by enabling the modelling of depth-first execution blockchains (such as Ethereum) as well as breadth-first execution blockchains (such as Tezos). They represented smart contract programs in Gallina, Coq's functional language. It is possible to derive certified programs from this language using other languages, such as Haskell [19] or OCaml [20]. In addition, they also developed a contract for Congress that is a simpler version of a DAO contract. There are some restrictions in their work, such as the gas cost is not computed automatically at the moment with their shallow embedding.

Zakrzewski et al. [21] assessed the practicability of formalising the Solidity programming language [3] and suggested formalising a subset of Solidity that includes its core data model and specific distinctive characteristics such as function modifiers, contracts with storage, and inheritance hierarchy. They utilised the Coq proof assistant to provide an interpreter for Solidity that is formalised, with an emphasis on dynamic semantics. Their work has some limitations, specifically, they only dealt with a subset of Solidity. Additionally, their work does not support C99-like block scoping for local variables. Furthermore, their focus has been on formalization and therefore cannot be utilized for smart contract verification.

Andrei [22] has verified Findel [23]-written financial derivatives on blockchain networks. Findel is a declarative financial domain-specific language (DSL). Next, the author used the Coq proof assistant to define Findel's formal semantics and test it against the Findel test suite. The author then enhanced its semantics with interactive ways to formalise and verify Findel contract properties. Finally, the author aimed to ensure no errors exist in the Findel contracts. The limitation of their work is when using Coq, the automated proof search techniques often do not provide proof certificates automatically, even though they are correct. This limitation can be overcome by using human-verified proof certificates. Furthermore, Findel does not support loops and refund mechanisms in case of errors.

Verification of smart contracts using model checking and symbolic execution. Mossberg et al. [24] presented Manticore, a dynamic symbolic execution framework which is open-source. The Manticore framework was designed to analyse Ethereum smart contracts and binary code. This framework has been implemented in Python. Manticore's architecture is flexible, enabling it to support conventional and unconventional execution environments. Its API allows users to customise their analysis. The aim of using Manticore is bug detection and code verification. Limitations of the Manticore tool are that it cannot detect various vulnerabilities, including suicide and integer overflows.

Verification by translation into other languages. Several efforts aim to translate the smart contract code into languages used for program verification. Ahrendt et al. [25] focus on verifying Solidity smart contracts by automatically translating them into Java. Their Java translation can use verification tools and benefit from contract-oriented and object-

oriented paradigms. They validated the translated software using KeY [26], one of the most potent object-oriented language verification tools supporting transactions and their abortion. One limitation of their approach is that is impossible in their approach to access values such as the current block number and timestamp, which is possible in Solidity.

Luís et al. [27] developed WhylSon, a tool for deductive verification of smart contracts written in Michelson, the Tezos blockchain's low-level programming language. WhylSon instantly converts a Michelson contract into a WhyML program. In addition, they built a WhyML shallow-embedding of smart contract instructions' axiomatic semantics. Finally, they used WhylSon to verify smart contracts automatically. One limitation of their work is that they did not include a formalisation of the internal aspects of cryptographic operations.

Using tools to verify and analyse smart contracts. There are several different studies to verify and analyse smart contracts using various tools. Nikolić et al. [28] proposed Maian, a tool for describing and reasoning about trace features using inter-procedural symbolic analysis and a concrete validator of the byte-code of smart contracts in Ethereum. Maian has been implemented in Python. They focused on three defining characteristics of trace vulnerabilities: discovering contracts that either hold funds permanently, leak to arbitrary users or can be terminated by anyone. The Maian tool is limited to flagged contracts that are actively operating in the forked Ethereum chain or contracts having available source code.

Grieco et al. [29] introduced Echidna, a static analysis tool and an open source for Ethereum smart contracts fuzzer. Echidna has been created using the Haskell programming language, which supports three key features: user-defined properties, assertion testing, and gas usage estimate characteristics. Echidna can test smart contracts developed using Solidity and Vyper [30] programming languages. However, Vyper is no longer actively maintained at the time of writing this paper. One limitation of Echidna is, that it works only on single-core machines. Furthermore, there is no room for improvement in the accuracy of gas usage measurement at the moment. Echidna is compatible with various contract development frameworks such as Truffle and Embark.

Verification of smart contracts written in novel languages. Sergey et al. [31] developed a new and intermediate-level programming language called Scilla, designed for safe smart contracts. Scilla is intended to function as both a compilation target and a standalone programming framework. Scilla provides robust safety assurances through type soundness, utilising System F [32] as its fundamental calculus. Implementing smart contracts ensures a clear distinction between the computational, state-manipulating, and communication aspects. This approach mitigates several well-known challenges from executing contracts in a Byzantine environment and proposes a framework for conducting lightweight verification of Scilla programs, which has been demonstrated by applying two domain-specific analyses on real-world use cases. Scilla has various limitations since it is a language launched only recently towards the end of 2019. Therefore, there may be errors and issues in this language. Furthermore, this language was created specifically for Zilliqa contracts and has not been as extensively used as other languages.

Bartoletti et al. [33] proposed a fundamental calculus for smart contracts called TinySol (Tiny Solidity). This calculus contains an imperative core, further enhanced with a sole construct for invoking contracts and effectuating currency transfers. The present formalisation is a foundation for providing semantics to the Ethereum blockchain. Moreover, this approach prevents the particular challenges Solidity faces, such as variations in invoking other contracts. Some limitations to their work include the lack of support for a gas mechanism and the absence of certain features present in Solidity. Furthermore, their work has yet to incorporate recorded timestamps in the Blockchain.

Crafa et al. [34] introduced Featherweight Solidity. This calculus formalises the key aspects of the Solidity language and allows reasoning about the safety qualities of the smart contract source code. Many problems, such as access to a function or state variable that does not exist, are discovered only during run-time, resulting in the stoppage and rolling back of transactions. They then suggested a type of system modification that statically catches additional faults, such as unsafe casts and call-back expressions, and is retro-compatible with the original Solidity code. Featherweight Solidity was specifically designed to avoid certain problems that arise inside smart contracts, and therefore there might still be flaws in Featherweight Solidity, not yet addressed yet in its design.

III. Background – The proof assistant Agda and Ethereum

A. The theorem prover Agda

Agda [14], see as well the books [35], [36], is a theorem prover based on intensional Martin-Löf type theory [37]. Agda is very similar to Haskell in both spirit and syntax [38], [39]. Programmers who know Haskell will find Agda easy to learn. The main difference to Haskell is that Agda is based on dependent types, and is as well an interactive theorem prover. Agda [40], [41] additionally features parameterized modules, mixfix operators, and Unicode characters. Agda uses the Emacs interface, which provides a development environment that assists developers in developing and verifying proof code.

The MAlonzo compiler [42]–[44] is used in Agda to transform programs into Haskell and executable code. Agda can therefore be regarded a programming language that supports dependent types and functional programming. As such, Agda is well-suited for creating programs, specifications, and proofs.

In Agda, there is no difference between types expressing data and types expressing formulas, and both exist in type signatures and program code. Agda requires all functions to be total and terminating for type checking [40]. Otherwise type checking might not terminate, and Agda would be inconsistent. We give here only a brief introduction, the reader might refer to our previous papers [1], [2], for a more detailed introduction into Agda.

As an example, the Agda standard library defines the inductive type of natural numbers as follows:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

The definition above includes a new type called ℕ with two constructors, zero and suc.

As an example for Agda's record type we define an implementation (Contract) of a simple smart contract, which contains the fields amount and fun. The field amount represents the balance (amount of ether) in each contract, and fun represents the programs executed when a function with a given name is called with arguments encoded as a message: In Subsect. IV-B, we explain the data type in more detail. The definition of Contract is as follows:

```
record Contract : Set where
  field
    amount : Amount
    fun    : FunctionName → Msg
             → SmartContractExec Msg
```

In Agda, we can define functions using pattern matching through the elements of (ℕ). An example is

```
_==b_ : ℕ → ℕ → Bool
zero  ==b zero  = true
zero  ==b suc n = false
suc n ==b zero  = false
suc n ==b suc m = n ==b m
```

The Boolean equality function decides whether two naturals numbers n m are equal. It is defined by recursion on n and m; we use pattern matching to decide which of the 4 cases applies, where the last one is a recursive call.

B. Ethereum

Ethereum is the first of the second generation of cryptocurrencies and the most prominent example of a blockchain platform fully supporting smart contracts. Vitalik Buterin [13] launched Ethereum in 2013 with the intention of overcoming several shortcomings shown by Bitcoin's scripting language. The primary contribution is full Turing completeness: Smart contracts in Ethereum are capable of supporting all forms of computing, including loops and calling of other contracts.

Ethereum is a kind of blockchain that includes a Turing-complete programming language as part of its core functionality. Anybody can deploy smart contracts. They are essentially a collection of functions, which can be called together with their arguments. In addition contracts have instance variables, which define its state. The writer of the smart contracts can add conditions required for the successful execution of its functions. Smart contracts allow anybody to design their own rules for ownership, forms of transactions and state transition mechanisms [13].

In the past, Ethereum was based on a consensus mechanism known as proof of work [13]. It is now built on proof of stake, which is, compared proof-of-work, more secure, uses less energy, and is more suited for adopting new scaling solutions [45]. Validators are compensated in cryptocurrency for their labour in processing transactions, executing smart contracts and contributing to the creation of blocks [46].

Every node in the Ethereum network operates under the Ethereum Virtual Machine (EVM), a virtual distributed computer designed specifically for the Ethereum network. This machine is responsible for carrying out the commands given by the network. The EVM executes EVM code, which is a machine language for smart contracts. Smart contracts written in high level languages such as Solidity are compiled into the EVM. After being converted into EVM code, the smart contracts are subsequently executed by the network's nodes. Solidity [3] is now among the most popular programming languages for writing smart contracts in Ethereum. Solidity is a high-level language that implements user interactions, provides the capability for groups that use different blockchains to share information and value and overcomes the limitations mentioned in the Bitcoin scripting language [47].

The state of Ethereum comprises accounts, and each account has a 20-byte address in addition to state transitions. The global state is a mapping between addresses and account statuses [13]. There are two kinds of accounts that may be held on Ethereum: externally owned accounts, which are managed by private keys, and contract accounts, which are controlled by deployed contract code [13].

There are four components that compose an Ethereum account [13], [48]: The first is a nonce, which is the number of transactions dispatched from a given address, or the number of contracts created by an account. Its purpose is to prevent replay attacks, where a transaction would be identically repeated by an adversary. The second is the balance, which represents the number of Wei owned by the specified address. Wei is ETH's smallest unit of currency, and 1 Ether equals 10^{18} Wei. The balance is as well used to pay transaction fees. The third is the contract code hash, namely the Keccak-256 hash of the Ethereum Virtual Machine (EVM) code associated with an account. This code is executed whenever the account receives a message call at its address. The last is a storage root, referred to as the 256-bit root node hash in a Merkle Patricia tree (commonly referred to as tries), a data structure used for safe and efficient data storage and retrieval. This tree is responsible for encoding the storage contents of an account.

The following are some of the fundamental elements that are included in every transaction in Ethereum [13]: the field that provides the signature of the sender of the transaction, the field that identifies the destination address of the transaction, the field that defines the bytecode of the smart contract or the parameter that is sent in when calling the contract, *startgas*, *gasprice* values, and data fields that are optional. *Startgas* and *gasprice* [13] restrict the amount of computation a transaction may do. The maximum number of computing steps that a transaction may perform is specified by *startgas*, and the transaction will fail if it exceeds its *startgas* limit. This solve the problem that the EVM is Turing complete, and it is

undecidable whether a program in a Turing complete language terminates. This would cause problems since validators of transaction have to execute transactions which includes the execution of smart contracts, without knowing whether they terminate. By adding the limit set by *startgas*, termination of execution is enforced, by stopping execution when the gas limit is exceeded, solving this problem. *Startgas* and *gasprice* [13] aid as well in preventing denial-of-service attacks. The *gasprice* is the charge the sender pays for each unit of gas used. The greater the *gasprice*, the greater the likelihood that a transaction will be mined rapidly. The Ethereum fee structure ensures that attackers pay for the resources they utilise. Computation, bandwidth, and storage are all part of this. As a result, if a transaction requires more resources, the gas cost will be greater.

A transaction modifies the Ethereum blockchain's state using the deterministic Ethereum state transition function [13]. The function begins by confirming the transaction's validity, including checking the signature and nonce. If the transaction is correct, then the function subtracts the transaction fee from the sender's account balance and increments the nonce. The receiver receives the required amount of Ether after paying the transaction cost per byte. The recipient's account is created if it doesn't exist. The contract code is run if the recipient's account is a contract. The state transition function returns all state changes except the miners' payment fees if the sender doesn't have enough Ether or the code execution runs out of gas.

IV. Modelling of Solidity smart contracts in Agda

In this section, we develop a simple and a complex model of Solidity-style smart contracts. First, we provide a brief overview of these models in Subsect. IV-A. Then, we explain the simple model in Subsect. IV-B and the complex model in Subsect. IV-C. Both are implemented in Agda.

A. Overview on simple and complex models

This subsection will explain the functioning of simple and complex models in the ledger. As shown in Figure 1, the ledger comprises various contracts, including Contract 1, Contract n, and so on. The complex model's Contract 1 comprises four fields, namely the contract balance (amount), function name (fun), pure function (purefunction), and pure function cost (purefunctionCost). In contrast, the simple model only has two fields, i.e., amount and fun, as it deals with simple instructions. As an illustration of how it works, Contract 1 will use the command call to call Contract n with the parameters (funname, msg). Contract n might call other contracts as well. Once Contract n returns the result using the command return, Contract 1 will continue execution which might result in calls to other contracts, until, if it terminates, it will return its result to the caller using the statement "return result (msg)". During this process, it will calculate as we the amount of gas used, and abort execution in case it runs out of gas.

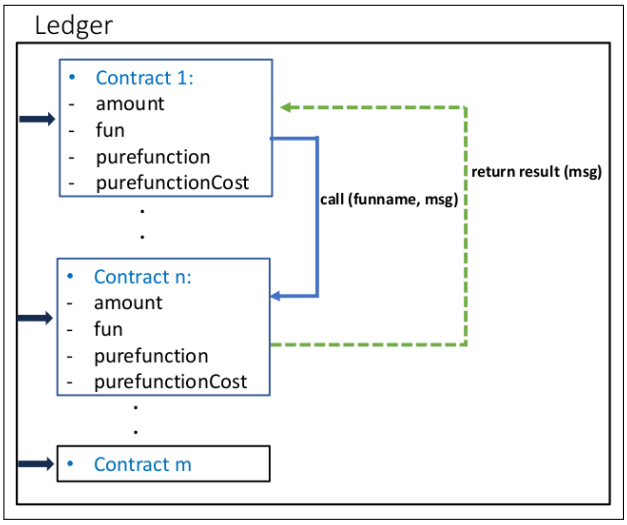

Fig. 1. Ledger in complex model

In addition, when returning the result to Contract 1, we utilize the state execution function to update the ledger state, as shown in Figure 2. The complex model comprises nine fields, ledger, executionStack, initialAddr, . . . , msgevalState. Conversely, the simple model has only the first five of those fields: ledger, executionStack, lastCallAddress, calledAddress and nextstep.

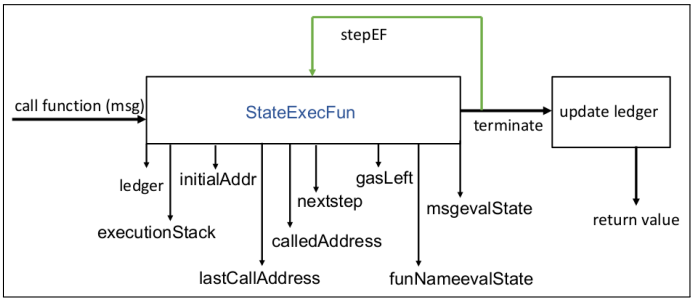

Fig. 2. Execution of function in complex model

In the following subsection, we explain the simple model in IV-B and complex model in IV-C in more detail.

B. Simple model of Solidity smart contracts in Agda

In this subsection, we develop a simple model of Solidity smart contracts, which supports basic executions such as updating smart contracts, transferring money, calling other smart contracts, obtaining the balance of each smart contract, but does not provide an explicit cost of gas.

In the Ethereum virtual machine, one may call functions by passing data to them as arguments. These arguments will then be serialise as a byte array, which is essentially a natural number. In order to provide an abstraction from this in our model, we have defined a type for messages (keyword data). Messages are inductively defined as natural numbers, or lists of messages. Messages allow to encode the elements of data types of Solidity. For instance, arrays are encoded as lists of

messages where each message encodes an element of the array. Maps are encoded as lists of pairs of messages, where pairs are lists of length 2 which represent the key and the element it is mapped to, both encoded as messages.

```
data Msg : Set where
  nat : ℕ        → Msg
  list : List Msg → Msg
```

Then, we define the execution stack, which is the list of currently open function calls. It is list of ExecStackEl, where ExecStackEl is defined as a record type as follows:

```
record ExecStackEl : Set where
  field
    lastCallAddress : Address
    calledAddress   : Address
    continuation    : Msg
                      → SmartContractExec Msg
```

ExecStackEl has three fields: lastCallAddress which gives the address which made the last call; calledAddress, the address which was called; continuation; which determines the next execution step to be executed depending on the message returned after the call to the function has been completed.

The ExecutionStack is a stack (or list) of ExecStackEl, listing the calls which are still open:

```
ExecutionStack = List ExecStackEl
```

We now define mutually SmartContractExec, which determines the next step in the execution of a smart command, CCommands, which is a command to be executed, and CResponse, which determines the answer returned, once a command is executed, as follows:

```
data SmartContractExec (A : Set) : Set where
  return : A → SmartContractExec A
  error  : ErrorMsg → SmartContractExec A
  exec   : (c : CCommands)
         → (CResponse c → SmartContractExec A)
         → SmartContractExec A

data CCommands : Set where
  transferc : Amount → Address
            → CCommands
  callc     : Address → FunctionName
            → Msg → CCommands
  updatec   : FunctionName
            → (Msg → SmartContractExec Msg)
            → CCommands

CResponse : CCommands → Set
CResponse (transferc amount addr) = ⊤
CResponse (callc addr fname msg)  = Msg
CResponse (updatec fname fdef)    = ⊤
```

SmartContractExec has three constructors: return, which will cause the execution to end and return its argument; error, which will cause execution to abort and return an error message; exec, which will execute a command, and depending on the response returned will continue execution. The function exec refers to the following CCommands which can be executed:

- transferc transfers a certain amount of money to a specific address;
- callc makes a recursive call to a function at a given address, with argument given by an element of Msg;
- updatec updates a function definition in the current contract;

In the full version of the code [15] we have as well commands for looking up the current address, the call address, and the balance of any address.

In case of transferc, the CResponse is the trivial type ⊤(having one element), in case of callc, the answer is the result returned by the function call executed, represented as an element of Msg, and in the case of updatec, it is an element of ⊤. The CResponse for the other commands (looking up the current address, the called address, and getting the balance of any address), are available at [15].

Furthermore, we define Contract as given by the balance and the functions to be executed, and a Ledger as a function which determines for each address the Contract at that address (with default values used for addresses which are not used):

```
record Contract : Set where
  field
    amount : Amount
    fun    : FunctionName → Msg
           → SmartContractExec Msg

Ledger = Address → Contract
```

The state of executing a smart contract StateExecFun consists of four fields: the current ledger (ledger), the execution stack (executionStack), the address which made the last call (lastCallAddress), the last address which was called (calledAddress), and the current code to be executed (nextstep):

```
record StateExecFun : Set where
  field
    ledger          : Ledger
    executionStack  : ExecutionStack
    lastCallAddress : Address
    calledAddress   : Address
    nextstep        : SmartContractExec Msg
```

Next we define a function stepEF, which executes one step of the execution of a contract, and a function stepEFntimes, which iterates stepEF n times. stepEFntimes can be regarded as execution with a first very simple form of gas limit (given by n). The types of those functions are as follows (the full definition can be found in the git repository [15]):

```
stepEF : Ledger → StateExecFun
       → StateExecFun
```

```
stepEFntimes : Ledger → StateExecFun
             → ℕ → StateExecFun
```

As an example, we create first the function constantFun, which returns the same number.

```
const : ℕ → Msg → SmartContractExec Msg
const n msg = return (nat n)
```

Constant functions represent variables, where we look up their content by applying them to the message nat 0.

We build now a ledger which at address 1 has a balance 40 and a contract implementing a simple counter. The counter is represented by the variable `"f1"`, and a function `"g1"`, which increments the variable represented by `"f1"` by 1. The function `"f1"` is initialised with the constant function returning 0 (representing a variable initialised as 0). Function `"g1"` looks up the current address (which returns 1), looks up the content of variable `"f1"` by applying it to nat 0. Then it makes an anonymous case distinction on the result (syntax $\lambda\{\cdots\}$): if the result was nat n, it updates `"f1"` to the constant function returning suc n; otherwise it raises an error. For other addresses, the amount will be 0, function names and arguments will return an error with message (`"Undefined"`). The same applies to other functions at address 1:

```
testLedger 1 .amount = 40
testLedger 1 .fun "f1" m = const 0 (nat 0)
testLedger 1 .fun "g1" m =
  exec currentAddrLookupc λ addr →
  exec (callc addr "f1" (nat 0))
  λ{(nat n) → exec (updatec "f1" (const (suc n)))
                   λ _ → return (nat (suc n));
    _       → error (strErr
                "f1 returns not a number")}
testLedger ow .amount = 0
testLedger ow .fun ow' ow''
  = error (strErr "Undefined")
```

C. Complex model of Solidity smart contracts in Agda

This subsection extends the simple model to a more complex one. Similar to the simple model, the complex model has structures and data types such as Msg and Ledger, and functions such as stepEF, stepEFntimes, and ExecutionStack. The complex model has more complex operation commands, such as updating and calling pure functions similar to the Solidity language, where it allows to redefine a pure function by referring to the full previous instance of that function. This allows to model maps, which in Solidity are finite functions from keys to a target type, more directly rather than encoding them as lists of pairs of messages. The complex model also has added gas cost, and a better recording of error messages.

We redefine the elements of the smart contract execution stack (ExecStackEl) by adding 3 more fields:

- costCont, the gas cost for continuation depending on the message returned when the current call is finished;
- funcNameexecStackEl, the last function called;
- msgexecStackEl, the argument with which the last called function was called.

The last two elements are used for displaying debugging information in case of an error.

The definition of ExecStackEl is as follows (we omit the fields defined in the simple model):

```
record ExecStackEl : Set where
  field
    - fields from the simple model
    costCont          : Msg → ℕ
    funcNameexecStackEl : FunctionName
    msgexecStackEl    : Msg
```

Similar to the simple model, we redefine mutually SmartContractExec, CCommands, and CResponse, as follows:

```
data SmartContractExec (A : Set) : Set where
  return : ℕ → A → SmartContractExec A
  error  : ErrorMsg → DebugInfo
         → SmartContractExec A
  exec   : (c : CCommands) → (CResponse c → ℕ)
         → (CResponse c → SmartContractExec A)
         → SmartContractExec A


data CCommands : Set where
- constructors from the simple model
- (excluding updatec)
  callPure : Address → FunctionName
           → Msg → CCommands
  updatec : FunctionName
           → ((Msg → MsgOrError)
           → (Msg → MsgOrError))
           → ((Msg → MsgOrError)
           → (Msg → ℕ) → Msg → ℕ)
           → CCommands
  raiseException : ℕ → String → CCommands


CResponse : CCommands → Set
- equations from the simple model
- (excluding updatec)
CResponse (callPure addr fname msg) = MsgOrError
CResponse (updatec fname fdef cost) = ⊤
CResponse (raiseException _ str)    = ⊥
```

In SmartContractExec, we add to return an extra argument ℕ (natural number). This is the cost for executing the return statement, which depends on the size of the return value. In case of error, we add debug information (DebugInfo), which includes four fields: the address which made the call, the current address, the last function that was called, and the argument with this the function was called. In case of exec, we add the response cost for each command (CResponse $c \to \mathbb{N}$). In the operations command (CCommands), we define two extra commands, which are callPure, which we use to call pure functions, and raiseException for raising an exception. We use as well a slightly different definition of

updatec, which we use to update pure functions and add an extra argument to calculate the pure function cost ((Msg → MsgOrError) → (Msg → ℕ) → Msg → ℕ)).
In the definition of CResponse, we add two more cases. In the case of callPure, it will return a message or error. In the case of raiseException, it is an empty type, since there is no continuation. The case of updatec has different arguments, but returns as in the simple model ⊤. The other commands and responses are the same as in the simple model and the full definition is available at [15].

Also, we redefine the complex implementation of smart contracts (Contract) by adding two extra fields, pure function (purefunction) and cost of executing the pure function (purefunctionCost). Pure functions are as in Solidity functions which don't call other functions (in our setting variables are represented by functions). In Solidity, it does not cost any gas when called externally; if called from an internal function, it will cost gas. Contract has the following additional fields, with the remaining fields defined as in the simple model:

```
record Contract : Set where
  field
    - fields from the simple model
    purefunction      : FunctionName
                      → Msg → MsgOrError
    purefunctionCost : FunctionName
                      → Msg → ℕ
```

In addition, we redefine the state of execution (StateExecFun) for the complex model by adding four more fields:

- initialAddr is the address which initiated the current sequence of calls;
- gasLeft is how much gas we have left in the next execution step;
- funNameevalState is the function name which was called; this will be used as debug information in case of an error;
- msgevalState is the argument with which the function name was called.

The definition of StateExecFun (with the remaining fields as in the simple model) is as follows:

```
record StateExecFun : Set where
  field
    - fields from simple model
    initialAddr     : Address
    gasLeft         : ℕ
    funNameevalState : FunctionName
    msgevalState    : Msg
```

To deal with the gas cost in the complex model, we define deductGas, which we use to deduct gas from the state execution function (StateExecFun), not from the ledger.

```
deductGas : (statefun : StateExecFun)
            (gasDeducted : ℕ)
            → StateExecFun
```

Then, we define stepEFgasAvailable, which shows the gas available in the smart contract code, and stepEFgasNeeded, which determines the gas needed for the execution of smart contract code.

```
stepEFgasAvailable : StateExecFun → ℕ

stepEFgasNeeded : StateExecFun → ℕ
```

We define an auxiliary function stepEFAuxCompare in order to compare stepEFgasAvailable and stepEFgasNeeded:

```
stepEFAuxCompare : (oldLedger : Ledger)
  → (statefun : StateExecFun)
  → OrderingLeq (stepEFgasNeeded statefun)
                (stepEFgasAvailable statefun)
  → StateExecFun
```

stepEFAuxCompare has two cases:

- If the gas available is greater than the gas needed, it will deduct the gas, process the transaction, and update the ledger.
- If the gas available is less than the gas needed, we have run out of gas. It will update the ledger to be the old ledger but with gas deducted, and abort execution reporting an out of gas error.

We create an example of a simple voting contract with gas cost included to demonstrate the complex model code. For the contract itself, we have four fields: amount (amount), function name (fun), pure function (purefunction), and pure function cost (purefunctionCost). For address 1, the amount is 100, and we have three functions (`"addVoter"`, `"deleteVoter"`, and `"vote"`). In addition we have two pure functions (`"checkVoter"` and `"counter"`). The explanation of the three functions is as follows:

- `"addVoter"` updates the pure function (`"checkVoter"`) by setting it to true for the new voter.
- `"deleteVoter"` does the same, but setting it to false for the deleted voter.
- `"vote"` first looks up the calling address and calls the pure function (`"checkVoter"`), to check whether the voter is allowed to vote (where (nat 0) represents false and (nat (suc n)) represents true). Then it calls voteAux to make a case distinction on this decision. If the voter is allowed to vote it increments the counter (pure function (`"counter"`)) by 1. Otherwise it will return an error. The type of voteAux is as follows:

```
voteAux : Address → MsgOrError
        → SmartContractExec Msg
```

voteAux will in case the message (result of checking whether the voter is allowed to vote) represents true delete the voter, lookup the counter, and if it was (nat n) increment it by 1. In all other cases, it raises an error.

The pure function "checkVoter" is initialised to 0, meaning no voter is allowed to vote, and "counter" is initialised to 0. For other addresses, the amount will be 0, and all pure functions and functions not specified before will return an error message ("Undefined") with debugging information As well for other pure functions the costs will be 1. In our contract, for brevity, we have only one candidate to vote for, like in the former GDR. In the git repository [15], we have in addition a more democratic example which allows to vote for multiple candidates (see guidelines.agda).

```
testLedger 1 .amount = 100
testLedger 1 .fun "addVoter" msg
  = exec (updatec "checkVoter"
    (addVoterAux msg) λ oldFun oldcost msg → 1)
    (λ _ → 1) λ _ → return 1 msg
testLedger 1 .fun "deleteVoter" msg
  = exec (updatec "checkVoter"
    (deleteVoterAux msg) λ oldFun oldcost msg → 1)
    (λ _ → 1) λ _ → return 1 msg
testLedger 1 .fun "vote" msg
  = exec callAddrLookupc (λ _ → 1)
  λ addr → exec (callPure addr "checkVoter"
                          (nat addr))
    (λ _ → 1) λ check → voteAux addr check
testLedger 1 .purefunction "counter" msg
  = theMsg (nat 0)
testLedger 1 .purefunction "checkVoter" msg
  = theMsg (nat 0)
testLedger 1 .purefunctionCost "checkVoter" msg
  = 1
testLedger 3 .amount = 100
testLedger ow .amount = 0
testLedger ow .fun ow' ow''
  = error (strErr "Undefined")
      ⟨ ow » ow · ow' [ ow'' ]⟩
testLedger ow .purefunction ow' ow''
  = err (strErr "Undefined")
testLedger ow .purefunctionCost ow' ow'' = 1
```

We have implemented functions which compute the resulting ledger and the result returned after executing a function and functions which compute the result returned by a pure function. These functions are defined recursively. In order to guarantee termination, we add a variable numberOfSteps which is initially set to the gas assigned and counted down in each execution step. Furthermore, we guarantee that the gas used and deducted in each execution step is at least one (technically, we achieve this by adding 1 to the gas specified). We maintain (because gas is reduced by at least 1 in each step) the invariant that the gas left is always $\leq$ numberOfSteps, so when the numberOfSteps is 0, and a execution step is to be carried out, there is no gas left, and one obtains an out-of-gas error. Therefore, the program passes the termination checker of Agda, with all necessary proofs carried out in Agda. See the executed voting example in the file (guidelines.agda) of the git repository [15].

V. Conclusion and Future Work

In this paper, we have presented the first step towards verifying smart contracts in Ethereum using weakest preconditions. This will give a precise meaning to a contract. We have developed smart Solidity-style contracts in two models. The first is the simple model, which includes features such as dealing with simple executions, returning the available balance in each contract, calling other smart contracts, transferring money to other contracts, and looking up the current and calling addresses. The simple model does not include gas costs at this stage. The second is the complex model, which provides additional features, such as gas cost, more complex executions, calling and updating pure functions, and calculating the pure function cost. We have built these models using the interactive theorem prover Agda. Agda is unique that it allows to write programs and verify them in the same language. This avoids translation errors from one program to another.

In future work, we will build an interactive program in Agda which executes simple and complex models. Then, we will verify the simple and complex models of smart Solidity-style contracts using the weakest preconditions [1], [2] to specify the criteria necessary to carry out a particular transfer in a smart contract. The ultimate goal is to have solidity contracts accompanied with a proof that it is correct w.r.t. weakest preconditions and therefore fulfils security guarantees for its execution.

References

[1] F. F. Alhabardi, A. Beckmann, B. Lazar, and A. Setzer, "Verification of Bitcoin Script in Agda Using Weakest Preconditions for Access Control," in *27th International Conference on Types for Proofs and Programs (TYPES 2021)*, ser. LIPIcs, vol. 239. Dagstuhl, Germany: Leibniz-Zentrum für Informatik, 2022, pp. 1:1–1:25, doi: https://doi.org/10.4230/LIPIcs.TYPES.2021.1.

[2] F. Alhabardi, B. Lazar, and A. Setzer, "Verifying correctness of smart contracts with conditionals," in *2022 IEEE 1st Global Emerging Technology Blockchain Forum: Blockchain & Beyond (iGETblockchain)*, 2022, pp. 1–6, doi: https://doi.org/10.1109/iGETblockchain56591.2022.10087054.

[3] Ethereum Community, "Solidity documentation," Retrieved 15 April 2023, Availabe from https://docs.soliditylang.org/en/v0.8.16/.

[4] N. Szabo, "Smart Contracts: Building Blocks for Digital Markets," *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, no. 2, p. 28, 1996, Availabe from https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[5] Zibin Zheng and Shaoan Xie and Hong-Ning Dai and Weili Chen and Xiangping Chen and Jian Weng and Muhammad Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.

[6] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2018, pp. 1–4, doi: https://doi.org/10.1109/ICCCNT.2018.8494045.

[7] A. Setzer, "Modelling Bitcoin in Agda," *CoRR*, vol. abs/1804.06398, 2018, Availabe from http://arxiv.org/abs/1804.06398.

[8] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in *Principles of Security and Trust*. Berlin, Heidelberg: Springer, 2017, pp. 164–186, doi: https://doi.org/10.1007/978-3-662-54455-6_8.

[9] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "Verification of smart contracts: A survey," *Pervasive and Mobile Computing*, vol. 67, p. 101227, 2020, doi: http://dx.doi.org/10.1016/j.pmcj.2020.101227.

[10] OCamlPro SAS, "Welcome to Liquidity's documentation!" Retrieved 15 April 2023, Availabe from https://liquidity-lang.org/doc/.

[11] R. O'Connor, "Simplicity: A New Language for Blockchains," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, ser. PLAS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 107–120, doi: https://doi.org/10.1145/3139337.3139340.

[12] A. M. Antonopoulos, *Mastering Bitcoin: Programming the open blockchain*, 2nd ed. O'Reilly, 2017.

[13] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," Retrieved 03 May 2014, Availabe from https://ethereum.org/en/whitepaper.

[14] Agda Community, "Welcome to Agda's documentation!" Retrieved 03 May 2023, Availabe from https://agda.readthedocs.io/en/v2.6.2/.

[15] A. Setzer and F. Alhabardi, "A model of the Solidity-style smart contracts in the theorem prover Agda," 2023, Available from https://github.com/fahad1985lab/A_model_of_Solidity_style_smart_contracts_in_the_theorem_prover_Agda_.

[16] J. B. Nielsen and B. Spitters, "Smart Contract Interactions in Coq," in *Formal Methods. FM 2019 International Workshops*. Cham: Springer International Publishing, 2020, pp. 380–391, doi: https://doi.org/10.1007/978-3-030-54994-7_29.

[17] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Berlin, Heidelberg: Springer, 2004, doi: https://doi.org/10.1007/978-3-662-07964-5.

[18] Coq Community, "The Coq Proof Assistants," Retrieved 30 Apri 2023, Availabe from https://coq.inria.fr/.

[19] S. Thompson, *Haskell: The Craft of Functional Programming*, 3rd ed. USA: Addison-Wesley Publishing Company, 2008.

[20] OCaml Community, "OCaml– functional programming language," Retrieved 30 Apri 2023, Availabe from https://ocaml.org/.

[21] J. Zakrzewski, "Towards verification of ethereum smart contracts: A formalization of core of solidity," in *Verified Software. Theories, Tools, and Experiments*, R. Piskac and P. Rümmer, Eds. Cham: Springer International Publishing, 2018, pp. 229–247, doi: https://doi.org/10.1007/978-3-030-03592-1_13.

[22] A. Arusoaie, "Certifying findel derivatives for blockchain," *Journal of Logical and Algebraic Methods in Programming*, vol. 121, p. 100665, 2021, doi: https://doi.org/10.1016/j.jlamp.2021.100665.

[23] A. Biryukov, D. Khovratovich, and S. Tikhomirov, "Findel: Secure derivative contracts for ethereum," in *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2017, pp. 453–467, doi: https://doi.org/10.1007/978-3-319-70278-0_28.

[24] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1186–1189, doi: https://doi.org/10.1109/ASE.2019.00133.

[25] W. Ahrendt, R. Bubel, J. Ellul, G. J. Pace, R. Pardo, V. Rebiscoul, and G. Schneider, "Verification of smart contract business logic," in *Fundamentals of Software Engineering*. Cham: Springer International Publishing, 2019, pp. 228–243, doi: https://doi.org/10.1007/978-3-030-31517-7_16.

[26] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, and M. Ulbrich, *Deductive Software Verification – The KeY Book: From Theory to Practice*. Springer Cham, 01 2016, vol. 10001, doi: https://doi.org/10.1007/978-3-319-49812-6.

[27] L. P. A. da Horta, J. S. Reis, M. Pereira, and S. M. de Sousa, "Whylson: Proving your michelson smart contracts in why3," 2020, doi: https://doi.org/10.48550/arXiv.2005.14650.

[28] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 653–663, doi: https://doi.org/10.1145/3274694.3274743.

[29] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 557–560, doi: https://doi.org/10.1145/3395363.3404366.

[30] Vyper Team, "Vyper documentation," Retrieved 15 April 2023, Availabe from https://vyper.readthedocs.io/en/stable/.

[31] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao, "Safer smart contract programming with scilla," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019, doi: https://doi.org/10.1145/3360611.

[32] J. C. Reynolds, "Towards a theory of type structure," in *Programming Symposium*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 408–425, doi: https://doi.org/10.1007/3-540-06859-7_148.

[33] M. Bartoletti, L. Galletta, and M. Murgia, "A minimal core calculus for solidity contracts," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Cham: Springer International Publishing, 2019, pp. 233–243, doi: https://doi.org/10.1007/978-3-030-31500-9_15.

[34] S. Crafa, M. Di Pirro, and E. Zucca, "Is solidity solid enough?" in *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2020, pp. 138–153, doi: https://doi.org/10.1007/978-3-030-43725-1_11.

[35] P. Wadler, "Programming Language Foundations in Agda," in *Formal Methods: Foundations and Applications*. Cham: Springer International Publishing, 2018, pp. 56–73, doi:https://doi.org/10.1007/978-3-030-03044-5_5.

[36] A. Stump, *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, 2016, doi: https://doi.org/10.1145/2841316.

[37] P. Martin-Löf, *Intuitionistic type theory*. Bibliopolis, 1984.

[38] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell, "Verifying Haskell Programs Using Constructive Type Theory," in *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 62–73, doi:https://doi.org/10.1145/1088348.1088355.

[39] A. Bove, P. Dybjer, and U. Norell, "A Brief Overview of Agda – A Functional Language with Dependent Types," in *Theorem Proving in Higher Order Logics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–7, doi: https://doi.org/10.1007/978-3-642-03359-9_6.

[40] N. A. Danielsson and U. Norell, "Parsing Mixfix Operators," in *Implementation and Application of Functional Languages*. Berlin, Heidelberg: Springer, 2011, pp. 80–99, doi: https://doi.org/10.1007/978-3-642-24452-0_5.

[41] T. Altenkirch, J. Chapman, and T. Uustalu, "Relative monads formalised," *Journal of Formalized Reasoning*, vol. 7, no. 1, p. 1–43, Jan. 2014, doi: http://dx.doi.org/10.6092/issn.1972-5787/4389.

[42] Agda documentation, "Compiler," Retrieved 02 Augest 2023, Availabe from https://agda.readthedocs.io/en/v2.6.2.1/tools/compilers.html#ghc-backend.

[43] Marcin Benke, "A compiler for Agda," 2007, Availabe from https://duch.mimuw.edu.pl/~ben/Papers/TYPES07-alonzo.pdf.

[44] Agda Wiki, "MAlonzo compiler," Retrieved 03 May 2023, Availabe from https://wiki.portal.chalmers.se/agda/Docs/MAlonzo.

[45] E. community, "PROOF-OF-STAKE (POS)," Retrieved 03 May 2023, Availabe from https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[46] F. Ritz and A. Zugenmaier, "The impact of uncle rewards on selfish mining in ethereum," in *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2018, pp. 50–57, doi:https://doi.org/10.1109/EuroSPW.2018.00013.

[47] D. Vujičić, D. Jagodić, and S. Ranđić, "Blockchain technology, Bitcoin, and Ethereum: A brief overview," in *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, 2018, pp. 1–6, doi:http://dx.doi.org/10.1109/INFOTEH.2018.8345547.

[48] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Retrieved 03 May 2014, Availabe from https://cryptodeep.ru/doc/paper.pdf.