DOI: 10.15514/ISPRAS-2019-31(3)-13

Overview of the Languages for Safe Smart Contract Programming

[1] *A.V. Tyurin, ORCID: 0000-0003-4820-3678 <a.tyurin@2016.spbu.ru>*
[1]*I.V. Tyulyandin, ORCID: 0000-0002-8429-8726 <i.tyulyandin@2015.spbu.ru>*
[1] *V.S. Maltsev, ORCID: 0000-0002-4948-3248 <v.maltsev@2016.spbu.ru>*
[1] *I.A. Kirilenko, ORCID: 0000-0003-4957-1974 <y.kirilenko@spbu.ru>*
[2] *D.A. Berezun, ORCID: 0000-0001-6306-275X <danya.berezun@gmail.com>*
[1] *Saint Petersburg State University, Mathematics and Mechanics Faculty*
7, University Embankment, Saint Petersburg, 199034, Russia
[2] *Higher School of Economics National Research University, Department of Computer Science*
16 Soyuza Pechatnikov Street, Saint Petersburg, 190121, Russia

Abstract. Blockchain technologies are gradually being found an application in many areas, especially in FinTech. As a result, a lot of blockchain platforms have emerged with the support of smart contracts that are intended to automate party interactions. However, it has been shown that they are prone to attacks and errors which lead to money loss. To date, there has been a wide range of approaches for making smart contracts safer that included analysis tools, reasoning models, and safer and more rigorous programming languages. In this paper, we provide an overview of smart contract programming languages design principles, related vulnerabilities, and future research areas. The provided overview is meant to outline the to date state of languages and to become a possible basis for future proceedings, and show approaches, used by the community, to reach safe and usable language for smart contracts. We have split all found vulnerabilities by source of their arising. Various languages' characteristics such as abstraction level, paradigm, Turing completeness and main features are summarized in the table. Additional information about languages is provided, e.g. model of execution and tools for static analysis.

Keywords: blockchain; smart contracts safety; programming languages

For citation: Tyurin A.V., Tyuluandin I.V., Maltsev V.S., Kirilenko I.A., Berezun D.A. Overview of the Languages for Safe Smart Contract Programming. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 157-176. DOI: 10.15514/ISPRAS-2019-31(3)-13

Обзор языков для безопасного программирования смарт-контрактов

[1] *А.В. Тюрин, ORCID: 0000-0003-4820-3678 <a.tyurin@2016.spbu.ru>*
[1] *И.В. Тюляндин, ORCID: 0000-0002-8429-8726 <i.tyulyandin@2015.spbu.ru>*
[1] *В.С. Мальцев, ORCID: 0000-0002-4948-3248 <v.maltsev@2016.spbu.ru,>*
[1]*Я.А. Кириленко, ORCID: 0000-0003-4957-1974 <y.kirilenko@spbu.ru,>*
[2] *Д.А. Березун, ORCID: 0000-0001-6306-275X <danya.berezun@gmail.com>*
[1] *Санкт-Петербургский государственный Университет,*
математико-механический факультет
199034, Россия, г. Санкт-Петербург, Университетская Набережная, д. 7
[2] *Национальный исследовательский университет «Высшая школа экономики»,*
Департамент информатики
190121, Россия, г. Санкт-Петербург, ул. Союза Печатников, д.16

Аннотация. Технология распределенного реестра блокчейн становится все более популярной и находит применение в различных областях, в том числе и в финансовых технологиях. Многие блокчейн-платформы поддерживают функциональность смарт-контрактов, которые предназначены для автоматизации заключения договоров. Известны примеры, где ошибки или небрежности в коде смарт-контракта приводят к потере активов, например, из-за атаки злоумышленника или непонимания разработчиком особенностей блокчейн-платформы. На сегодняшний день существует множество различных подходов, которые позволяют сделать смарт-контракты безопаснее. Среди них инструменты анализа кода, модели вычислений и семантики языков программирования смарт-контрактов. В этой работе мы приводим обзор языков программирования смарт-контрактов, принципов их построения, а также потенциальные ошибки в программе смарт-контракта. Основная цель этого обзора — рассмотреть текущее на момент написания статьи состояние языков смарт-контрактов и возможные направления для будущих исследований, а также показать подходы, используемые сообществом для создания безопасного и удобного (с точки зрения абстракции) языка. Характеристики множества языков, такие как: уровень абстракции, парадигма, Тьюринг-полнота, проект, где язык используется, инструменты для анализа кода, система ограничения и главные особенности — были рассмотрены и сведены в таблицу. Предоставлена дополнительная информация о языках, например, о модели выполнения. Также мы кратко описали и разделили все найденные нами уязвимости по источникам их возникновения.

Ключевые слова: блокчейн; безопасность смарт-контрактов; языки программирования

Для цитирования: Тюрин А.В., Тюляндин И.В., Мальцев В.С., Кириленко Я.А., Березун Д.А. Обзор языков для безопасного программирования смарт-контрактов. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 157-176 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-13

1. Introduction

Initially, blockchains were designed for cryptocurrency management based on transactions. Further such systems involved *smart contracts* usage to enhance transactions, making them more sophisticated. This enabled to move part of an application logic into the blockchain, thus allowing to provide customizable redeeming conditions [1], develop crowdfunding systems [2], and other applications based on blockchain technology [3]. Fundamentally smart contracts are programmable objects beyond blockchain, intended to represent *automatable*[1] and *enforceable*[2] agreements [4].

Since smart contracts are essentially programs that are executed within the blockchain and written in some programming language, bugs and errors are possible. Erroneous transaction behavior can lead to financial damage. For example, a not-reentrancy of a function has caused $40 million loss [5]. Moreover, due to the immutable nature of the blockchain, it is often impossible to fix a contract

[1] «Automatable» rather than «automated» since parts of an agreement may require some human input.

[2] Enforceable either by law or by tamper-proof computer code.

with a bad[3] behavior that is already on the chain, i.e. contracts are irrevocably committed. One possible approach to detect such unwanted behaviors and minimize the number of vulnerabilities is to provide a way to formalize smart contracts properties and vulnerabilities. It will help to specify vulnerabilities sources and facilitate reasoning about smart contracts.

In [6] provided by IOHK research[4], an ontology that provides a set of basic conceptual primitives is specified. It can be used to construct desired propositions about smart contracts. It is not intended to be the only true ontology, rather the useful one. According to the ontology, blockchain based smart contracts can be considered as computations over blockchain state, that include the changing over time state itself as well as a transition function. And we will further refer to *modality* properties as to relationships between states, possibility or necessity properties that should be maintained throughout transitions.

These concepts allow thinking about smart contract behavior abstractedly over details. For example, consider a Deadline-*dependent Transfer*, a smart contract controlling property transfer between recipients[5]. Only *Recipient 1* may transfer the *Item* during some time interval prior to the deadline, while only *Recipient 2* may transfer the *Item* once the deadline has been passed. A *modality* property can be formulated in the following way. There always should be a blockchain state where at least one system participant who controls the *Item*, being transferred, exists, i.e. the absence of dead states in the blockchain.

Unfulfillment of those properties in blockchain based smart contracts may lead to money loss and malicious attacks. For example, a vulnerable sequence of smart contract library calls in PARITY wallet led to $150 million freezes on wallets [7].

State inconsistency and weaknesses may be caused by a number of different reasons such as blockchain-specific behavior, execution environment bugs, a model of underlying programming language that is not amenable to proof constructions, non-intuitive representation of programs in languages with good models, unintuitive semantics of underlying programming language for people who lack programming experience etc [8]. Also, some modality properties may never be proved because of possible non-termination of a program, which basically depends on a certain programming language.

Thereby, to make smart contracts secure it is desirable to be able to specify the intended behavior and properties that they should fulfill. These properties fulfillment can be provided with machine-checkable proofs and facilitated with more intuitive programming languages accompanied by tools for static analysis and formal verification to reduce the number of errors.

To date, various approaches, languages, and tools have been proposed: extensive type systems and various programming paradigms [9], programming languages that have easily checked termination conditions [10, 11], high-level languages that encourages safer programming via abstractions [12, 13], and intermediate and low-level languages that ease formal verification and compilers development [14–16].

Smart contract programming languages design is influenced by domain ontology, encountered vulnerabilities and ease of reasoning about modality properties. So, in this paper, we concentrate on the incorporation of known approaches used in design and development of smart contracts programming languages, proceed through vulnerabilities and domain-specific concepts that have been considered during design process, provide a classification of current efforts, and emphasize topics for future research.

The paper is organized as follows. Section 2 provides a short evaluation of similar works. Section 3 gives a brief summary of blockchain architecture principles relevant to languages and tools design. Section 4 describes known vulnerabilities of smart contracts, classifying them for future analysis. Section 5 provides a survey of smart contract programming languages and their design ideas and principles, according to known vulnerabilities and blockchain architectures. In section 6 we discuss possible research gaps and future work. Section 7 concludes the paper.

[3] Here, by a «bad» contract behavior, we mean any behavior that is unexpected or undesirable by the contract owner, caused by any reason.

[4] IOHK company is one of the main customers of research in peer-to-peer networks. See https://iohk.io/about/ for details.

[5] Between users or other smart contracts.

2. Related works

Surely this paper is not the only one surveying smart contract programming languages, and we are aware of a couple of similar works. So, [17] provides an overview of smart contracts programming languages, security properties, and verification methods along with some classification of them. However, despite a good coverage, the proposed survey is rather superficial in a sense that it describes languages through the specification of their features, not going deep into design foundations that have provided the features. Another work [18] gives an overview of some distributed ledger systems, smart contracts languages, and technologies that might facilitate safety and performance, or make new applications possible. The paper is not aimed entirely at languages, hence it leaves the description without design foundations and any classification according to whether desirable properties or design principles. [15] also contains some overview of existing languages and their features, but the survey is performed from the perspective of comparison between them and the language proposed in the paper.

In contrast, this work is intended to enhance language coverage, provide foundations and intuition for reasoning, classification of languages, properties, and design fundamentals along with vulnerabilities that have influenced them.

3. Background

Since smart contracts are computations on a blockchain, underlying blockchain protocol basically sets the path for language and tools design. In this section, we review a few protocol details that influence further development of languages. Substantially there are two widespread blockchain architectures on top of which smart contracts are built to date — *UTxO-based* and *account-based* blockchains that allow stateless and stateful smart contracts respectively.

3.1 UTxO

Unspent transaction output, UTxO, model was introduced with the emergence of BITCOIN blockchain. A typical BITCOIN transaction contains a list of inputs that specify the funds that the transaction issuer can transfer and a list of outputs, that represent the way these funds are intended to be transferred. Each output can be used as an input for another transaction. For example, an issuer can set the amount of currency for each output or specify conditions, under which a possible receiver of funds can spend them, also they can specify themselves as the receivers to get so-called change. A set of UTxO consists of all transactions outputs that have not been yet used as inputs.

Redeeming conditions for transaction outputs in BITCOIN are defined with programs written in BITCOIN SCRIPT [10]. These programs describe properties that must be satisfied for the redeemer to be able to use these transaction outputs as their transaction inputs in order to spend the credits. The spender should provide input values to each locking script of referenced outputs of the previous transaction such that all scripts evaluate to value true, e.g. they may provide their wallet address and transaction signature to verify the authority.

Such scripts are stored within transactions and are being maintained only during a transaction, thus they have no state. Further scripts have limited access to blockchain data and essentially they are pure stateless functions of transaction data, i.e. of input parameters. Despite limitations, scripts along with transaction signatures can express complex redeeming conditions such as *multi-signature* payments, deposit providing, escrow, and dispute mediation, access to external data using oracles, time-locks, payment channels, cross-chain atomic trades etc [19]. Throughout the paper, we regard these scripts as stateless smart contracts.

3.2 Account-based blockchains

Account-based blockchains maintain an explicit state throughout transactions. A *state* is a mapping between account addresses and balances. Within these blockchain systems, each transaction is a mapping between the states. Basically, these systems are transaction-based state machines.

ETHEREUM is an example of such a system [20]. In ETHEREUM smart contracts are similar to users' accounts in a sense that they have their own address and a balance. Smart contracts are stored inside the blockchain and essentially these contracts are lists of functions that can be invoked through users' transactions or other contracts messaging. These functions are defined with bytecode of the corresponding execution environment called *Ethereum Virtual Machine*, EVM. Since any smart contract has a balance, it is a stateful function of a data transaction (or a message) and blockchain state, in which the transaction takes place, so they can write to blockchain state or read from it. Contracts state typically involves a stored amount of currency. However, in general, it can have arbitrary persistent storage that is maintained throughout the transitions of the blockchain.

3.3 Preventing the Denial-of-service attacks (DOS)

Despite the underlying blockchain model, smart contracts are computations that are replicated over blockchain via consensus protocol. To prevent DOS-attacks the number of computations for every program representing a smart contract should be restricted beforehand. Restriction mechanism depends on the underlying programming languages properties. One of the main properties in the context of smart contracts is halting, i.e. whether every program that has been written in it terminates or not. BITCOIN SCRIPT program always terminates since language is not Turing-complete and it does not have loops, or recursion, or any other mechanism that provides infinite computations. However, the size of a program also affects the performance of the system behind it. Thus BITCOIN SCRIPT programs are limited by the stack size and number of computationally heavy instructions, i.e. transactions that contain a script that does not satisfy restrictions are rejected.

For programs written in languages that do not guarantee program termination, e.g. EVM bytecode, program execution is limited via a gas system. *Gas* is basically an amount of cryptocurrency specified for contract execution. Fixed units of gas are charged to a miner for every instruction being executed. If the specified amount of gas is expired, execution of the contract stops. Furthermore, EVM contracts also have a limited stack size.

4. Smart contract weaknesses

In this section, programming language-level vulnerabilities that may cause unfulfillment of modality properties and possible mistakes are classified. It is worth to notice, that the most common property arising in distributed systems is that results of computations should be deterministic. While many smart contract programming languages have been designed with determinism in mind, sometimes general purpose programming languages are used for development [21]. A detailed overview of potential risks of non-determinism and causes can be found in [22].

We consider SOLIDITY language for stateful contracts since it is the most popular smart contract programming languages and generally it was one of the first languages that revealed such weaknesses, unfortunately on its own instance. Despite originally being known as unsafe, the language is evolving and to date its compiler is able to warn about code that might misbehave. However, Solidity has provided the foundation for the design of other languages. The most famous errors that have caused contracts failure are DAO [5] and PARITY [23].

SOLIDITY vulnerabilities are classified in the following subsections based on what level they occur on and the reasons that cause them. Code examples of the weaknesses could be found in [20, 24–28]. Possible attacks are discussed in [29]. Also, it is worth to mention, that SOLIDITY is a Turing complete language, meaning that in general fulfillment of particular modality properties cannot be proved, even despite guaranteed termination due to gas limit.

4.1 Block content manipulation

Block of transactions in blockchains is formed by one of the participants who have the ability to influence block content. Thus, careless blocks handling may cause a number of errors.

Front-Running (Transaction-ordering dependence): It is important to be careful of transactions order. For example, Alice has deployed contract with possibility to sell a product and set a price for it. Bob wants to buy the product, and Alice wants to set a higher price. Let's assume, they want to do it at the same time. If Bob's request is the first, Alice loses money. In another case, Bob's transaction can be rejected, or Bob will spend more money than he expected.

Weak sources of randomness: Random values should be deterministic for all nodes in the network due to consensus considerations. One way to get randomness is to use pseudorandom values. Variables of contract, even the private ones, meta-variables of a block, or a hash of a previous and next block cannot be used as a source of entropy. In some blockchains (including ETHEREUM) it is possible to have influence over these variables during the validation process. A pseudo-random value in smart contract code can be predicted by a malefactor. Precalculation can be done via code analysis.

4.2 Contract interaction

A smart contract should be able to interact with other contracts. The following vulnerabilities appear due to the fact that smart contracts cannot rely on each other's behavior.

Unchecked return values for low-level calls: There are three functions to send ether [30] from account to account in ETHEREUM: send() and call() that return false if an error occurs but the transaction execution continues, and transfer( ) that rolls back the transaction in case of error. Low-level functions callcode() and delegatecall() behave in the same way as functions send() and call() . Thus handling of false value of corresponding functions is needed to avoid undesirable behavior of contract. According to Luu et al. [31], 27.9% of smart contracts in ETHEREUM blockchain do not check returned values.

Reentrancy: An external contract can call back functions of a caller contract before the first invocation has finished. It can lead to undesirable recursive function interactions and allow the callee contract to take over the control flow. The example of this vulnerability is a famous DAO smart contract [5].

Callstack bound: A failure may occur when an external call is made, but the program stack has reached its limit. Stack overflow is possible in smart contract languages. In EVM call stack is limited to 1024 stack frames. If the exception is not properly handled by a contract, the malefactor can use it to attack.

4.3 Resource limits

If the smart contract language is Turing-complete, there is a need in metering[6] mechanism to prevent infinite execution. ETHEREUM charges a fee, named gas. Amount of gas is proportional to the number of executed commands by EVM. Every transaction is bounded with the maximum amount of gas as well as blocks.

Infinite loops: Mistakes and misprints in operators usage may keep contracts syntactically correct but strongly affect their logic. For example, writing =+ instead of += in a loop terminating condition may lead to unexpected program behavior and even to an infinite loop. Moreover, in this case, excessive gas consumption may occur. It also includes situations when the number of memory addresses being used is significantly increased, e.g. when the number of elements in a map grows, it becomes too expensive to iterate over it.

[6] Metering is a way to limit and charge the execution of a smart contract.

4.4 Arithmetics

In SOLIDITY arithmetics is available on unsigned integers only and the language does not provide any arithmetic operations check for correctness. This class of mistakes mostly refers to common programmer errors. In the case of smart contracts, they may lead to a huge loss of assets. Thus, it is common to consider them as vulnerabilities in order to attract programmers attention.

Overflow and underflow: These vulnerabilities arise because numbers can have a fixed size. In case of ETHEREUM, maximum value for $uint(uint256)$ is 2^{256-1} and minimum – 0. A programmer has to manually checks overflow and underflow.

Floating points and precision: SOLIDITY does not have fixed and floating point types. Instead, a programmer has to emulate them via integers. All integer divisions are rounded down. Careless handling of such operations may cause unexpected program behavior.

4.5 Storage access

The following vulnerabilities are caused by negligent memory usage and access.

Uninitialized storage pointer: Local structures, arrays, and maps link to storage zero address by default. Using these objects without initialization will lead to overwriting whatever is in zero address.

Write to an arbitrary storage location: A smart contract can store some data and wrong variable assignment can break it. SOLIDITY has reference types. Mistake with references can lead to internal state corruption. If an array index is out of range, the exception will be thrown, and the smart contract will be reverted.

4.6 Internal control flow

This class of vulnerabilities is caused by a complex control flow graph structure and an ability to manipulate it.

Using inherited functions and variables: It is possible to use inheritance in smart-contracts languages with the object-oriented paradigm. SOLIDITY allows multiple inheritance. If several super-classes have a method or variable with the same name, their behavior in sub-class depends on the inheritance order. It could shadow previously defined values or functions and lead to undesirable results.

Using built-in functions: Programmers should be aware of using built-in functions and their behavior. E.g., someone would like to use assertions to check program invariant. SOLIDITY `assert()` function is intended for this purpose. In case of failure, this method throws an exception and does not return the remaining gas. Thus, to check for changing values, such as input data, it is recommended to use `require()` statement which in the same case does transaction rollback and returns remaining gas.

Using deprecated functions: It is not clear what new compiler versions do with deprecated functions. Therefore, it is not recommended to use these objects.

Locked assets: Contracts should provide a way to manage assets. Suppose in the example the contract has a method to take assets but does not have code to give them back. Due to smart contract code immutability in blockchain history, it is impossible to upgrade or fix this contract. It will cause property loss.

4.7 Authorization

Authorization is a major part of a person identification mechanism, designed to verify the permission for actions. Incorrect or insufficient authorization can lead to the following vulnerabilities.

Incorrect initialization: When the smart contract was deployed to a blockchain, it should be initialized. Often initialization contains sensitive operations such as a setting contract's owner. An error in this action may violate the logic of the smart contract. In SOLIDITY, the *constructor* is a special function, which is called once to set the contract's state. In new SOLIDITY versions, constructors are denoted by a special keyword that made the definitions more obvious. But in earlier versions (less than 0.4.22) constructor is just a function with the same name as the class has. Thus, a typo in constructors' name makes it a usual function, which can be called by anybody since default modifier for a function is *public*.

Function default visibility: Incorrect access modifiers usage or a lack of them can lead to undesirable behavior. For example, calling the function that changes the contract owner with public access modifier allows everyone to become its owner. Default modifier for SOLIDITY is public. Thus, it is strongly recommended to explicitly define visibility for all functions and variables.

5. Smart contract languages

In this section, smart contract languages are considered with respect to their main features, paradigms, and common properties such as Turing-completeness, metering mechanism, reasoning, type system, code analyzers, etc. To reduce the number of subsections we have classified languages with respect to their level of usage.

Low-level: These languages are designed for direct execution by the underlying execution environment. Most concepts and principles of formal semantics, computational model, metering, logic for reasoning about programs, and typing are often introduced on that level. Furthermore, to date, smart contracts are mostly stored on the blockchain in low-level bytecode, which imposes suitability considerations. Examples of such languages are BITCOIN-SCRIPT [10], EVM [32], MICHELSON [33].

High-level: Languages with the idea of making the writing of contracts easier for developers via readability and safer high-level syntactic constructs enhanced by a type system that provides machine services abstractions. Safety aspect appears here and refers to the languages ability to guarantee the integrity of these abstractions and abstractions introduced by the programmer using definitional facilities of the language. In a safe language, such abstractions can be used abstractly while in an unsafe language they cannot: in order to completely understand how a program may (mis-) behave, it is necessary to keep in mind all sorts of low-level details such as the layout of data structures in memory and the order in which they will be allocated by the compiler [34]. The semantics of both levels should be considered here[7]. Examples of such languages are SOLIDITY [35], FLINT [12], and LIQUIDITY [36].

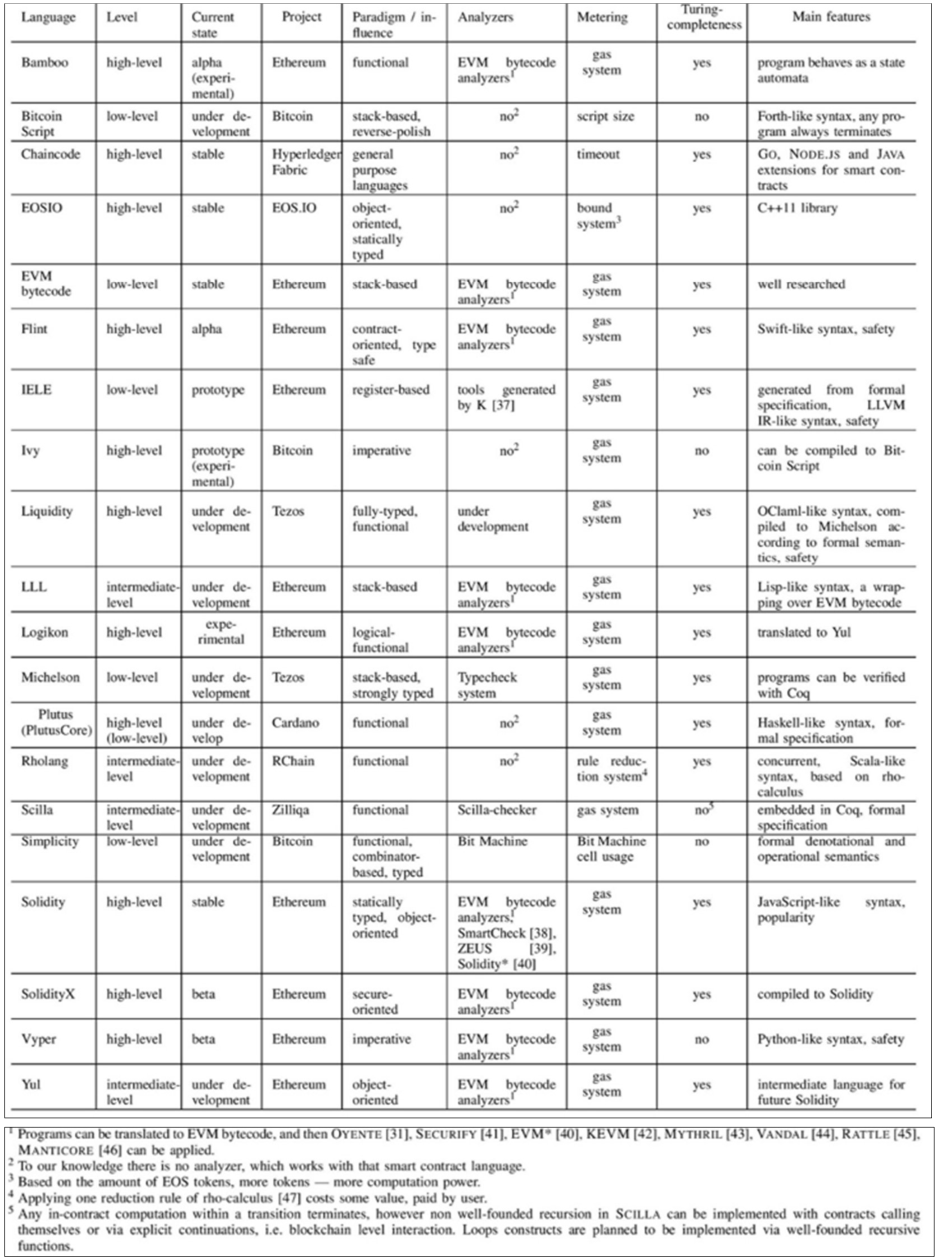
Intermediate-level: Languages that present a compromise between a high-level source and low-level target languages. As a general rule, they are designed in order to simplify program verification or static analysis, relying on the computation model, type system, reasoning, semantics, etc. Furthermore, they allow making unification of compilation, i.e. providing a language that can be compiled for different platforms. SCILLA [15] is an example of such a language.

It is also useful to emphasize some desirable language properties that affect language design.

- Reasoning – language behavior model should allow to specify modality properties and facilitate proving of their (un-) fulfillment. Underlying calculus model and type system are aimed at this.
- Safety – language abstractions should hold integrity property. Rigorous semantics promotes this.
- Expressivity – basically language should be expressive to fit a possible various range of use cases.
- Readability – language representation of a contract behavior should be intuitive, i.e. be easy to inspect and write with.

[7] Fundamentally safeness spreads to other levels since low-level language is an abstraction of its implementation, e.g. a virtual machine.

Table 1. Smart Contract Languages



Language	Level	Current state	Project	Paradigm / influence	Analyzers	Metering	Turing-completeness	Main features
Bamboo	high-level	alpha (experimental)	Ethereum	functional	EVM bytecode analyzers[1]	gas system	yes	program behaves as a state automata
Bitcoin Script	low-level	under development	Bitcoin	stack-based, reverse-polish	no[2]	script size	no	Forth-like syntax, any program always terminates
Chaincode	high-level	stable	Hyperledger Fabric	general purpose languages	no[2]	timeout	yes	GO, NODE.JS and JAVA extensions for smart contracts
EOSIO	high-level	stable	EOS.IO	object-oriented, statically typed	no[2]	bound system[3]	yes	C++11 library
EVM bytecode	low-level	stable	Ethereum	stack-based	EVM bytecode analyzers[1]	gas system	yes	well researched
Flint	high-level	alpha	Ethereum	contract-oriented, type safe	EVM bytecode analyzers[1]	gas system	yes	Swift-like syntax, safety
IELE	low-level	prototype	Ethereum	register-based	tools generated by K [37]	gas system	yes	generated from formal specification, LLVM IR-like syntax, safety
Ivy	high-level	prototype (experimental)	Bitcoin	imperative	no[2]	gas system	no	can be compiled to Bitcoin Script
Liquidity	high-level	under development	Tezos	fully-typed, functional	under development	gas system	yes	OClaml-like syntax, compiled to Michelson according to formal semantics, safety
LLL	intermediate-level	under development	Ethereum	stack-based	EVM bytecode analyzers[1]	gas system	yes	Lisp-like syntax, a wrapping over EVM bytecode
Logikon	high-level	experimental	Ethereum	logical-functional	EVM bytecode analyzers[1]	gas system	yes	translated to Yul
Michelson	low-level	under development	Tezos	stack-based, strongly typed	Typecheck system	gas system	yes	programs can be verified with Coq
Plutus (PlutusCore)	high-level (low-level)	under develop	Cardano	functional	no[2]	gas system	yes	Haskell-like syntax, formal specification
Rholang	intermediate-level	under development	RChain	functional	no[2]	rule reduction system[4]	yes	concurrent, Scala-like syntax, based on rho-calculus
Scilla	intermediate-level	under development	Zilliqa	functional	Scilla-checker	gas system	no[5]	embedded in Coq, formal specification
Simplicity	low-level	under development	Bitcoin	functional, combinator-based, typed	Bit Machine	Bit Machine cell usage	no	formal denotational and operational semantics
Solidity	high-level	stable	Ethereum	statically typed, object-oriented	EVM bytecode analyzers[1] SmartCheck [38], ZEUS [39], Solidity* [40]	gas system	yes	JavaScript-like syntax, popularity
SolidityX	high-level	beta	Ethereum	secure-oriented	EVM bytecode analyzers[1]	gas system	yes	compiled to Solidity
Vyper	high-level	beta	Ethereum	imperative	EVM bytecode analyzers[1]	gas system	no	Python-like syntax, safety
Yul	intermediate-level	under development	Ethereum	object-oriented	EVM bytecode analyzers[1]	gas system	yes	intermediate language for future Solidity

[1] Programs can be translated to EVM bytecode, and then OYENTE [31], SECURIFY [41], EVM* [40], KEVM [42], MYTHRIL [43], VANDAL [44], RATTLE [45], MANTICORE [46] can be applied.

[2] To our knowledge there is no analyzer, which works with that smart contract language.

[3] Based on the amount of EOS tokens, more tokens — more computation power.

[4] Applying one reduction rule of rho-calculus [47] costs some value, paid by user.

[5] Any in-contract computation within a transition terminates, however non well-founded recursion in SCILLA can be implemented with contracts calling themselves or via explicit continuations, i.e. blockchain level interaction. Loops constructs are planned to be implemented via well-founded recursive functions.

Every smart contract language has domain specific instructions or/and types, e.g. cryptographic primitives, assets types, messaging instructions. So we will not emphasize this aspect much. Notable

features and models of several languages with respect to desirable properties are discussed below while a summary of a more expanded set of languages is presented in the table on the Table 1.

5.1 Low-level languages

1) **BITCOIN SCRIPT** is an untyped[8] stack-based low-level language for stateless smart contracts development in BITCOIN and handles transaction verification process. It is intentionally non-Turing-complete with the restricted instruction set where some opcodes are removed e.g. multiplication, division, strings operations, bitwise logic, due to possible overflow vulnerabilities and implementation bugs. Everything is allocated on the stack of limited size words while a program has access to some transaction fields e.g. a hash of transaction data, time field. Thus every program is a pure function of transaction data, i.e. transactions are *self-contained*.

To our knowledge, BITCOIN SCRIPT has no formal semantics, which makes metering ad-hoc and does not enforce formal verification. Furthermore, its stack-based nature and bytecode make smart contracts less auditable since only bytecode is stored inside transactions. Metering is performed via expensive operators counting and script size evaluation. However script's input is arbitrary, hence BITCOIN SCRIPT allows the specification of redemption properties like signature checking, pay-to-public-key-hash, pay-to-script-hash, multisignature checking, and arbitrary data storage inside transactions [1, 10, 48].

2) **SIMPLICITY**: is designed for extending BITCOIN SCRIPT capabilities. It is intended to enhance expressiveness, while enabling static analysis that allows to efficiently bound the number of computations, maintaining BITCOIN SCRIPT design of self-contained transactions, and providing formal semantic to facilitate reasoning about programs. It is anticipated to be used as a compilation target for high-level languages and deployed to *sidechains* [49]. SIMPLICITY is a typed non-Turing-complete combinator-based language with terms based on *Gentzen's sequent calculus*. Every SIMPLICITY type is finite: it contains finitely many values. Hence SIMPLICITY does not support recursive types and can express only finitary functions.

The core of SIMPLICITY consists of nine combinators for term construction with the corresponding denotational semantics. The language is formalized in COQ as well as the correctness of some functions built up from combinators, e.g. *half-adder* or *SHA-256 function*. Generally, the completeness, i.e. the notion that any function between SIMPLICITY types can be expressed with combinators, is verified in COQ.

Further, the operational semantics of SIMPLICITY is defined within the abstract machine called BIT MACHINE, intended to ease bounding of the number of computations, i.e. metering. It is designed to crash at anything that resembles undefined behavior. BIT MACHINE is an abstract imperative machine which state consists of two non-empty stacks of data frames formed by an array of cells. The machine has a set of instructions that manipulate the two stacks and their data frames, and corresponding operational semantics is defined by translating a SIMPLICITY expression into a sequence of BIT MACHINE instructions. It allows computational resources measuring with respect to cells and frames, e.g. the number of executed instructions, copied cells, maximum cells in both stacks at the given point, the number of frames in both stacks. Operational semantics correctness and its correspondence to the denotational semantics are verified in COQ. Furthermore, the set of core combinators can be extended for implementing a signature checking that requires transaction data, thus SIMPLICITY programs can be built to implement the pay-to-script hash scheme [50].

Summarizing, SIMPLICITY stateless nature and rather simple functional semantics without recursion and unbounded loops facilitate equational reasoning, avoiding complex logic. It provides means for formal verification of programs as well as static analysis more capable to effectively bound the number of computational resources. To date SIMPLICITY has a HASKELL implementation under development [51].

[8] More precisely stack operates with byte vectors, which can be interpreted depending on the opcode.

3) **EVM**: is a bytecode language for *Ethereum Virtual Machine*. It is designed to support and execute arbitrary computations over ETHEREUM account-based blockchain, i.e. programs with loops and recursion. EVM is a *stack-based*, Turing-complete machine of 256-bit words with the memory model of word addressed byte array. The machine also has a persisted *storage* which is maintained between transactions and is a part of the blockchain state. It is a word-addressable word array. Program code is separated from data. Access to and modification of data in different types of memory is charged differently from storage — the most expensive to stack and memory being equally charged. The formal execution model and the environment is specified in ETHEREUM Yellow paper [32].

There are efforts on specifying formal semantics for EVM in OYENTE [31], F* [52], KEVM [42], and LEM [53] that focus on formal verification tools and detecting and avoiding insecure features of EVM, e.g. *delegatecall*, *overflows*, *undefined call/return*. Also, the poor human-readability of bytecode is a flaw. ETHEREUM includes many implementations of EVM, e.g. in JAVA SCRIPT, C++, PYTHON, and a promising WEBASSEMBLY implementation [54].

4) **IELE**: is a language defined within *K-framework*[9] [14]. It was designed to overcome EVM drawbacks with an idea of correctness by construction and formal verification in mind. It is intended to be secure and human-readable and to serve as a compilation target for high-level languages, thus unifying compilers construction. IELE is a register-based untyped[10] language: instructions operate on and store their output in an infinite number of virtual registers and have access to a persistent *storage* — the unbounded sparse array of arbitrary-precision signed integers. The language implementation is generated from its formal specification defined in K-framework, which provides generation of verification tools, debugger, interpreter, model checker, etc. IELE has functions and defines a call/return convention where a called function expects a specific number of parameters and returns a specific number of values or corresponding error status[11]. Furthermore, IELE avoids some insecure EVM features, e.g. by introducing *delegatecall* functionality and maintains arbitrary-precision arithmetic. Its operational semantics specifies contracts internal state, blockchain state, and transition rules, i.e. contract's code, intra-contract call stack, remaining gas, and the state of the local memory and virtual registers, storage content, balances, etc. Thus IELE makes formal verification less tedious, enhances human-readability, eliminates undefined, and implementation-defined behaviors, i.e. it is considered to be safe[12].

Gas costs for computation time are based on instructions asymptotic and the gas cost for memory is based on peak memory consumption. Gas model is designed to allow arbitrarily large valued instructions and to avoid artificial limits on the size of data or call stacks while preserving the existing goals of the EVM gas model. However, while arithmetics may cause overflows in EVM, in IELE it may cause out-of-gas exception, starting from some input size. Gas formulas are also specified in K.

5) **MICHELSON** [33, 55]: is a typed stack-based language designed to be on-chain code for stateful smart contracts in TEZOS. It is intended to be a more readable compilation target and more amenable for formal verification.

A MICHELSON program supports high-level types (e.g. *map*, *list*, *set*, etc.) and receives an input stack with parameters and storage being pushed on. It evaluates to a result stack with an output value and new storage or can fail. The language does not support closures in the sense that every function has an empty environment. Messaging with other contracts is performed through passing a storage and not maintains the stack between calls. The types are predefined[13] and monomorphic, further types of input, output, and storage of a contract are fixed and it is statically ensured that resulting storage type is preserved. MICHELSON has a built-in type for cryptocurrency and operations defined for this type are mandatory checked for *underflow/overflow*. Typing is done via types propagation. Due to its computation model, MICHELSON has a straightforward semantics, based on rewriting rules defined on stack and syntax. Also, it defines what is considered as well-typed stacks and the resulting outputs. MICHELSON is currently implemented in OCAML via GADT with an interpreter defined corresponding to the semantics while leaving the type checking to OCAML. It is anticipated to replace current implementation with a one verified with either COQ or F* [56].

6) **PLUTUS CORE:** is a typed language designed for use as a transaction validation language in *UTxO*-based blockchain systems. Fundamentally it is eagerly-reduced higher-order polymorphic λ-calculus extended with iso-recursive types, higher kinds, and a library of basic types and functions, hence it has a straightforward operational semantics. The language is meant to be a compilation target since it is difficult to write and read but it is intended to be formally verifiable in proof assistants.

PLUTUS CORE program is a closed term, and its execution is performed by (possibly non-terminating) reduction of welltyped terms. All types can be normalized and normalization process always terminates. Further, operations on types allow to deal with sized types, i.e. sized integers or bytestrings that allows them to be tracked in the type system to facilitate charging for the appropriate amount of gas and detecting overflows at the type level. The language has a specified abstract machine intended to be amenable for a verification reference implementation. Moreover, PLUTUS CORE has its formal specification defined in K [57, 58].

Transaction validation is performed similarly to BITCOIN SCRIPT. Validation is successful if the PLUTUS CORE program reduces to a non-error value within an allotted number of steps. But it is more extended in a sense that a program has a read-only access to world state passed through a *monad* [59, 60].

PLUTUS CORE is an on-chain language for CARDANO blockchain and is embedded into HASKELL. Furthermore, the blockchain system itself is implemented in HASKELL as well as off-chain computations, e.g. wallets, it allows type checking on the level of the interaction between off-chain applications and on-chain code.

5.2 High-level languages

1) **SOLIDITY**: is a very rich and expressive high-level object-oriented Turing-complete language [35] for writing smart contracts for EVM with a syntax similar to JAVASCRIPT and C++. It has static types, inheritance, libraries, complex user-defined types supporting, and other features. As a consequence, that causes its prevalence as well as a large number of potential vulnerabilities (see section IV).

2) **SOLIDITYX**: is a high-level language [61] which compiles to SOLIDITY. SOLIDITYX is a *secure-oriented* language, which means that it has a defense from some vulnerabilities by default, for example, all access modifiers are *private* by default. However, SOLIDITYX is in beta development now and it is not recommended to be used in production.

3) **VYPER** (aka VIPER): is a high-level language for implementing smart contracts for the EVM [13]. It is PYTHON3 derived programming language. VYPER is an alternative to SOLIDITY that is aimed at code security, clarity, and unambiguity, for example, it excludes constructions that can lead to misleading code. To achieve this VYPER does not support modifiers, class inheritance, inline assembly, function overloading, operator overloading, recursive calling, infinite-length loops,

[9] Framework used to produce implementation derived from formal specifications, based on logic rules.

[10] Arbitrary-precision signed integer is the main datatype.

[11] For reference, in EVM caller sends an arbitrary byte stream containing the call arguments values since functions are represented as a set of JUMP labels.

[12] IELE is stated to be the first real-world language that is designed and implemented using formal semantics, with a zero gap between the formal specification and the implementation.

[13] A programmer cannot define their own types.

binary fixed point. The language also leverages overflow checking, array bounding, and limited state modification.

4) **FLINT**: is a high-level statically-typed contract-oriented language aimed to write robust smart contracts on EVM [12]. FLINT provides a mechanism to specify actors that can interact with a contract, immutability by default, assets types, and safer semantics with overflows causing revert of a transaction and explicit states.

5) **BAMBOO**: is a high-level language compiling to the EVM [62]. Its compiler is implemented in OCAML thus BAMBOO is well amenable to formal verification. BAMBOO creates clarify state transitions and avoids reentrancy problems by default. However, it does not support loops and assignments into storage variables, except array elements, which improves the ability of contracts to be verified but complicates their development.

6) **LOGIKON**: is a high-level logical-functional language compiled to YUL [63]. LOGIKON program represents a set of logical constraints statically and formally verified.

7) **IVY**: is a language [64], designed to simplify programming of stateless smart contracts for BITCOIN. Compare to BITCOIN SCRIPT, in IVY program it is possible to use named variables, named clauses, domain-specific types, syntax sugar for function calls.

8) **LIQUIDITY**: is a functional, statically and strongly typed language, compiled down to MICHELSON. It has OCAML syntax and keeps safety guaranteed by MICHELSON, while providing high-level constraints. LIQUIDITY has a formal specification of the compilation semantics [65] and supports decompilation back from MICHELSON, based on the graph produced by symbolic execution that is eventually transformed into LIQUIDITY *AST*. This feature greatly enhances readability, since stack-based MICHELSON code is rather hard to inspect manually.

9) **CHAINCODE**: is a smart contract program, written for HYPERLEDGER FABRIC [21] blockchain. CHAINCODE can be developed with GO, NODE.JS or JAVA. The code should implement a special interface to interact with the blockchain network. Unlike ETHEREUM smart contracts, CHAINCODE does not have account address or associated assets, but the smart contract can have a mapping of the real assets to the internal state. CHAINCODE has the similar conception to database stored procedures. When a transaction is created, CHAINCODE is called to perform operations according to the transaction data. Possible operations are: read, update or delete data, stored in the ledger. Also, it is possible to invoke or read the state of another CHAINCODE, if the caller has enough permissions.

5.3 Intermediate-level languages

1) **YUL** (JULIA or IULIA): is an intermediate language [66]. It can be compiled to a number of backends: EVM 1.0, EVM 1.5 and eWASM. It is planning to use YUL as an intermediate language in the future versions of the SOLIDITY compiler. YUL can be used for "inline assembly" inside SOLIDITY.

2) **RHOLANG**: is a functional, concurrent, based on *rho-calculus* [47] language [67], used in project RCHAIN. A smart contract in terms of RCHAIN is a process, which has persistent state, its own code, and associated address. Execution of code is done by applying the reduction rule of rho-calculus. RHOLANG has behavioral types [68], reflection, reactive API, asynchronicity. Synchronization primitives for parallel execution of transactions are *messages* and *channels*. Messages are the way to communicate smart contracts with each other, sending values through channels. A user has to pay a cost in special tokens, named *Phlogiston*, to the node in the system for computational resources. These tokens will be used for executing smart contract's code. Rate-limiting mechanism looks like the *gas system* in ETHEREUM. Unlike EVM, where gas metering is done on the VM level, manipulations on *Phlogistons* are injected in smart contract's source code by RHOLANG compiler.

3) **SCILLA**: is intended to be an intermediate level language as a translation target for high-level languages to facilitate program analysis and *verification* before compiling to executable bytecode.

SCILLA is a typed language built on stateful smart contracts, i.e. contracts that have a state represented with a storage and that can communicate either with other contracts via messages or with the off-chain world by raising events or with blockchain explicitly reading blockchain data. The language design is aimed to facilitate formal reasoning providing clear and principled semantics.

Specifically, its semantics is based on communicating automata that separate contract specific computations called *transitions* and blockchain-wide interactions, i.e. messaging with other contracts, thus making transitions atomic. Atomicity is achieved through allowing only tail-calling communications which eliminate reentrancy problems. However non-tail calls are needed for some computations e.g. passing and saving some value back from the callee, it is implemented with explicit continuations mechanism. Nevertheless, possible nonterminating execution can be caused by non-well-founded recursion, which is going to be handled with gas usage. Further, SCILLA specifies pure, i.e. that change the state and impure transitions and those reading blockchain data, e.g. block number with OCAML based syntax.

SCILLA has been shallow-embedded in COQ, specifying such properties as contract terminology, contract state, and transitions along with blockchain states, which allows properties verification in isolation. So its design implies leveraging of formal reasoning to prove different modality properties, e.g. safety[14], liveness[15] or termination for well-founded recursive functions. It is anticipated to enhance support for automating the proofs of safety/temporal properties.

4) **LLL**: is a Lisp-like language [69] for EVM. Main purpose of LLL is to provide a little bit higher level of abstraction upon EVM bytecode, i.e. programmer has more high-level constructions to work with the stack. Also language has more functionality over the base set of EVM opcodes, such as multiary operators (they can be applied to one or more arguments, the result of following code (+ 1 2 3 4 5) is 15), including files, control structures, and macro definitions. LLL has an analog of variables, it makes automatic memory management for saving values.

6. Discussion

We briefly described notable approaches for specification of smart contracts intended behavior and analysis of behavioral properties. However, this survey is nevertheless incomplete. The area of blockchain and smart contracts is under active research. The community tries to apply different approaches and ways in the area of smart contract languages and their execution environments development. Some of them are Turing-completeness, paradigm (e.g. imperative, object-oriented, functional), level of abstraction, a way to limit code execution (metering systems such as ETHEREUM gas, time bounds, number of instructions) and a formal theory on which a language is based.

In the rest of the section, we discuss contributions that have not been classified in previous sections, propose aspects that may worth future researching and related work, and summarize possible pros and cons of provided aspects.

Recall that most smart contracts in blockchains are irreversible, i.e. they are hard to fix once they are deployed. One approach to mitigate this is a design pattern provided in [70, 71] that leverages using *delegatecalls*. It suggests deploying contracts with another *dispatcher* contract. The increased number of messages makes analysis and reasoning more complicated since dispatcher contracts should be robust and safe then. Another approach is platforms that allow *upgradable* contracts [72].

Arguable concept is the representation in which contracts are deployed to a blockchain. Most of the systems included in our survey store on-chain code in some low-level form. Such form hardens auditability, while also may serve as a uniform compilation target. That facilitates the development with different languages. There are platforms where contracts are stored as programs written in high-level safe languages [72]. Another possible approach for this is decompilation from low-level byte

[14] These are invariants that hold through the lifetime of a contract, exposing that nothing should go wrong.

[15] Basically, it states that something should eventually happen.

code to more high level code like in MICHELSON and LIQUIDITY case. However, to our knowledge, only this couple of languages have formalized semantics of compilation, while none of the known works provides the correctness of interpretation and interpretation after compilation at all, i.e. the correctness of the compiler or the commutativity of the implied diagram.

One more problem is a metering system for smart contracts, such as ETHEREUM gas and its analogs. Gas estimation is in general undecidable. It could be useful to find mechanisms to predict gas consumption. Improper estimation may lead to vulnerabilities (e.g. DoS-attacks), or to fails during code execution (e.g. ETHEREUM out-of-gas exception). Gas consumption depends on many factors such as memory usage and blockchain state. Various adaptive methods like type system are already surveyed PLUTUS [58], rigorous semantics with asymptotic analysis as in IELE [14], or dynamic adjustment as adaptive gas cost mechanism in [73] may be promising, as well as methods based on symbolic paths exploration and resource analysis [74, 75]. For example, PLUTUS design of unbounded integers allows metering statically due to its type system, while unbounded integers in IELE allows only dynamic gas evaluation. One may apply techniques like RAML [76]. Gas reducing optimization are also worth considering[16].

Since smart contracts use cases are yet to be researched, it is undesirable to restrict either statefulness of contracts or Turing-completeness of languages they are written in. The compromise between an ability to run arbitrary computations on the blockchain and amenability to reasoning defines future research topics. For instance, in [9] dependent types of IDRIS language are leveraged for writing provable smart contracts, that are compiled down to run on ETHEREUM. Languages based on models, which better describe an interaction between contracts based on message passing may become future research objectives, e.g. languages based on process calculus [77]. Extensive type systems in such systems also worth researching, e.g. behavioral type systems or linear epistemic ones [78]. Type annotating while writing a contract with such languages is often non-trivial as well as robust and safe contracts development in general. There are researches aimed at domains formalizing, e.g. *finances* and at the design of simpler languages that are embedded in some safe language for only domain purposes [79]. Such domain specific languages tend to be visual to ease the development process for non-experts in programming. Approaches aimed at actor's behavior are as well interesting. There is a DSCP contracting protocol for trading proposed in [80]. The protocol was verified using game theory and statistical models, such as Markov decision processes.

There is still another point about properties to consider. It is modality properties formulating, an i.e. specification of such a property a smart contract should satisfy. If the property of unfulfillment can be proved, it would prevent some exploit, e.g. already mentioned DAO. Some such properties can be seen in [81]. It proposes BITML – Bitcoin Modelling Language that leverages process calculus to describe interactions between participants and generate BITCOIN transactions according to symbolic semantics. In [82] EVM is formalized in LEM for modeling smart contracts behavior with some properties defined.

To outline the discussion, it is worth to notice that many researches avoid the infrastructure around the language, i.e. development environments, testing and deployment tools, extensive *API* libraries. However, these are essential components of successful development and a field for a plenty of practical studies, since to date only ETHEREUM has a rather complete infrastructure.

7. Conclusion

As smart contracts platforms are intended to reasonably automate the economy, smart contracts should be safe and robust. In this paper, we have presented an overview the state of art of smart contract programming languages. We have classified weaknesses and vulnerabilities smart contracts are prone to. Languages calculus models, semantics, and type systems have been surveyed as well as other properties according to reasoning, safety, expressiveness, and readability. In the end, we have summarized related work and possible future research topics.

[16] Due to safety considerations, such optimization should be proven to be semantically equivalent. However, we are not aware of any related results.

References

[1] Bitcoin contract. URL: https://en.bitcoin.it/wiki/Contract (Date: 2019-01-30).

[2] Solidity-example-crowdfunding. URL: https://github.com/zupzup/solidity-example-crowdfunding (Date: 2019-01-30).

[3] D. Macrinici, C. Cartofeanu, and S. Gao. Smart contract applications within blockchain technology: A systematic mapping study. Telematics and Informatics, vol. 35, no. 8, 2018, pp. 2337–2354.

[4] C. D. Clack, V. A. Bakshi, and L. Braine. Smart contract templates: foundations, design landscape and research directions. CoRR, vol. abs/1608.00771, 2016.

[5] A 50 million hack just showed that the dao was all too human. URL: https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/ (Date: 2019-01-30).

[6] D. McAdams. An ontology for smart contracts. URL: https://cryptochainuni.com/wp-content/uploads/Darryl-McAdams-An-Ontology-for-Smart-Contracts.pdf (Date: 2019-02-07).

[7] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts sok. In Proc. of the 6th International Conference on Principles of Security and Trust, 2017, pp. 164–186.

[8] G. Destefanis, A. Bracciali, R. Hierons, M. Marchesi, M. Ortu, and R. Tonelli. Smart contracts vulnerabilities: A call for blockchain software engineering. ResearchGate, 2018.

[9] Safer smart contracts through type-driven development. URL: https://publications.lib.chalmers.se/records/fulltext/234939/234939.pdf (Date: 2019-01-30).

[10] Bitcoin script. URL: https://en.bitcoin.it/wiki/Script (Date: 2019-01-30)

[11] R. O'Connor. Simplicity: A new language for blockchains. CoRR, vol. abs/1711.03028, 2017.

[12] Flint. URL: https://github.com/flintlang/flint (Date: 2019-01-30).

[13] Vyper. URL: https://github.com/ethereum/vyper (Date: 2019-01-29).

[14] T. Kasampalis, D. Guth, B. Moore, T. Serbanuta, V. Serbanuta, D. Filaretti, G. Rosu, and R. Johnson. Iele: An intermediate-level blockchain language designed and implemented using formal semantics. University of Illinois, Tech. Rep., http://hdl.handle.net/2142/100320, July 2018.

[15] I. Sergey, A. Kumar, and A. Hobor. Scilla: a smart contract intermediate-level language. CoRR, vol. abs/1801.00687, 2018.

[16] Plutus core specification. URL: https://github.com/input-output-hk/plutus/tree/master/plutus-core-spec (Date: 2019-01-30).

[17] D. Harz and W. J. Knottenbelt. Towards safer smart contracts: A survey of languages and verification methods. CoRR, vol. abs/1809.09805, 2018.

[18] P. L. Seijas, S. Thompson, and D. McAdams. Scripting smart contracts for distributed ledger technology. Cryptology ePrint Archive, Report 2016/1156, 2016, https://eprint.iacr.org/2016/1156.

[19] Contract. URL: https://en.bitcoin.it/wiki/Contract (Date: 2019-01-30).

[20] Ethereum contract security techniques and tips. URL: https://github.com/ethereum/wiki/wiki/Safety (Date: 2019-01-29).

[21] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In Proc. of the Thirteenth EuroSys Conference, 2018, pp. 30:1–30:15.

[22] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun. Potential risks of hyperledger fabric smart contracts. In Proc. of the 2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), 2019, pp. 1–10.

[23] 300m in cryptocurrency accidentally lost forever due to bug. URL: https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether (Date: 2019-01-30).

[24] Smart contract weakness classification and test cases. URL: https://smartcontractsecurity.github.io/SWC-registry/ (Date: 2019-01-29).

[25] Decentralized application security project. URL: https://dasp.co (Date: 2019-01-29).

[26] Security considerations. URL: https://solidity.readthedocs.io/en/latest/security-considerations.html (Date: 2019-01-29).

[27] Vulnerabilities description. URL: https://github.com/trailofbits/slither/wiki/Vulnerabilities-Description (Date: 2019-01-30).
[28] Smart contract weakness classification and test cases. URL: https://smartcontractsecurity.github.io/SWC-registry/ (Date: 2019-01-22).
[29] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts sok. In Proc. of the 6th International Conference on Principles of Security and Trust, 2017, pp. 164–186.
[30] Ether — the crypto-fuel for the ethereum network. URL: https://www.ethereum.org/ether (Date: 2019-01-30).
[31] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 254–269.
[32] D. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. URL: https://ethereum.github.io/yellowpaper/paper.pdf (Date: 2019-01-31).
[33] Michelson language. URL: https://www.michelson-lang.com/ (Date: 2019-01-31).
[34] B. C. Pierce. Types and Programming Languages, 1st ed. The MIT Press, 2002.
[35] Solidity. URL: https://github.com/ethereum/solidity (Date: 2019-01-29).
[36] Liquidity. URL: https://github.com/OCamlPro/liquidity (Date: 2019-01-29).
[37] Grigore Roșu and T. F. Șerbănută. An overview of the k semantic framework. The Journal of Logic and Algebraic Programming, vol. 79, no. 6, 2010, pp. 397–434.
[38] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In Proc. of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, 2018, pp. 9–16.
[39] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In Proc. of the Network and Distributed Systems Security (NDSS) Symposium, 2018.
[40] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In Proc. of the 2016 ACM Workshop on Programming Languages and Analysis for Security, 2016, pp. 91–96.
[41] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 67–82.
[42] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, B. Moore, Y. Zhang, D. Park, A. Stefanescu, and G. Rosu. Kevm: A complete semantics of the ethereum virtual machine. In Proc. of the 2018 IEEE 31st Computer Security Foundations Symposium, 2018, pp. 204–217.
[43] Mythril. URL: https://github.com/ConsenSys/mythril-classic (Date: 2019-01-29).
[44] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. Vandal: A scalable security analysis framework for smart contracts. CoRR, vol. abs/1809.03981, 2018.
[45] Rattle. URL: https://github.com/trailofbits/rattle (Date: 2019-01-30).
[46] Manticore. URL: https://github.com/trailofbits/manticore (Date: 2019-01-30).
[47] L. G. Meredith and M. Radestock. A reflective higher-order calculus. Electronic Notes in Theoretical Computer Science, vol. 141, 2005, pp. 49–67.
[48] Bitcoin weaknesses. URL: https://en.bitcoin.it/wiki/Weaknesses (Date: 2019-01-30).
[49] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, Enabling blockchain innovations with pegged sidechains. 2014. https://blockstream.com/sidechains.pdf (Date: 2019-01-30).
[50] Mediawiki. URL: https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki (Date: 2019-02-5).
[51] Simplicity. URL: https://github.com/ElementsProject/simplicity (Date: 2019-02-5).
[52] Grishchenko I., Maffei M., Schneidewind C. A semantic framework for the security analysis of ethereum smart contracts – technical report (2018). URL: https://secpriv.tuwien.ac.at/tools/ethsemantics. (Date: 2019-01-30).
[53] Formalization of ethereum virtual machine in lem. URL: https://github.com/pirapira/eth-isabelle (Date: 2019-01-30).
[54] Ewasm: design overview and specification. URL: https://github.com/ewasm/design (Date: 2019-01-30).
[55] Michelson: the language of smart contracts in tezos. URL: http://www.liquidity-lang.org/doc/reference/michelson.html (Date: 2019-01-30).
[56] Why michelson? URL: https://www.michelson-lang.com/why-michelson.html (Date: 2019-02-5).
[57] Plutus core semantics. URL: https://github.com/kframework/plutus-core-semantics (Date: 2019-01-30).
[58] Plutus implementation and tools. URL: https://github.com/input-output-hk/plutus (Date: 2019-01-30).

[59] The extended utxo model. URL: https://github.com/input-output-hk/plutus/tree/master/docs/extended-utxo (Date: 2019-02-5).
[60] Is it smart to use smart contracts? URL: https://plutusfest.io/presentations/Philip-Wadler/Wadler30.pdf (Date: 2019-02-5).
[61] Solidityx. URL: https://solidityx.org/ (Date: 2019-01-30).
[62] Bamboo. URL: https://github.com/pirapira/bamboo (Date: 2019-01-30).
[63] Logikon. URL: https://github.com/logikon-lang/logikon (Date: 2019-01-31).
[64] Ivy: Bitcoin smart contracts. URL: https://github.com/ivy-lang/ivy-bitcoin (Date: 2019-01-30).
[65] Çagdas Bozman, M. Iguernlala, M. Laporte, F. L. Fessant, and A. Mebsout. Liquidity: Ocaml pour la blockchain. Journées Francophones des Langages Applicatifs 2018, 2018.
[66] Yul. URL: https://solidity.readthedocs.io/en/latest/yul.html (Date: 2019-01-30).
[67] Rchain and rholang. URL: https://www.rchain.coop/platform (Date: 2019-01-30).
[68] D. Ancona, V. Bono, and M. Bravetti. Behavioral Types in Programming Languages. Hanover, MA, USA: Now Publishers Inc., 2016.
[69] G. Wood. LLL. URL: https://lll-docs.readthedocs.io/en/latest/index.html (Date: 2019-01-30).
[70] Upgradable contract with solidity. URL: https://gist.github.com/Arachnid/4ca9da48d51e23e5cfe0f0e14dd6318f (Date: 2019-01-30).
[71] Proxy libraries in solidity. URL: https://blog.zeppelin.solutions/proxy-libraries-in-solidity-79fbe4b970fd (Date: 2019-01-30).
[72] The pact smart-contract language. URL: http://kadena.io/docs/Kadena-PactWhitepaper.pdf (Date: 2019-01-30).
[73] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. CoRR, vol. abs/1712.06438, 2017.
[74] E. Albert, P. Gordillo, A. Rubio, and I. Sergey. GASTAP: A gas analyzer for smart contracts. CoRR, vol. abs/1811.10403, 2018.
[75] M. Marescotti, M. Blicha, A. E. J. Hyvärinen, S. Asadi, and N. Sharygina. Computing exact worst-case gas consumption for smart contracts. In Proc. of the International Symposium on Leveraging Applications of Formal Methods, 2018.
[76] J. Hoffmann, A. Das, and S. Weng. Towards automatic resource bound analysis for ocaml. CoRR, vol. abs/1611.00692, 2016.
[77] J. Baeten. A brief history of process algebra. Theoretical Computer Science, vol. 335, no. 2, 2005, pp. 131–146.
[78] H. Deyoung and F. Pfenning. Reasoning about the consequences of authorization policies in a linear epistemic logic. In Proc. of the Workshop on Foundations of Computer Security, 2009.
[79] S. Thompson and P. L. Seijas. Marlowe: Financial contracts on blockchain. Lecture Notes in Computer Science, vol. 11247, 2018, pp. 356–375.
[80] G. Bigi, A. Bracciali, G. Meacci, and E. Tuosto. Validation of decentralized smart contracts through game theory and formal methods. Lecture Notes in Computer Science, vol. 9465, 2015, pp. 142–161.
[81] M. Bartoletti and R. Zunino. Bitml: A calculus for bitcoin smart contracts. In Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 83–100.
[82] Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. Lecture Notes in Computer Science, vol. 10323, 2017, pp. 520–535.

Информация об авторах / Information about authors

Алексей Валерьевич ТЮРИН учится на четвёртом курсе на кафедре системного программирования СПбГУ. В его научные интересы входит исследование и создание инструментов для разработки смарт-контрактов и теория формальных языков.

Alexey Valerievitch TYURIN is a fourth-year student at the Department of Software Engineering at St. Petersburg State University. His scientific interests include research and creation of tools for the development of smart contracts and the theory of formal languages.

Иван Владимирович ТЮЛЯНДИН бакалавр кафедры системного программирования СПбГУ 2019 года выпуска. Областями научных интересов являются блокчейны, а также языки смарт-контрактов и их свойства.

Ivan Vladimirovitch TYULYANDIN is a Bachelor of the Department of Software Engineering at St. Petersburg State University (2019). Areas of scientific interest are blockchains, as well as languages of smart contracts and their properties.

Владимир МАЛЬЦЕВ перешел на четвёртый курс на кафедре системного программирования СПбГУ. Исследовательские интересы Владимира: анализ состояния блокчейн-сети и разработка приложений поверх блокчейна.

Vladimir MALTSEV moved to the fourth course at the Department of Software Engineering at St. Petersburg State University. Vladimir's research interests: analyzing the state of the blockchain network and developing applications on top of the blockchain.

Яков Александрович КИРИЛЕНКО закончил кафедру системного программирования СПбГУ, преподает на кафедре с 2006 года, совмещая с исследованиями и работой в коммерческих проектах по темам научных интересов: статический анализ кода, реинжиниринг программных комплексов, технология программирования киберфизических систем.

Jacob Alexandrovitch KIRILENKO graduated from the Department of Software Engineering at St. Petersburg State University. He has been teaching at the Department since 2006, combining research and work in commercial projects on scientific interests: code static analysis, software reengineering, cyber-physical programming technology.

Даниил Андреевич БЕРЕЗУН является кандидатом физико-математических наук. Он защитил кандидатскую диссертацию на тему «Трассирующая нормализация» в марте 2018 года. В настоящее время Даниил является руководителем исследовательской группы метавычислений и распределённых технологий в составе лаборатории языковых инструментов JetBrains Research, а также доцентом в Высшей школе экономики.

Daniil Andreevitch BEREZUN is a PhD in computer science. He defended his PhD thesis on traversal-based normalization in March 2018. Daniil graduated from the St. Petersburg State University, Mathematics and Mechanics Faculty, Department of Computer Science in 2014. Nowadays, he is the head of Metacomputations and Distributed Technologies research group of Programming Languages and Tools Lab in JetBrains Research. He also works as a lecturer at the Department of Computer Science at the Higher School of Economics.