Future Generation Computer Systems 164 (2025) 107563

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Smart contract languages: A comparative analysis

Massimo Bartoletti [a,*], Lorenzo Benetollo [b], Michele Bugliesi [b], Silvia Crafa [d], Giacomo Dal Sasso [d], Roberto Pettinau [c], Andrea Pinna [a], Mattia Piras [a], Sabina Rossi [b], Stefano Salis [a], Alvise Spanò [b], Viacheslav Tkachenko [a], Roberto Tonelli [a], Roberto Zunino [e]

[a] Università degli Studi di Cagliari, Cagliari, Italy
[b] Università Ca' Foscari Venezia, Venezia, Italy
[c] Technical University of Denmark, Copenhagen, Denmark
[d] Università di Padova, Padova, Italy
[e] Università degli Studi di Trento, Trento, Italy

ARTICLE INFO

Keywords:
Smart contracts
Blockchain
Decentralized applications
Cryptocurrencies
Programming languages

ABSTRACT

Smart contracts have played a pivotal role in the evolution of blockchains and Decentralized Applications (DApps). As DApps continue to gain widespread adoption, multiple smart contract languages have been and are being made available to developers, each with its distinctive features, strengths, and weaknesses. In this paper, we examine the smart contract languages used in major blockchain platforms, with the goal of providing a comprehensive assessment of their main properties. Our analysis targets the programming languages rather than the underlying architecture: as a result, while we do consider the interplay between language design and blockchain model, our main focus remains on language-specific features such as usability, programming style, safety and security. To conduct our assessment, we propose an original benchmark which encompasses a wide, yet manageable, spectrum of key use cases that cut across all the smart contract languages under examination.

1. Introduction

Smart contracts have played a pivotal role in the evolution of blockchain technology, paving the way for the emergence of the new paradigm of Decentralized Applications (DApps). As the DApps continue to gain popularity and become pervasive, the complexity of their business logic and the distributed, often open, nature of the underlying platforms over which they execute make their development an increasingly challenging task. In this article, we review the current advances in smart contract languages and assess them to gain fresh insights into their design principles and the impact on the programming practices they convey. Our analysis targets the programming languages rather than the underlying architectures, acknowledging that the design of robust smart contract languages is a prerequisite for a principled development of reliable and secure DApps.

Methodology We start with an analysis of the tiered structure of blockchain platforms: our goal here is to single out the key architectural choices that affect the design and implementation of smart contracts. We then analyze and compare a selection of mainstream smart contract languages, based on an original benchmark we have developed to encompass a wide spectrum of key real-world DApp use cases [1].

To carry out our comparative analysis, we isolate six paradigmatic smart contract languages and their underlying blockchains – Solidity on Ethereum, Rust on Solana, Aiken on Cardano, (Py)TEAL on Algorand, Move on Aptos, and SmartPy on Tezos – as representatives of the permissionless platforms that have become mainstream and have gained widespread adoption in the development of DApps. While some of these platforms exist in different incarnations – e.g. Vyper is an alternative to Solidity on Ethereum, as Ligo is to SmartPy on Tezos and Plutus to Aiken on Cardano – the results of our analysis remain largely consistent across these alternatives. In fact, languages operating on the same platform generally exhibit the same relevant properties relative to the features we target in our assessment, namely security, code readability, and usability.

The focus of our assessment is permissionless blockchains. Permissioned blockchains, in turn, are out of our present interests, as they usually come with general-purpose programming languages in which all the blockchain-specific features are managed within ad-hoc libraries that interact with the underlying blockchain consensus layer [2].

Main contributions Several analyses of blockchain platforms and smart contract languages have appeared in the recent literature (cf. Section 5). One of the distinctive features of our present endeavor, one

* Correspondence to: Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, via Ospedale 72, 09124 Cagliari, Italy.
E-mail address: bart@unica.it (M. Bartoletti).

https://doi.org/10.1016/j.future.2024.107563
Received 18 April 2024; Received in revised form 6 October 2024; Accepted 12 October 2024
Available online 22 October 2024

which sets it apart from previous experiments, is the hands-on nature of our experience with the use cases developed for benchmarking smart contracts. The benchmark itself, "Rosetta Smart Contracts" [1], constitutes a major contribution, in that it encompasses a representative selection of common cases in DApp development that provides a smart contract *chrestomathy*, the initial core of a standard test bed for a qualitative assessment of current and future smart contract languages (and platforms). Experimenting with the implementation of the use cases across the different languages proves very effective to enhance the understanding of the challenges in smart contract design, and of how such design is influenced by the tiered structure of the underlying blockchain. Specifically, we identify the choices made at the *contract layer* (as opposed to the lower, *consensus layer*) as the most influential for the design and the relevant properties of the overlying smart contract languages. At the contract layer, the blockchain is best understood as an asset-exchange state machine, where transactions activated by smart contract rules contribute to a state transition by either creating new assets or exchanging assets among users. Based on this view, we propose a categorization of smart contract languages based on the distinction between the two main models incarnating an asset-exchange state machine: the *account-based model* and the *UTXO model*. This perspective sheds new light on the interplay between the blockchain data and computational models on the one side, and the design principles of smart contracts on the other side.

Our analysis also emphasizes the relevance of adequate language support for the key aspects of smart contract design: assets management, contract-to-contract interactions, and costs. Specifically, tailored type-level abstractions for creating, exchanging and operating with assets are a fundamental ingredient in preventing common errors and vulnerabilities such as asset loss, double spending, or unauthorized transfers. On a different, but related account, native support for certain functionalities of the underlying platform (e.g., custom tokens) is pivotal for key properties in security as well as in efficiency.

Structure of the paper We start in Section 2 with an overview of smart contract platforms. Besides serving to set a common terminology for the analysis, this section also highlights how the basic choices at the contract layer influence smart contracts development, security and performance. We demonstrate this by discussing the (pseudo-code) implementation of a common use case in the account-based model (both in its stateful and stateless incarnations) and for the UTXO model. In Section 3 we take a brief tour of the six smart contract languages in our selection, discussing their main features. The core of the paper is Section 4, where we develop our hands-on comparative analysis. In Section 5 we contextualize our contribution in the scientific literature. Finally, in Section 6 we conclude with a discussion of the key insights derived by our analysis.

2. Smart contracts on blockchains

Blockchain smart contracts are best understood as collections of executable rules that are triggered by user ***transactions*** to activate the exchange of assets and other forms of interaction between users. The underlying architecture is a tiered structure comprising two main layers[1] both of which influence the way smart contracts are programmed, their efficiency and the security properties they convey. Below, we outline the key aspects of the design of smart contract languages in relation to the distinguishing features of this layered architecture.

[1] Blockchains are typically described as comprising more layers, including, from the bottom up, *network, consensus, data* and *application* [3]. The two-tier representation we adopt allows us to isolate the aspects that are relevant to our present focus on smart contracts and smart contract languages.

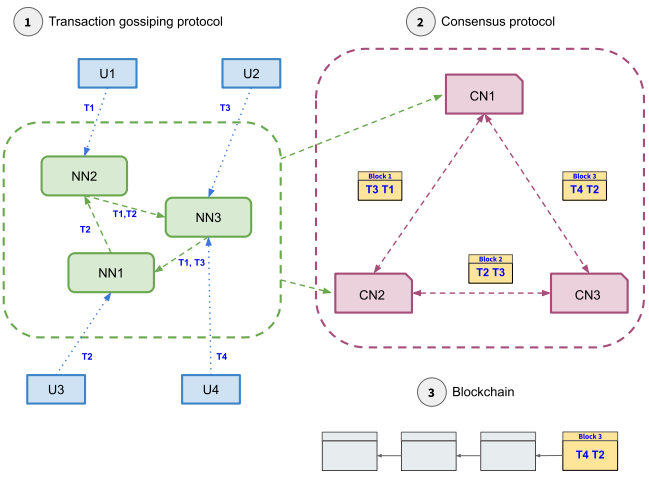


Fig. 1. Life cycle of transactions. The blue, green and red boxes represent, respectively, the users submitting transactions, the networking nodes and the consensus nodes of a blockchain. In Section 1 of the figure, the users create transactions and transmit them to some networking node (NN); the networking nodes, in turn, run a gossiping protocol to share the knowledge of the received transactions. The mempool (dashed green container) is a distributed data structure that represents this shared knowledge of transactions. In Section 2, we see the consensus nodes (CN) collect the transactions from the mempool, propose blocks of transactions (the yellow boxes), and gossip them to the other consensus nodes. Section 3 shows the blockchain extended with the new block selected by the consensus nodes.

2.1. The consensus layer

The consensus layer rests on the data and network services provided by the underlying infrastructure and sets the rules for participation in the blockchain platform. The rules vary from platform to platform, but generally include a protocol for propagating transactions across the networking nodes, and a consensus protocol for ordering the transactions and grouping them into blocks. In the transaction gossiping protocol (part 1 of Fig. 1), the networking nodes broadcast the transactions they received from users, collecting them into a distributed data structure (called *mempool*).[2] In the consensus protocol (part 2), the nodes select a set of transactions from the mempool, and order it into a *block* that they propose to the other consensus nodes. The consensus nodes then run a protocol to choose, among the proposed blocks, which one will be the next block in the sequence of blocks constructed so far — the so-called *blockchain*. Once the consensus nodes reach an agreement on one of the proposed blocks, the chosen block is cryptographically linked to the previous ones (e.g., the new block contains the hash of the previous one), effectively making it part of the blockchain (part 3).

At the consensus layer, the blockchain can be seen as a global state machine whose state (replicated at all consensus nodes) is the blockchain, and the state transitions coincide with (the steps that contribute to) the additions of new blocks.

2.1.1. Key properties and incentives

The consensus layer must guarantee three key properties: *safety* (honest nodes have the same view of the blockchain), *liveness* (new transactions are regularly added to the blockchain), and *finality* (the transactions added to the blockchain are never reverted). In permissionless blockchains, our focus in the present paper, these properties must be enforced without assuming any specific notion of trust among nodes, except that the majority of resources (computational or financial) is controlled by *rational* nodes that participate in the protocols for profit. Consequently, the consensus protocols must be resistant to *Sybil attacks*,

[2] A few blockchain platforms (e.g., Hedera and IOTA) deviate from this design pattern, avoiding the transaction mempool.

making sure that artificially crafting new nodes does not give more than a negligible advantage to the adversary. Such attacks are mitigated by providing economic incentives to honest nodes that play by the rules. In addition to block rewards, these incentives come in the form of ***fees*** that depend on various factors, e.g. the amount of work needed to execute a transaction, the size of the allocated storage, and the pace at which a transaction is included in a block. To avoid incurring higher costs than needed, developers must adopt design patterns that reduce the amount of on-line computations and on-chain storage in favor of their off-chain counterparts.

2.1.2. Transaction ordering

Most consensus protocols leave the participant nodes free to choose which transactions from the mempool to include in a block, and in which order. As a result, such protocols provide no guarantee of a *fair ordering* [4] on how transactions are processed. This, in turn, may open the door to attacks against contracts whose logic depends on the order in which their triggering transactions are processed: e.g., a user may send a transaction to reveal the solution to a bounty contract, while another user front-runs that transaction to win the bounty. Some blockchain platforms are systematically targeted by these attacks, which have detrimental effects on decentralization, transparency, and trustworthiness [5,6]. From the point of view of developers, transaction-order dependence could be mitigated, in principle, by crafting contracts so that any transaction can be executed in exactly one state. In practice, doing so would create an unacceptable congestion effect in high-bandwidth contracts, like e.g. those used in DeFi. More effective forms of mitigation are possible through ad-hoc protocols [7].

2.2. The contract layer

The contract layer sits on top of the consensus layer and hosts the execution environment for smart contracts. Whereas at the consensus layer we see the blockchain as a state machine whose state transitions correspond to the additions of new blocks, at the contract layer what we observe is the execution of each transaction, i.e. the smart contract rules it activates and their interaction with the environment. As a result, though smart contracts may be programmed to perform arbitrary tasks, especially in Turing-complete languages, at the contract layer the blockchain is best understood as an asset-exchange state machine in which the state keeps track of the asset balance for each user, and every transaction contributes to a state transition by either creating new assets or exchanging existing assets among users. Smart contracts and smart contract languages may be classified accordingly, based on the model they adopt for representing the balance state and the accounting of assets.

2.2.1. Accounting models

Two main models have emerged so far: *account-based* and *UTXO* models.[3] The former was first introduced by Ethereum and then adopted or revisited by other mainstream blockchains, including e.g. Solana, Avalanche C–Chain, Aptos, Hedera, Algorand and Tezos. The latter was introduced by Bitcoin, and then extended by Cardano and IOTA.

Account-based model In the account-based model, the blockchain state stores the deployed contracts and keeps track of the asset balance (henceforth the *balance state*) as a map that associates each account with the amount of assets the account owns. Accounts come in two types: user accounts and contract accounts, each equipped with a balance. Transactions update the balance state by either deploying a new (user or contract) account or changing the account-balance map: an asset-transfer transaction is enabled only if the sender account owns all the assets to be transferred. In general, in the account-based model a transaction specifies (i) the users who have signed the transaction, (ii the receiver account (in case it is a contract account, the transaction includes the function to be invoked and its arguments), (iii) the amounts of assets to be transferred from the signers to the receiver, and (iv) the transaction fee.

To illustrate, a `Bank` contract handling deposits and withdraws would be structured as in the following pseudo-code:

```
contract Bank {
  var accounts          // map (user => asset)
  deposit() {
    expect [k]=tx.signed // tx is signed by k
    v = tx.from(k) // tokens sent to contract
    accounts[k]+=v           // trace transfer
  }
  withdraw(amnt) {
    expect [k]=tx.signed
    require accounts[k]>=amnt
    send(amnt,k)             // transfer to k
    accounts[k]-=amnt        // trace transfer
  }
  getTotalBalance() {
    return balance         // contract balance
  }
}
```

The contract uses the local (persistent) `accounts` variable, a key–value map that keeps track of the amounts deposited and withdrawn: this map provides the code-level representation corresponding to the underlying cryptocurrency balances associated with the `Bank` contract and its users' accounts. Once the contract is created, the `deposit` and `withdraw` actions operate at two levels: on the `accounts` map and on the underlying balances of the accounts involved in the transaction. The `accounts` map is updated explicitly by the contract code, while the underlying balances are updated implicitly by the runtime. A user `A` willing to deposit cryptocurrency tokens may do so by signing a transaction with receiver `Bank` that invokes `deposit`. Executing the action checks that the amount is authorized by the signer, *automatically* subtracts the specified amount of tokens from `A`'s account (assuming that there are enough), and adds an equal amount to the balance of the `Bank` account (noted `balance` in the pseudo-code). The `withdraw` method, in turn, allows anyone to withdraw from the `Bank`, provided that they have previously deposited enough tokens. If so, `send(amnt,k)` removes `amnt` tokens from the contract balance and adds them to the balance of the signer's account, updating the `accounts` map accordingly.

UTXO model Unlike the account-based model, the UTXO model makes no reference to any explicit notion of account balance in its representation of the contract-level blockchain state. Instead, the balance for each user is traced implicitly by the *inputs* and *outputs* carried along within the executed transactions. Transaction *outputs* include an amount of assets and a script that specifies the *spending condition* for these assets, i.e. the condition stating how they can be *redeemed by* (i.e. unlocked to be transferred to) another transaction. Transaction *inputs*, in turn, are references to unspent (i.e. yet to be transferred) outputs of previous transactions, and provide data and the *witnesses* to *unlock* (i.e. validate) the spending conditions of the referenced output. In other words, each new transaction spends outputs of previous transactions, and produces new outputs that can be consumed by future transactions. Each unspent output can only be consumed once, as a whole, by exactly one input. Then, the blockchain state is encoded as the set of unspent transaction outputs: the balance for each user is the sum of all the unspent outputs

[3] Given that there appears to be, as yet, no standard terminology for these concepts, we adopt naming schemes that we believe are best suited to render the underlying concepts and help grasp the key features of the existing platforms.

within the transactions that can be redeemed by the user (i.e. those directed to the public keys the user controls).

To illustrate, the transaction below has a single unspent output holding one token, noted 1:T. Its script requires the redeeming transaction (rtx) to include A's signature in its signers list:

T_1
...
out[0]: script = A in rtx.signed value = 1:T

To spend T_1's single output (noted T_1.out[0]), a redeeming transaction must refer to T_1 from its inputs and validate the script by having A as (one of) its signers. This is accomplished by the transaction T_2 below:

T_2
in[0]: out = T_1.out[0]
signed = [A]
out[0]: script = owner in rtx.signed and rtx.out[0].script == script and rtx.out[0].value == value data = {owner:A} value = 1:T

T_2 is signed by A and its script checks that (i) the redeeming transaction is signed by the user stored in the `owner` field of the current transaction; (ii) the script and the value in the redeeming transaction are the same as in T_2. Note that, although any transaction redeeming T_2 must preserve its script and value, it can change the `owner`. In a sense, the script implements a non-fungible token (NFT): to change the ownership of the NFT, the current owner must spend the output with a new transaction (signed by herself) that specifies the new owner.

Account-based vs. UTXO We can compare the two accounting models along two main dimensions: (i) the design patterns induced by their representation of the contract-level state, and (ii) their interaction with the underlying consensus protocols.

At the design level, the account-based model is typically perceived as more intuitive and friendly, as it rests on programming concepts that are familiar to developers. Simply, contracts are standalone modules collecting executable services to be invoked by the users via transactions that operate on the assets kept in their accounts. In the UTXO model, instead, assets and contracts are interdependent, the latter acting as guards for the former, both embedded within transactions with no explicit reference to any notion of user account. In the simplest incarnations of the UTXO model (such as the one in the previous example), each unspent output is managed by the associated script in the transaction. More complex scripts are also at the avail of programmer, to express transactions that consume multiple unspent outputs and create multiple new outputs. Still, the resulting programming practice remains somewhat cumbersome (cf. Section 2.3 for a comparison on a concrete example, and Section 4.2 for a discussion of actual smart contract languages).

As to the interaction with the underlying consensus layer, the two models have trade-offs. On the one hand, UTXO models are exposed to liveness failures, as triggering a transaction may get stuck because all the referenced UTXOs are spent by other transactions. The resulting UTXO congestion effect, occurring when multiple transactions try to spend the same output, represents a non-trivial challenge for developers, especially for high-bandwidth contracts such as, e.g., Decentralized Finance (DeFi) protocols [8,9]. On the other hand, the account-based model appears weaker in that it is exposed to transaction-ordering attacks. As we said earlier (cf. Section 2.1.2), given that the balance state is updated only when transactions are committed, account-based models leave transaction senders with no means to predict whether, when and in which balance state their transactions are executed. The resulting effect, known as *transaction-ordering dependence*, is troublesome as it opens the door to a variety of security attacks [10]. In blockchains where the consensus protocol does not guarantee fair transaction ordering, such attacks are carried out systematically by colluding consensus nodes, which leverage the economic incentives of contracts to extract value from user transactions [5,6]. A further class of attacks exploit the dependency on transaction ordering to alter the contract execution flow and, consequently, the transaction fees. In the UTXO model, instead, a transaction can be executed in exactly one state, given by the UTXOs in its inputs. As a result, UTXO scripts do not have any dependency on transaction ordering, nor do they incur in attacks exploiting transaction fees.

2.2.2. Contract storage models

While the choice of the accounting model is certainly the classification dimension for contract languages, another aspect that is worth emphasizing is the way that smart contract languages account for a notion of *persistent* contract state. Smart contracts often require some kind of memory to keep track of data and information that should persist across multiple executions. In programming language jargon that memory would be called *state*, but to avoid confusion with other notions of blockchain state we refer to it as (*contract*) *storage*. Account-based models typically encompass *stateful* contracts, which encapsulate the storage directly with themselves as the `accounts` map in the `Bank` contract. Notable exceptions are Aptos and Solana, in which the contract storage is held in separate data structures (e.g., accounts) and referenced from the contract. In UTXO models, instead, contracts are by default *stateless* scripts that are discarded once their associated output is spent. A stateful form of UTXO contract may still be accounted for, however, by using the transaction fields as storage, and requiring the spending and redeeming transactions to contain the same contract with updated data in transaction fields (cf. the UTXO version of the bet contract in Section 2.3)

2.3. Exemplifying smart contracts at work: a bet contract

We conclude this overview with a more extensive example that illustrates the different design patterns in the account-based and UTXO models. We use again pseudo-code to show the core concepts and stay away from the specific features of the different blockchains and contract languages.

The contract involves two players who can join the bet by depositing 1 unit of token T each. When the players join, they choose an oracle who will determine the winner, and set a deadline to close the bet 1000 blocks after the one where the join occurred, at the latest. When the oracle announces the winner, the winner can redeem the whole pot of 2:T; if instead the oracle does not choose the winner by the deadline, then both players can redeem their bets, withdrawing 1:T each.

Fig. 2 shows the stateful version of the account-based contract. The stateless version in Fig. 3 follows the same design with the difference that the contract variables must be stored in a separate account, owned by the contract, and accessed from within the contract with a reference to that account that is passed as an argument to all the contract methods.

The UTXO contract, in Figs. 4 and 5, draws on very different design principles. The $T_{\texttt{init}}$ transaction constructs the contract script `UTXO_Bet`, which is then passed unchanged to $T_{\texttt{tjoin}}$ along with an updated `data` field. The transactions $T_{\texttt{join}}$, $T_{\texttt{win}}$ and $T_{\texttt{timeout}}$ in Fig. 4, in turn. act as the activating actions for the contract rules corresponding to the account-based methods. Finally, T_A and T_B represent the players' bets. The script `UTXO_Bet` ensures that (i) the contract is preserved when spending $T_{\texttt{init}}$ with $T_{\texttt{join}}$, (ii) the storage is updated correctly, (iii) and the terminal transactions $T_{\texttt{win}}$ and $T_{\texttt{timeout}}$ correctly transfer the funds from the contract to the players.

2.4. Cross-chain interactions

DApps can span across multiple blockchains, making it possible the exchange of different native crypto-assets [11–13]. In general, cross-chain interactions presuppose a communication layer (e.g., a decentralized bridge systems), and a consensus-agnostic communication protocol (e.g., CCIP from Chainlink). Cross-chain interactions are

```
contract STFUL_ACCT_Bet  {
  var p1,p2,oracle,deadline; // storage
  join(o) {
    require balance==0:T  // init condition
    expect [k1,k2]=tx.signed
    require tx.from(k1)==1:T // get 1:T from k1
         && tx.from(k2)==1:T // get 1:T from k2
    // at this point, balance==2:T
    p1=k1; p2=k2; oracle=o
    deadline=blockH+1000 // block height + 1000
  }
  win(winner) {
    expect [o]=tx.signed
    require o==oracle && balance==2:T
    require winner==p1 || winner==p2
    send(2:T,winner)
  }
  timeout() {
    require blockH>deadline && balance==2:T
    send(1:T,p1); send(1:T,p2)
  }
```

Fig. 2. Account-based contract: *stateful code*. The players start the contract by calling `join`, which requires them to deposit `1:T` each and to set an oracle. The first condition ensures that `join` is the first action triggered: the (system controlled) variable `balance` is initialized to 0 and automatically updated by the transaction (referenced to by `tx`) invoking the `join` method. The `expect` clause requires that the transaction is signed by exactly two keys, and binds them to `k1` and `k2`. The next condition requires that each player deposits `1:T` in the contract along with the call: namely, executing `join` removes `1:T` from the accounts of both players, and adds `2:T` to the contract balance. Finally, the players and the oracle identifiers are recorded in the contract storage together with the deadline. The `win` action transfers `2:T` to the winner, chosen between the two players by the oracle, who is the only possible caller. Both players can call `timeout` after the deadline to redeem their bets.

```
contract STLESS_ACCT_Bet {
  join(s,o) {
    require owns(STLESS_ACCT_Bet,s)
    require s.balance==0:T
    expect [k1,k2]=tx.signed
    send(k1,1:T,s); send(k2,1:T,s)
    s.p1=k1; s.p2=k2; s.oracle=o
    s.deadline=blockH+1000
  }
 win(s,winner) {
    require owns(STLESS_ACCT_Bet,s)
    expect [o]=tx.signed
    require o==s.oracle && s.balance==2:T
    require winner==s.p1 || winner==s.p2
    send(s,2:T,winner)
  }
  timeout(s) {
    require owns(STLESS_ACCT_Bet,s)
    require blockH>s.deadline && s.balance==2:T
    send(s,1:T,s.p1); send(s,1:T,s.p2)
  }
}
```

Fig. 3. Account-based contract: *stateless code*. Each method takes an extra parameter `s`, that is an account to store the contract state: deploying an instance of the contract requires to generate a new account to store its state. The condition `owns(STLESS_ACCT_Bet,s)` ensures that the store is controlled by the contract: if not, it would be easy for an adversary to execute a contract action in an illegal state, subverting the contract rules. Unlike in the stateful version of the contract, inbound tokens are not passed along with the contract call, but are rendered as explicit `send` actions. The owners of the accounts where these tokens are taken from must authorize the transfer, by signing the transaction (as done in the `join` method).

important, but clearly out of scope for our comparison of smart contract languages. That said, a special mention is in order for *native* cross-chain architectures, as they may be seen as an alternative to smart contracts in the DApps paradigm. In fact, such architectures are designed to host

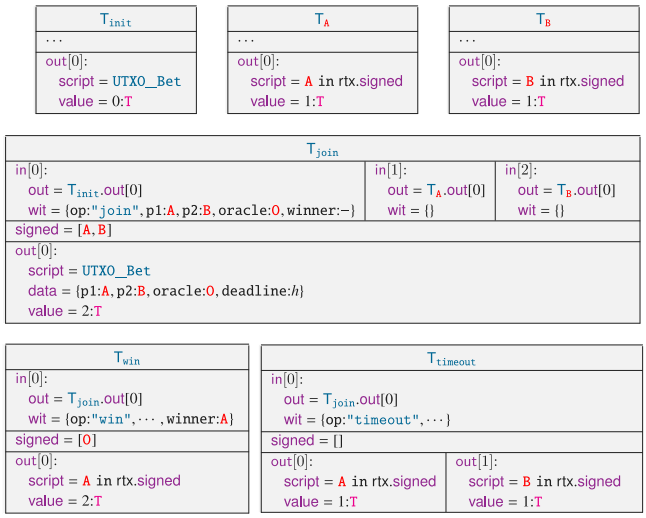


Fig. 4. Transactions for the UTXO-based bet contract. T_{init} creates the contract: the script `UTXO_Bet` is specified in Fig. 5. Players join the contract by sending T_{join}, which spends T_{init} (preserving its script), and the two `1:T` bets provided by the players. Note that both players must sign T_{join}, and so they must agree on the values of the witnesses `p1`, `p2` and `oracle`. T_{join} records these witnesses in its storage, as well as the `deadline`. T_{join} can be spent either by T_{win} or $T_{timeout}$, which terminate the contract. T_{win} must be signed by the oracle, and can be spent only by the `winner` set in the witnesses. In the figure, we assume that the winner is `A`, and accordingly the script of T_{win} requires that the redeeming transaction (rtx) is signed by `A`. $T_{timeout}$ requires no signatures, and it splits `2:T` in two outputs of `1:T` each, that can be spent by the two players. The script `UTXOBet` in T_{join} ensures that the transactions T_{win} and $T_{timeout}$ are constructed according to these rules.

multiple, application-specific blockchains, each tailored for a given use case, and communicating through specific protocols. In other words, having multiple blockchains each running a single contract is as an alternative to deploying multiple contracts on a single blockchain.

Notable cases of native cross-chain architectures include Cosmos and Polkadot. In Cosmos, application-specific blockchains are called *Appchains*, and the interaction among different contracts is rendered as inter-chain communication over the IBC protocol. The protocol manages specific operations such as the transfer of tokens both between accounts in the same Appchain and across accounts operating on different Appchains. Appchains may be programmed in GoLang (a general-purpose programming language), and CosmWasm (a Rust derivative). Communication with blockchains external to Cosmos relies on bridges that support IBC (such as Gravity). In Polkadot, the application-specific blockchains are called *parachains*, implemented through the Substrate framework with its native smart contract language ink! (again a Rust derivative). Parachains communicate via the *Cross-Consensus Messaging* (XCM) language over the transport layer provided by the Polkadot network. XCM is designed to be used outside of Polkadot as well, but requires the implementation of a dedicated bridge.

3. A tour of smart contract languages

We overview in this section the main features of our selection of smart contracts languages.

3.1. Solidity/Ethereum

Solidity is one of the first contract languages, dating back to 2014, and it is currently the main high-level contract language for the blockchains that support the Ethereum Virtual Machine (EVM), i.e. Ethereum, Avalanche C–Chain, and Hedera among the others. Solidity contracts must be compiled to EVM bytecode in order to be executed by the consensus nodes of these blockchains.

```
contract UTXO_Bet { // Stateful UTXO-based
  expect [op,p1,p2,oracle,winner]=rtx.in[0].wit
  if op=="join":
    // rtx must be signed by p1,p2 to redeem 1:T
    require rtx.signed==[p1,p2]
    require value==0:T
    expect [ri0,ri1,ri2]=rtx.in
    require ri0==ctxo // Tjoin.in[0]=Tinit.out[0]
    require ri1.value==1:T // p1's bet
    require ri2.value==1:T // p2's bet
    expect [ro0]=rtx.out   // Tjoin has 1 output
    require ro0.script==script // preserve script
    require ro0.value==2:T
    require ro0.data.p1==p1 // set p1
    require ro0.data.p2==p2 // set p2
    require ro0.data.oracle==oracle // set oracle
    require ro0.data.deadline==blockH+1000
  elif op=="win":
    require rtx.signed==[data.oracle]
    require len rtx.in==1 // Twin has 1 input
    require winner in [data.p1,data.p2]
    expect [ro0]=rtx.out
    require ro0.script=="{winner} in rtx.signed"
    require ro0.value==2:T
  elif op=="timeout":
    require blockH>data.deadline
    require len rtx.in==1 // Ttimeout has 1 input
    expect [ro0,ro1]=rtx.out // ... and 2 outputs
    require ro0.script=="{data.p1} in rtx.signed"
    require ro0.value==1:T
    require ro1.script=="{data.p2} in rtx.signed"
    require ro1.value==1:T
  else require false
}
```

Fig. 5. Pseudocode of a *stateful UTXO-based* bet contract. The spending condition is a switch between three cases, corresponding to the transactions T_{join}, T_{win} and $T_{timeout}$. Note that only the first case requires the script to be preserved, while the others define the scripts of the redeeming transactions as simple a signature verification, terminating the contract and transferring the funds to the players (2:T to the winner for T_{win}, and 1:T each for $T_{timeout}$).

Solidity adheres to the account-based stateful model outlined in Section 2.2.1. Accounts are partitioned into user accounts (a.k.a. Externally Owned Accounts, or EOAs) and contract accounts, and are uniquely identified by an *address*. Contracts, akin to classes in object-oriented languages, have methods to access and update the storage, which consists of the contract balance and variables. Contract variables can record fixed-size data as well as dynamic data structures like arrays and key–value maps. Transactions are signed by a single EOA and trigger contract calls, possibly transferring units of the native cryptocurrency (ETH) from the caller EOA to the contract. The called contract, in turns, can trigger calls to other contracts. Transactions can deploy new contracts; the same contract code can be deployed multiple times, each instance having its own address and storage. Control structures include unbounded loops and recursion, but in practice all computations are bounded by the fee mechanism (see Section 4.8).

Solidity is statically typed, with types of variables and methods specified explicitly by the programmer. It features subtype polymorphism and ad-hoc polymorphism. Some types support type-safe implicit conversions; type-unsafe explicit conversions lead to compile errors. The language has some type-unsafe primitives (e.g., external function calls and inline assembly), which require special care from the programmer [14–16]. Solidity supports multiple inheritance between contracts. Each source file can define multiple contracts and import code from external files. This allows the reuse of code components (libraries, interfaces, and contracts).

Because of its familiar JavaScript-like syntax and its procedural programming style, Solidity is usually considered an easy language to learn. However, it has a few design quirks that, together with the inherent complexity of current DApps, have deep implications on the security of smart contracts. We will discuss some of them later in Section 4.

3.2. Rust/Solana

Rust is a general-purpose programming language, which was adopted as the main smart contract language for Solana. As for Solidity on Ethereum, also Rust must be compiled to bytecode in order to be executed by Solana nodes.

Solana follows a *stateless* account-based model: contracts take the form of procedures, without an associated state. Therefore, any data these procedures interact with is stored within separate accounts, supplied as parameters. Accounts are partitioned into EOAs and contract accounts, but unlike Ethereum, in Solana any EOA is owned by a contract account and can store data associated to that contract account, which instead only stores executable code and it is the only one with write permission. In general, state updates are regulated by the principles of *ownership* and *holdership*: the entity who knows the private key is considered as the holder of the account, while the owner (always a contract account) is the only one that can modify the account data. Special pre-defined contract accounts manage the creation of accounts, the transfer of native currency, and the minting of custom tokens. While this design mandates supplementary checks in the contract to ensure security, it also enables the parallel processing of transactions. To this purpose, transactions specify all the accounts whose data will be read or written throughout their execution: in this way, the runtime environment can detect when two transactions can be executed concurrently: namely, if no transaction reads or writes parts of the state that are written by the other transaction, then the two transactions are parallelizable [17]. Whereas in Ethereum a transaction represents a single contract call, in Solana a transaction can contain several calls, each of which may be related to a distinct contract. These calls are carried out sequentially, and the failure of any one of them results in discarding the changes of the entire transaction. The maximum size of transactions is limited to ∼1KB in order to bound the amount of calls.

Rust is statically typed: notably, its type system can statically detect bugs such as null-pointer dereference, which instead lead to run-time errors in other programming languages like C++. To do that, the type system rigorously tracks data possession throughout the program, enabling it to operate without a garbage collector by detecting memory allocations and deallocations at compile time. The type system ensures that references do not outlive the data they point to causing dangling pointers and that data is not mutated unexpectedly.

Writing contracts directly in Rust poses several challenges to developers, e.g.: (i) contracts must be encapsulated into a single procedure, which must switch to the right part of code depending on the parameters; (ii) the data structures exchanged between the contract and its clients must be manually serialized/deserialized; (iii) contracts must check that the accounts passed as parameters carry the authorizations of the legitimate holders and owners (see Section 4.4). To partially relieve developers from this bureaucracy, the Anchor framework offers higher-level abstractions atop the raw Rust layer [18]. Anchor allows developers to write contracts as sets of methods, and it eliminates the need to manually encode data structures, specifying the contract interface through an Interface Definition Language. Additionally, Anchor automatically performs some of the above-mentioned security checks, based on the types associated with the accounts. A downside is a doubling of deployment fees compared to pure-Rust.

3.3. Aiken/Cardano

Cardano is currently the main smart contract platform following the UTXO model. Cardano extends the UTXO model of Bitcoin in two directions [19]: it follows a stateful storage model, allowing users to include arbitrary data in transaction outputs, and it features a Turing-complete script language, which overcomes the expressiveness limitations of Bitcoin contracts [20]. Cardano consensus nodes execute scripts written in Plutus Core, a low-level untyped lambda-calculus. Although this language is Turing-complete, in practice computations are bounded by the fee mechanism. There are a few high-level languages that compile into Plutus Core, both general-purpose and DSLs. The first high-level contract language for Cardano was Plutus Tx, a general-purpose typed functional language that is a subset of Haskell. This allows Cardano developers to use Haskell to code both the on-chain and off-chain parts of a decentralized application. A main advantage of this approach is the guarantee of consistency between the two parts, e.g. a client will never pass values with the wrong type to the contract. A disadvantage is that, when one is only interested in the on-chain part, using this general framework may be unnecessarily complex. Other languages supported by Cardano are focussed just on the on-chain part: they include Marlowe (a domain-specific language for financial contracts), Opshin, and Aiken.

Aiken, in particular, is a high-level language with a minimal set of features for programming the on-chain part [21], Similarly to Plutus Tx, Aiken is a functional language compiling to Plutus Core. Aiken is used to write the spending conditions of UTXO transactions, akin to the pseudo-code in Fig. 5. This involves checking all the parts of the transaction output that is being spent, and parts of the spending transaction, including its outputs, to ensure that the spending transaction represents a valid update of the contract state. Consequently, programming in Aiken (or any other Cardano languages) requires a paradigm shift w.r.t. the other languages in our selection, which instead support the procedural style. This has repercussions on code readability and security (see Sections 4.2–4.4). Said that, the language is strongly typed, featuring algebraic types and pattern matching, parametric polymorphism, and recursive types. Aiken features recursion, so preserving the (theoretical) Turing-completeness of the underlying Plutus Core language. Aiken also supports anonymous and higher-order functions.

3.4. (Py)TEAL/Algorand

Algorand is a blockchain platform launched in 2019, which over the years has updated its smart contract capabilities several times, passing from a simple model of stateless contracts to Turing-powerful stateful contracts.

Algorand follows the stateful account-based model. Every account (both user and contract) holds a balance of the native cryptocurrency and of custom tokens, as well as data associated to contracts. Unlike Ethereum, where the contract state is entirely stored in a contract account, in Algorand it is distributed across different components: a key–value storage associated to the contract account, a key–value storage associated to user accounts, and further keyed storage segments (called *boxes*), used to overcome the strict size limits of the contract storage (just 8KB shared among a maximum of 64 key–value pairs).

The Algorand nodes execute a custom bytecode, which is the compilation target of higher-level contract languages, using TEAL as an intermediate assembly-like language. The TEAL instruction set is similar to that of a stack-based machine, with only a few abstractions over low-level details. E.g., function invocations are performed via a *call* instruction rather than a plain jump, and a separate call stack is used to store function arguments and return values. Although this requires some stack manipulation to move arguments from the call stack to the operand stack whenever needed, one can easily recover function arguments at constant offsets in the call stack, rather than having them buried deep in the operand stack. TEAL types are limited to byte arrays and unsigned integers. The contract itself is also able, when called, to generate so-called "inner" transactions, which can transfer assets, call other contracts, and more.

To reduce the burden of directly writing TEAL bytecode, a few higher-level languages and frameworks have been proposed, e.g. PyTeal, Beaker, Tealish, TealScript, and PuyaPy. Among them, the most widespread is the pairing PyTeal/Beaker, a library of Python bindings through which one can write Python code that produces TEAL bytecode at run-time. In this way, programmers can use familiar higher-level constructs, like logical/arithmetic expressions, control flow, variables and key–value maps, and functions. Overall, the resulting code is not too dissimilar from the procedural-style code one could obtain e.g. in Solidity. Still, some quirks remain about the handling of storage and of inner transactions (see Section 4.2).

3.5. Move/Aptos

Move is a smart contract language inspired by Rust that has been embedded into multiple blockchain platforms, including Libra/Diem, Starcoin, Aptos and Sui. One of Move's highlights is its static type system based on *linear types*. Linear types enforce the so-called *must-move* semantics, ensuring that tokens (and resources in general) are never replicated or lost. This is a major constraint when writing programs and has a number of implications on the safety properties of the compiled code. Even though linear typing does not prevent a programmer from writing a wrong program in one way or another, it surely helps in crafting correct implementations where illicit replication or deletion of tokens is statically rejected.

Another highlight of Move is allegedly being chain-agnostic. This is not entirely true though: each embedding must deliver a porting of the language tailored to the platform's peculiarities, providing a custom framework and a standard library, as well as applying a few tweaks to the language. In this section we delve into Aptos, a direct successor of Libra/Diem (now dismissed).

The Move/Aptos programming model revolves around a few key principles. Contracts take the form of modules, containing `struct` definitions and functions. Structs are the basis for representing data structures, while functions establish the only interface for module clients to create, access, or modify such data structures. Struct fields can be accessed only from within the module code, granting information hiding and comprehensive control over the operations involving the datatypes therein defined. Once created by a module function, the type system treats structs as first-class resources that cannot be copied or implicitly discarded, only permitting either movement between program storage locations or passing around between function calls. This discipline takes place fully at compile time and is enforced by linear types. Linearity checks can be disabled through *abilities*: tagging a struct with the `copy` ability renders it a *value* open to duplication, while the `drop` ability enables destruction at the end of the scope.

Different Move variants offer distinct persistent storage representations, aligning with the peculiarities of the underlying platform. Aptos defines the *global storage* as a map from account addresses to resources encoded by a struct datatype. The creation of a resource in the global storage is exclusive to the contract signer, performed through a special language primitive. Accessing and modifying resources is less restrictive: anyone can request (`borrow`) a reference to a resource via the account address under which the resource is stored.

Move/Aptos follow a *stateless* account-based model. Global variables are not allowed unless constant, which implies that modules are stateless at the language level. This affects how contracts are implemented: all the relevant data, say the contract *state*, must be stored using user-defined datatypes and eventually retrieved from the global storage.

Each time a contract is run, the address of the invoking user account (the *signer* account) is passed as an argument with a special type `signer` that guarantees that it is non-copiable and it cannot be put into a

Table 1
Features of the contract layer of some of the main smart contract platforms.

Platform	Accounting model	Contract storage	Fees depend on ...	Main contract languages	Programming style
Ethereum	Account-based	Stateful	Tx computation	Solidity	Procedural
Solana	Account-based	Stateless	Num. signers, Data size	Rust	Procedural
Cardano	UTXO	Stateful	Tx size, Tx computation	Plutus, Aiken	Approval
Aptos	Account-based	Stateless	Tx computation, Data size	Move	Procedural
Algorand	Account-based	Stateful	Constant	PyTeal	Procedural
Tezos	Account-based	Stateful	Tx computation, Data size	SmartPy, Ligo	Procedural

user-defined struct datatype or saved on the global storage. This design prevents a contract from performing actions on behalf of other users than the current signer. Such security measures have some drawbacks: for the same reason why only the signer can write a new record on the global storage, any contract involving multiple participants (e.g., auctions, bets, games, *etc*..) must rely on explicit *opt-in*, implying a voluntary choice to engage in a specific activity. This means that each participant has to perform the first write operation; then any participant can access data stored by other accounts through reference borrowing.

3.6. SmartPy/Tezos

The Tezos blockchain features a few high-level contract languages, including Liquidity, Archetype, LIGO, and SmartPy. Among them, the last two seem the most actively supported: here we consider SmartPy, since its Python-like style lends itself to a more direct comparison with Algorand's PyTeal.

Tezos follows the account-based stateful model. Its consensus nodes execute low-level code written in Michelson, a statically-typed, stack-based and Turing-complete bytecode language. SmartPy, as the other Tezos high-level languages, must be compiled into Michelson in order to be executed.

SmartPy exploits meta-programming on top of Python: i.e., SmartPy contracts are just (decorated) Python programs, which are transformed into Michelson code by the SmartPy compiler. Meta-programming allows developers to use the syntax and control structures of SmartPy match Python's, as well as to use Python libraries. The language is fully typed, with type inference performed after a transformation into an intermediate OCaml code (see Section 4.5). When unable to infer a datatype, the SmartPy compiler generates an error and requires an explicit cast. Meta-programming decorators are used to specify the contract interface, i.e. the set of its public functions, the contract storage, and testing scenarios. Datatypes of the contract storage do not correspond to the native Python datatypes, but are defined through the SmartPy library. The deployment of a SmartPy contract specifies the initial contract storage, which is set via the contract constructor. Unlike Ethereuum, this initial storage is statically incorporated in the Michelson code, and the contract cannot use external data (e.g., the caller's address) to initialize its storage. Contract code cannot contain externally defined data, such as externally-defined contracts.

4. Comparative analysis

In this section we perform a comparative analysis of the smart contract languages presented in Section 3. We outline below the key elements of our comparison. A first, high-level view is in Table 1, which classifies languages/platforms according to the architectural aspects discussed in Section 2. A more in-depth comparison is based on our hands-on experience on developing a common benchmark of use cases. We describe our benchmark in Section 4.1, and then in Sections 4.2–4.4 we exploit it to compare the programming styles of contract languages, their verbosity and readability, and the security implications of their design. Then, in Sections 4.5 and 4.6 we discuss the role of the tool chain (compiler and static analyzers) in preventing vulnerabilities and other loopholes. In Section 4.7 we analyze the support for the integration of on-chain and off-chain components. In Section 4.8 we compare the fee models of the blockchain platforms. Finally, in Section 4.9 we reflect on our experience in developing the benchmark, by discussing how the availability of platform functionalities affects the development of smart contracts.

Table 5 summarizes our assessment.

4.1. Smart contracts benchmark

The "Rosetta Smart Contracts" benchmark [1] is a specialization of Rosetta Code to the realm of smart contracts. It showcases the contract languages discussed in Section 3, using them to implement a diversified class of use cases. Two main drivers have influenced our choice of the use cases: first, to provide a representative selection of common DApp use cases, such as those in the Openzeppelin library for Ethereum; secondly, to serve as an adequate test-bed for a comparative analysis of the functionalities supported by the different smart contract languages and platforms.

The benchmark currently includes 21 use cases, whose implementations are distributed across 151 source code files, with a cumulative size of ~900KB and ~18K LoC. Table 2 enumerates the use cases and the functionalities required to implement them. These functionalities represent the basic features that are provided by smart contract languages, possibly exploiting the low-level primigtives made available by the underlying blockchain platforms where the smart contracts are executed. To illustrate, the Bet use case described in Section 4 requires the following functionalities: (i) "native tokens": the contract involves transfers of native cryptocurrency (from the players to the contract for the `join` action, and for the contract to the players for the `win` and `timeout` actions); (ii) "multisig transactions": the `join` action must be simultaneously authorized by both players; (iii) "time constraints": the `timeout` action must be enabled after a given deadline; (iv) "transaction revert": some transactions must be reverted when some conditions are not satisfied (e.g., when the `win` action is not authorized by the oracle). As shown later in Table 4, not all languages/platforms provide native support for all the functionalities listed in Table 2. When that is the case, we resort to workarounds, possibly adapting the specification of the use case (e.g., if multisig transactions are not available, in the Bet contract we can split the `join` action in two actions, one for each player). See Section 4.9 for more details about these workarounds.

4.2. Comparison overview

Roughly, we can partition the smart contract languages presented in Section 3 into two classes, according to the programming style they induce: the *procedural style* and the *approval style*. The former class includes languages where the contract reacts to transactions by updating its state and/or the ledger state (possibly distributed across multiple accounts): Solidity, Rust, Move and SmartPy all belong to this class. The latter includes languages where the contract is expected to approve or discard a single transaction or a group of transactions: Aiken belongs to this class. TEAL/PyTeal follows a hybrid approach, supporting both styles. As we will see in Section 4.4, the programming style is one of the factors that contribute to the security of contracts, and it is strictly related to the level of abstraction provided by the language over the underlying blockchain platform.

Table 2
Functionalities required by the use cases in the benchmark. Entries marked with ⚙ [i] denote functionalities that can be used to implement workarounds in case the functionality marked with ✓[i] is not natively available in the given language/platform.

Use case	Native tokens	Custom tokens	Multisig transactions	Contract updates	Transaction batches	Time constraints	Key–value maps	Dynamic arrays	Bounded loops	Randomness	Transaction revert	Contract-to-contract calls	Delegate contract calls	In-contract deployment	Hash on arbitrary messages	Versig on arbitrary messages	Bitstring operations	Arbitrary-precision arith.	Rational arith.
Bet	✓		✓			✓					✓								
Simple transfer	✓										✓								
Token transfer		✓[1]					⚙[1]				✓	⚙[1]							
HTLC	✓					✓					✓				✓				
Escrow	✓										✓								
Auction	✓					✓	✓				✓								
Crowdfund	✓					✓	✓				✓								
Vault	✓					✓					✓								
Vesting	✓					✓					✓								
Storage								✓											
Simple wallet	✓										✓								
Price bet	✓										✓	✓							
Payment splitter	✓						✓	✓			✓								
Lottery	✓		✓			✓				✓[1]	✓				⚙[1]		⚙[1]		
Const. prod. AMM		✓[1]					⚙[1]				✓	⚙[1]						✓	✓
Upgradeable proxy				✓[1]							✓	⚙[1]	⚙[1]						
Factory							✓	✓			✓			✓					
Decentralized identity											✓				✓	✓			
Editable NFT		✓[1]				⚙[1]					✓	⚙[1]							
Anonymous data								✓	✓		✓								
Atomic transactions					✓[1]			⚙[1]	⚙[1]		⚙[1]				⚙[1]	⚙[1]			

Solidity and **SmartPy** are those that most closely follow the procedural style: contracts have code (a set of procedures) and a state that can be updated in reaction to procedure calls. Despite the strong similarity between these two languages, important differences exist. A notable one lies in the interaction with other contracts. In Solidity, a method `f` can call another contract's method `g` at any point, interrupting the execution of `f` to start that of `g`. In SmartPy, instead, the execution of `g` takes place only *after* the caller `f` has completed. This design choice has repercussions on the programming style and on the security: on the one hand, Solidity's design leads to more natural implementations (e.g., `f` calls an oracle `g` to get some value, and then uses that value in its continuation), but on the other hand it is a cause of attacks (see Section 4.4). Programming the same behavior in SmartPy requires `g` to callback the contract of `f` after it has finished its execution, and store the return value in the caller's storage (which is somehow less natural).

Rust, either raw or using **Anchor**, while still adhering to the procedural style, substantially departs from Solidity and SmartPy. This is only in part explained by the stateless nature of Solana, and by the additional checks on the accounts passed as parameters that this model requires. Programming in raw Rust, as discussed in Section 3.2, requires a careful and often verbose approach, increasing error-proneness due to the extensive use of boilerplate code. Although these issues are partially mitigated by the Anchor framework, Solana contracts are more verbose than those in other account-based platforms (see Section 4.3).

Move, although still based on the stateless account-based model, induces a unique procedural programming style, centered around linear types. Defining data types requires some care: assets cannot be mixed with ordinary data within the same struct, since a different treatment is needed. While integer values can be modified and updated like in common programming languages, assets and resources in general cannot be modified, copied or dropped and must be put into a non-copiable and non-droppable wrapper type in order to be manipulated. According to our experience, making a Move program compile and work properly can require a substantial effort, all the more so when dealing with asset transfers: its strict type system puts the developer on rigid rails; escaping such rails would likely lead to compile-time errors. Similarly to Move, Rust also supports ownership types (so enforcing the same strict discipline over the copiability of datatypes). However the programming model imposed by Solana does not fully exploit the Rust's complex type system, sticking to a more conventional programming practice. In particular, currency and assets are not represented by uncopiable/undroppable datatypes in Rust/Solana, and so their linearity is not statically guaranteed by Rust's type system, but by run-time checks.

Cardano substantially differs from all the other platforms discussed in this paper, being the only representative of the UTXO model. First, the current contract state is recorded in the current unspent transactions that encode the contract. Then, performing a contract action means spending that transaction with a new one that sets the new contract state: therefore, the contract does not compute the new state (as in account-based platforms), but it just verifies that the state in the redeeming transaction is a correct update of the old one. This motivates the paradigm switch from the procedural style to the approval style. **Aiken** brings a purely functional flavor to the table, making code overall robust thanks to strong types and data immutability, albeit verbose and difficult to write for developers trained in procedural programming paradigms. As noted in the pseudo-code of the UTXO Bet contract in Fig. 5, the contract script must check several transaction fields, e.g. the data fields where the contract state is stored. For instance, transferring a token from the contract to some address requires checking that the spending transaction has some outputs with suitable signature verification scripts. This workflow is more complex and verbose than in the account-based model, where an explicit call to some transfer primitive achieves the same goal. Admittedly, Aiken features the typical arsenal of constructs provided by functional languages, including the record update syntax, which somewhat reduces possible errors when updating the state. However, when the contract logic is complex, correct state management turns out to be a cumbersome task and programmers may still introduce errors despite the robust and type-safe design of Aiken. Unlike in account-based models, where interactions between contracts can be rendered directly as contract calls, in the UTXO model contract calls are not meaningful. Indeed, calling a contract would require the caller to perform a sort of "internal" transaction to trigger a computation step of the callee. Although these internal transactions are not featured by Cardano, some forms of composability between contracts are possible, e.g. by multi-input transactions that force dependencies between the scripts of the spent outputs. More sophisticated interactions can be obtained by resorting to layer-2 implementations of asynchronous message-passing [22].

Algorand, being the platform whose contract layer and languages have changed the most during its lifespan, is also the one for which it is most difficult to bring a definitive assessment. Originally, Algorand only supported *smart signatures*, i.e. simple stateless contracts whose primary purpose was that of deciding whether to approve the transactions coming from the smart signature's address [23]. According to our rough taxonomy, smart signatures follow the approval style. After a number of updates, the contract layer was enriched with so-called *applications*, a basic form of stateful smart contracts, but still leveraging smart signatures for handling assets transfers. This contract model was a hybrid between the approval style (needed to write the smart signature part of the contract) and procedural style (needed for the application, which handles the contract state). The introduction of inner transactions and application accounts (see Section 3.4) to the contract layer has made it possible to eliminate the need for smart signatures in stateful contracts, allowing them to construct and submit their own transactions. Effectively, this makes the current programming practice of Algorand adhere to the procedural style.

4.3. Code verbosity and readability

As a rough comparison between contract languages, we measure in Table 3 the LoC of the implementations in our benchmark (restricting to the use cases where all the implementations are available). As expected after Section 2.3, the UTXO-based model, here represented by **Aiken**, leads to more verbose implementations than account-based models. Among the latter, **Anchor** for Rust is definitely the more verbose. This is due in part to the language bureaucracy and in part due to the need to handle data in multiple accounts, which is a consequence of how Solana renders the stateless model. However, statelessness alone does not cause verbosity: e.g., **Move** contracts are more concise than Solana's, which is penalized by the additional account validation checks. The other languages in the account-based model have, on average, similar verbosity: we just note that the slightly higher LoCs of Move are counter-balanced by the increased robustness due to static typing (cf. Section 4.5).

Regarding readability, in the absence of a widely accepted metric we resort to a qualitative evaluation. In general, we have a poor readability when understanding the behavior of a contract requires a low-level knowledge of the structure of blockchain transactions. This is the case e.g. of Aiken and **PyTeal**: in the first case the problem seems inherent to the closeness of Aiken to the UTXO model, while in the second case it seems related to the handling of storage and of inner transactions. PyTeal is also a witness of the fact that a good readability is not always implied by a low verbosity. The readability of Move contracts is strictly related to the understanding of linear types: developers unfamiliar with these concepts will find it quite difficult to make some sense of a Move contract. In Anchor/Solana, poor readability is caused by a combination of factors: unfamiliarity with the Rust ownership model and the distribution of the state across multiple accounts.

4.4. Security implications of language design

The design of a smart contract language and of the underlying contract layer has deep implications on the security of the applications built on them. A paradigmatic example is the famous reentrancy issue of **Ethereum**, which has been the basis of several real-world attacks [24,25]. The issue arises from the combination of a few unfortunate design choices at the EVM level: (i) called methods always have a reference to the caller; (ii) any method can call any other method; (iii) there are no bounds on the depth of nested calls; (iv) the most critical contract field, the ETH balance, is implicitly updated as a side effect of method calls. Putting it all together, it may happen that when a contract calls another contract, the callee might call back its caller in such a way as to modify its state variables, bringing it into an inconsistent state where it performs unwanted actions (e.g., double-sending tokens to the adversary) that would not be possible in consistent states [26]. Reentrancy attacks can be countered by using design patterns ensuring that state updates are applied before potential reentrant calls, or by making contract calls mutually exclusive. However, systematically taking care of every call in a contract (including the pure transfers of currency) is quite demanding and error-prone.

Table 3
Lines of code (LoC), excluding comments and empty lines, of a selection of use cases implementations. For Solana we show LoC of Anchor code, since it is more succinct than pure Rust.

Use case	Solidity Ethereum	Anchor Solana	Aiken Cardano	PyTeal Algorand	Move Aptos	SmartPy Tezos
Bet	39	137	158	110	62	53
Simple transfer	18	91	120	49	30	21
HTLC	25	123	115	60	49	31
Escrow	41	176	120	94	45	28
Auction	51	152	221	129	40	45
Crowdfund	31	182	129	103	49	33
Vault	40	166	171	103	57	38
Vesting	39	149	125	105	48	28
Storage	11	82	75	32	23	18
Simple wallet	47	183	169	87	108	47
Average	*34*	*144*	*140*	*88*	*51*	*34*

Reentrancy attacks are dealt with in various ways by the other platforms considered in this survey. In **Solana**, reentrancy attacks are still possible but limited by the fact that re-entry is possible only as self-recursion. In **Cardano**, reentrancy is ruled out by the absence of contract calls. The same goes with **Aptos**: invoking another contract is not possible unless its module is known at compile-time, and mutual recursive calls between modules are forbidden at compile-time. Combined with the absence of callbacks or delegate calls, this rules out reentrancy by design. **Algorand** is not vulnerable to reentrancy attacks, because, even though contract-to-contract calls are possible, a contract cannot call itself, even indirectly. In **Tezos**, as already mentioned in Section 4.2, reentrancy attacks are mitigated by the fact that the caller function must complete, committing to its state, before performing other calls.

Besides reentrancy, different smart contract languages/platforms suffer from different security concerns. **Solana**, in particular, is prone to weaknesses related to its stateless model, which requires contract callers to provide the account containing the data to be read/written by the contract. Omitting some proper validations on accounts passed as input is a source of attacks: a notable example was the wormhole attack, which caused a loss of more than $320 million [27]. A specific vulnerability of this kind is the *absence of signer verification*. Besides checking that the provided account is valid for a specific operation, the contract must ensure that the transaction is signed by the *holder* of that account. Omitting this check can lead to vulnerabilities. For instance, if the developer omits this check in the `win` method of the `Bet` contract in Fig. 3, then a malicious player could provide the oracle address without the corresponding signature, and set itself as the winner (bypassing the oracle altogether). A related weakness is the *absence of ownership verification*. For example, assume that the ownership check is omitted in the `timeout` method of the stateless `Bet` contract in Fig. 3. Then, an adversary could call `timeout` with a specially-crafted account that allows him to withdraw the whole pot. By confirming that only the contract itself can modify the stored information, the data integrity remains protected. Finally, the ability to invoke malicious or counterfeit contracts inside another contract invocation stems from the user's capability to supply any contract account, prompting the need for measures to verify the authenticity of the invoked contracts.

Aiken follows the approval style, in that the contract must check the transaction fields to decide whether to approve an incoming transaction or not. Forgetting even a single check may give rise to security vulnerabilities, possibly allowing an adversary to set a data field of the new state to an arbitrary value. The same concerns apply to **PyTeal**, when used to write (approval-style) smart signatures.

Most of **Algorand** weaknesses revolve around its peculiar treatment of memory. In order to disincentivise the abuse of on-chain storage, every account must maintain a minimum balance that varies depending on how much memory it is using in the blockchain (which, in turn, depends on the number of distinct assets owned, contract data stored, *etc*..). Managing this balance constraint is tricky: developers must make sure that accounts the contract interacts with (and the contract account itself) always satisfy the minimum balance. This can create problems as transactions may unexpectedly fail, as they may lead the contract (or another account) to hold a balance lower than the allowed minimum. In particular, when emptying a contract account, it is essential to distinguish the case in which assets are sent from the case in which the contract account is closed.

Further security implications of the fee mechanism design are discussed later in Section 4.8.

4.5. Compile-time checks

With the exception of PyTeal/Algorand, all the languages considered in this paper feature strong typing. **Solidity** supports subtype polymorphism, allowing programmers to implement contracts by inheriting other contracts in an object-oriented fashion. It also features static visibility modifiers for functions and state variables, and dynamic (programmable) modifiers to restrict access to functions depending on run-time parameters. Extra static checks performed by the compiler detect potential overflows/underflows and division by zero, stack size limit vulnerabilities, and unwanted variable shadowing caused by inheritance. **Rust** supports object-orientation, subtyping and parametric polymorphism. Its compiler also tracks references and data ownership, ensuring memory safety and preventing data races. While Rust is a safe language, Solana does not provide an interface of the same quality, imposing several weakly typed programming patterns for writing contracts. This renders the powerful checks performed by the Rust compiler irrelevant to some extent. **Move**'s resource-oriented programming model is inspired by Rust: ownership of data is explicitly defined and enforced by the type system, and a borrow checker similar to Rust's prevents multiple mutable references to the same resource. Linear types further add to the number of static checks by preventing code from replicating or losing currency and assets in general, ultimately mitigating double spending through typing. Such features are similar to Rust's in principle, though in Move they are more integrated with the language syntax and straightforward for the programmer. That is actually due to the fact that Move is a special-purpose language specifically tailored for asset management in smart contract programming. **Aiken** too is a special-purpose language that stands out of the pack, as it delivers a purely functional style, with static typing and type inference. Although this is fundamental to the safety of the validator script, static typing alone is not sufficient to rule out logic errors, as discussed in Section 4.4.

SmartPy and **PyTeal**, although both based on Python, are substantially different when it comes to static checks. PyTeal contracts are just Python programs that produce TEAL bytecode when executed. Instead, SmartPy contracts are compiled into Michelson, a typed bytecode language. The static typing and type inference supported by SmartPy are preserved by the compilation through type reconstruction. Furthermore, SmartPy contracts can carry type annotations, accessible as structured values through an API. These are actually runtime entities for Python but are converted into type annotations in Michelson at translation time. Such a hybrid approach improves the safety of SmartPy while retaining the simplicity of the Python syntax. At the time of writing, Algorand lacks a compelling high-level language with static typing. Programming in TEAL is equivalent to coding in assembly, thus with little to no static checks on the code. Although PyTeal features a rudimentary type system, type errors are still possible when encoding or decoding stored data, possibly leading to unpredictable errors and mishandling of the required datatypes.

Overall, with the notable exception of Move linear types, which can prevent double-spending, the type systems of the other languages can mostly prevent bad coding practices rather than some forms of vulnerability. As noted in Section 4.4, language design, when specifically tailored to rule out certain kinds of attacks in the first place, is more effective than most common forms of typing.

4.6. Contract analysis and verification

While compile-time checks are useful to rule out vulnerabilities due to common programming errors, they cannot guarantee that a contract respects some ideal behavior in the presence of adversaries. Several tools have been developed to detect potential vulnerabilities in contracts. This is especially true for **Ethereum**, where dozens of bug detection tools with varying detection capabilities exist [28–30]. In **Solana**, current security tools include VRust [18] and FuzzDelSol [27]. Both tools can detect Solana-specific vulnerabilities, like e.g. the absence of signer checks and owner checks discussed in Section 4.4, using different techniques (inter-procedural data flow analysis for VRust, coverage-guided fuzzing for FuzzDelSol). In **Algorand**, current tooling includes Panda [31], which is based on symbolic execution of TEAL code, and Tealer, which searches suspicious patterns in the control-flow graph extracted from the TEAL code.

More sophisticated tools enable the verification of contract implementations against an ideal, abstract description of their behavior. For **Solidity**, this kind of analysis is partially supported by the assertion checker incorporated in the compiler, and by a few external analysis tools [32,33]. However, due to the intricacies of the Solidity/EVM semantics, these tools have several limitations in their precision and expressiveness of target properties [32]. **Move** features a property specification language that can be used by programmers to annotate function invariants. Such invariants are verified at compile time through by the Move Prover, which is bundled with the Aptos toolchain. A bytecode verifier validates compiled contracts at deploy-time, preventing maliciously crafted code from being uploaded to the blockchain. Notably, the bytecode verifier enforces the same type-safety properties (including linearity) that are enforced by the Move compiler over the original source code. The work [34] applies the Move Prover to the formal verification of relevant functional requirements of modules of the Aptos Framework.

Verification tools for **Tezos** include MiChoCoq and ConCert, which verify the functional correctness of contracts against a specification based on pre- and post-conditions in the Coq proof assistant [35,36]. Other static analyzers exist, based on refinement types [37] and on abstract interpretation [38,39].

4.7. On-chain/off-chain interactions

Off-chain systems are essential to extend blockchain features (e.g. layer 2 and blockchain interoperability protocols) and provide users with Web3 services and decentralized applications. Blockchain features (both at the consensus and at the contract layers), contract languages, and off-chain libraries all impact the development of off-chain systems. In particular, the interaction between off-chain and on-chain systems depends on how data flows to/from contracts. In account-based platforms, data can be fed to contracts through method invocations. Some systems (Rust/Solana and Algorand) only allow base types as parameters in calls to contract entry-point functions, whereas Solidity, Anchor/Solana and Beaker/Algorand also support structured data. Outputting data from contracts is usually done through return values of

method calls. In the platforms where return values are not supported, contract outputs can be either written in other accounts (Solana) or embedded in transaction data (e.g., Algorand, Tezos, Cardano).

Depending on where contract outputs are written, off-chain components use different techniques to retrieve them. In Move and SmartPy, the off-chain component can directly call methods because the blockchain preserves the interface and types. In the other systems, the off-chain components must first code the contract public interface before calling a method. The other output retrieval technique is based on listening to events (or logs) emitted by the contract. This is fully supported in Solidity, Move, and SmartPy. Contracts in PyTeal do not emit events but, as mentioned, rely mainly on the log to output results. The other languages considered in this survey do not support events emissions. In Anchor and Aiken, low-level transaction logs can be exploited instead.

Programming off-chain systems is facilitated by official or third-party supported SDKs and libraries (namely Web3). Solidity has stable support in a wide range of programming languages (including those for embedded devices). There are different versions of web3-like libraries and SDKs available for all of the other platforms examined. In particular, JavaScript libraries exist for Rust, Aiken (in two independent projects: Lucid and Mesh), Move, and SmartPy. Python libraries provide support for Aiken, PyTeal, Move and SmartPy (Taquito). Rust libraries are available for Rust and Algorand.

4.8. Fees

The fee model established by the contract layer has non-negligible repercussions on the programming of smart contracts: developers must have a good understanding of the fee model in order to avoid paying more fees than strictly needed or incurring in potentially insecure programming patterns.

In **Ethereum**, fees depend on the sequence of EVM instructions needed for executing a transaction, and are paid by its sender. Each EVM instruction has a cost, specified in terms of *gas units*. The fee is the total amount of gas units consumed to execute the transaction times the price for gas unit. The number of gas units per transaction is bounded: transactions exceeding such limit pay the fee, but have no other effects on the blockchain state. So, although contracts can have unbounded loops and recursion, in practice all computations are bounded. The gas limit also bounds the contracts size, making it necessary to downsize the contract code or distribute its logic across multiple contracts. The gas mechanism is a notorious source of attacks. At the network level, DoS attacks [40] exploit the discrepancy between the gas units associated to EVM instructions and the actual computational resources needed for their execution [41]. Dealing with these attacks caused several revisions of the gas costs (e.g., EIP150, EIP1559, EIP2929), possibly breaking existing contracts that depend on gas costs. Fees can also be the basis for attacks to contracts. E.g., a contract with a method that iterates over a dynamic data structure, such as a key–value map, can be attacked by making the structure grow until the iteration exceeds the maximum gas limit: in this way, the contract gets stuck, and its funds frozen. By combining the fee mechanism with transaction-ordering dependence, attacks based on the unpredictability of fees are possible: e.g., an adversary might front-run a transaction to change the contract state so to cause the transaction to be reverted or pay more fees than expected. The gas mechanism adopted by **Tezos** is conceptually similar to that of Ethereum, and therefore suffers from similar issues.

These attacks are not possible in the platforms where fees are predictable, as in Solana, Cardano and Algorand. In **Solana**, transaction fees are determined solely by the number of required signatures for a transaction, rather than the amount of resources used. Besides transaction fees, Solana imposes fees on the data stored in accounts, to incentivize users not to waste on-chain space. If an account has not enough balance to cover the rent, it faces removal. Accounts can be exempted from paying fees by holding a balance that is at least equivalent to two years' worth of rent. Taking rent fees into account influences contract development in Solana. E.g., in the stateless `Bet` contract (Fig. 3), upon the completion of a final action the developer should close the storage account `s` and return the remaining value, used for rent, back to the initializer. This requires an explicit coding of additional operations into the contract.

In **Cardano**, transaction fees depend on the transaction size and on the number of CPU steps and memory needed for its execution. All these data are predictable before sending the transaction, since Cardano is not subject to transaction-ordering dependencies, being based on the UTXO model.

In **Algorand**, although contracts are executed after compilation to low-level code as in Ethereum and Tezos, transaction fees are calculated differently. Namely, while in Ethereum and Tezos the fees depend on the sequence of executed low-level instructions, in Algorand they are determined only by the transaction size, with a lower bound set by the platform.

The fee model in **Aptos** incorporates elements from the models proposed by EVM, Algorand, and Solana, featuring a base minimum fee along with computation costs (referred to as I/O costs) and "storage rent" fees. Aptos transactions require a two-component fee structure that includes execution I/O costs and storage fees. The computation costs are measured in gas units, with the price fluctuating based on the network's load. The storage component is priced at a fixed rate in the platform's principal cryptocurrency. The storage fee can be refunded when the allocated space is deleted (as in Solana).

4.9. Native vs. programmable functionalities

Developing our smart contracts benchmark was instrumental in understanding how different patterns are rendered in different languages/platforms. We detail in our repository [1] the workarounds we adopted to implement the use cases in the various languages, and summarize below our main findings. For a quick reference, Table 4 depicts a comprehensive recap of the functionalities discussed below, plus a number of additional minor ones, for each language/platform explored in this paper.

Custom tokens All blockchain platforms come with a principal cryptocurrency (e.g., ETH for Ethereum), which is minted under the control of the consensus protocol and is exchangeable among users via direct transfers or programmatically via smart contracts. Many real-world contracts use tokens to represent custom assets (in our benchmark, the Token transfer use case). Unlike the principal currency, the minting of these custom tokens is not regulated by the consensus protocol, but rather by a user-defined policy. Solana, Cardano, Algorand, Tezos and Aptos support tokens natively, and allow contracts to define their transfers similarly to the native cryptocurrency. In Move, which supports parametric polymorphism, custom assets are implemented via the generic type `Coin`, whose type parameter `CoinType` specifies the fungible asset type. Programmers can ensure that only assets of the same type are exchanged: this is achieved through the combined action of static typing and a dynamic lookup mechanism of resources driven by types. In Ethereum, instead, tokens are not supported natively, and so they must be programmed as contracts, by implementing standard interfaces (e.g. ERC20/ERC721 for fungible/non-fungible assets). This comes at a cost for developers, as it is their duty to prevent asset duplication, unintended losses and other mishandling. Furthermore, malicious token implementations could be an attacks vector to smart contracts [42].

Multisig transactions Another discriminating feature is given by *multisig* transactions, i.e. transactions that can carry the signature of multiple users. They are required e.g. in the Bet use case, where two players must simultaneously deposit 1 token to join the game. The Cardano and Solana implementations fully respect the specification, since the underlying platforms support multisig transactions. Platforms such as Tezos and Ethereum do not provide native support for this feature,

Table 4
Functionalities supported by smart contract languages/platforms. The first group refers to the functionalities described in Section 4.9. Checkmarks ✓denote functionalities that are available natively in the language or via the blockchain APIs. The symbol ⚙ denotes functionalities that can be implemented in a smart contract with some practical workaround (e.g., ERC20/ERC721 interfaces for custom tokens in Ethereum). An empty cell indicates that the functionality is not supported by the platform. Further workarounds are implemented in our benchmark [1].

Functionalities	Solidity Ethereum	Rust/Anchor Solana	Aiken Cardano	TEAL/PyTeal Algorand	Move Aptos	SmartPy Tezos
Native tokens	✓	✓	✓	✓	✓	✓
Custom tokens	⚙	✓	✓	✓	✓	✓
Multisig transactions	⚙	✓	✓	✓	✓	⚙
Contract updates	⚙	✓		✓		⚙
Transaction batches	⚙	⚙	✓	✓	⚙	⚙
Time constraints	✓	✓	✓	✓	✓	✓
Key–value maps & Dynamic arrays	✓	✓	✓	✓	✓	✓
Bounded loops	✓	✓	✓	⚙	✓	✓
Randomness	⚙	⚙	⚙	✓	✓	⚙
Transaction revert	✓	✓	✓	✓	✓	✓
Contract-to-contract calls	✓	✓		✓		✓
In-contract deployment	✓			✓	✓	✓
Delegate contract calls	✓				✓	
Hash on arbitrary messages	✓	✓	✓	✓	⚙	✓
Versig on arbitrary messages	✓		✓	✓	✓	✓
Bitstring operations	✓	✓		✓	✓	✓
Arbitrary-precision arithmetic		✓	✓			
Rational arithmetic	⚙	⚙	✓	⚙	⚙	✓

but a workaround exists, as illustrated in the implementation of the Bet contract, where the effect of multisig transactions is captured by a specific pattern that splits the join action into two steps. An alternative workaround is to implement a *multisig contract*, which performs some given actions only if authorized by at least a given number of users. In Tezos, multisig contracts can be crafted by exploiting lambdas and the ability to verify signatures on arbitrary messages (furthermore, they are natively supported by the official client). Algorand supports multisig through multi-signature accounts, that is special sender addresses that have to be created by off-chain code and require a set of signatures to be authorized to perform the transaction. Aptos offers a similar mechanism based on off-chain code.

Contract updates Among the platforms considered in this paper, only Solana and Algorand allow to update contracts once deployed. In the other platforms, this feature can be simulated through an UpgradeableProxy contract, which intermediates the interactions between callers and a callee, allowing the owner to update the callee address (and so, the contract that processes function calls). The Solidity implementation exploits delegate calls to ensure that the caller and callee interact as there were no proxy intermediation. In Solana, although contract updates are supported (at the cost of transferring account ownership to the new contract to remedy mutating restrictions), implementing the proxy does not seem possible. Cardano does not support contract calls, therefore the proxy cannot be implemented. Still, contract updates are possible by making the validating script accept any transaction signed by the owner, allowing them to effectively replace the old script with the new one. Aptos does not support contract calls or contract updates either. SmartPy allows for contracts updates through lambda functions. Namely, the contract is represented as a mapping, whose values are lambdas. A method call is then translated into calling the corresponding lambda, while updating the contract is performed by updating the mapping.

Transaction batches In some use cases (e.g., circular trades and group payments) it is useful to batch transactions to ensure that either all or none of the transactions in a batch are executed. Among the platforms considered here, transaction batching is supported natively only by Algorand (both client-side and contract-side) and by Solana (only client-side). In the absence of native support, a similar effect can be obtained by deploying a contract with a function that performs a *specific* sequence of function calls. The atomic transactions use case in our benchmark generalizes this by using a single contract that can process *arbitrary* transaction batches. Our Solidity implementation exploits delegate calls to ensure transparency of the caller contract. In Cardano, batching is not rendered in the strict sense of the term, but it is implicitly implemented by the UTXO mechanism. For instance, if we want two payments, say $1:T$ from A to B and $1:T'$ from B to C, to happen atomically, we can obtain the same effect by a *single* transaction with two inputs (one redeeming $1:T$ with A's signature and the other redeeming $1:T'$ with B's) and two outputs, controlling $1:T$ and $1:T'$ with B's and C's signatures, respectively. Although Aptos does not support transaction batches natively, programmers can pack multiple actions in a single Move script, i.e. a code block that can invoke functions defined in contract modules and is executed atomically (similarly to Ethereum's workaround).

Time constraints Most platforms allow contracts to set time constraints by making the current block number or transaction timestamp readable by the contract. This is a common feature in real-world scenarios: in our benchmark, it occurs e.g. in the Bet, Auction, Crowdfund, HTLC, Vault and Vesting use cases. In Cardano, contracts cannot access the global blockchain state (including the block number), but time constraints can be implemented leveraging the validity interval field of transactions. Knowing only a time interval rather than the exact time might introduce some approximations w.r.t. the ideal behavior. E.g., in the Aiken version of the Vesting contract, the beneficiary can receive slightly less than the amount prescribed by the vesting function at the exact time the transaction is processed. This discrepancy arises since the amount is determined as a function of the (lower bound of) the validity interval.

Key–value maps, dynamic arrays, and bounded loops All the languages considered in this paper support dynamic data structures such as key–value maps and arrays. In the stateful account-based models, they are stored in the contract account. In Solana, instead, values are stored in accounts, whose addresses serve as keys for key–value maps. For this specific purpose, Solana uses special addresses that are deterministically generated but lack corresponding private keys and are tailored to be under the exclusive control of a designated smart contract. In Aiken/Cardano, dealing with dynamic data structures raises some

efficiency concerns, since updating the contract state requires sending a transaction carrying the *whole* new state. In contracts whose state involves arrays or key–value maps that can grow significantly during execution, these transactions may become larger and larger. This has two drawbacks, in that the transaction fees increase with the transaction size, and in that there is a hard limit (16KB) on this size. These issues could be potentially mitigated by using cryptographic techniques (e.g., Merkle trees) to minimize the amount of data stored on-chain. Currently, this has to be implemented manually by the programmer, as Aiken does not feature constructs to automatize the management of large data structures exploiting these cryptographic primitives. In Algorand, dynamic data structures can be implemented by using *boxes*, i.e. pieces of memory that can be allocated at any time during the lifetime of the contract, at the cost of an increased minimum balance for the contract account. These boxes are, however, fairly expensive. When a use case requires a key–value map indexed on account addresses, the use of the local storage of these accounts is preferred to that of boxes. Consider, for instance, a contract that receives deposits from users, and that needs to record the amounts transferred by each user (like e.g. in the crowdfund use case in our benchmark). In Algorand this can be achieved by distributing the map across the local storage of each account depositing tokens to the contract. As a single call can only read the content of a limited number of boxes (8 per call), it is not possible to iterate over structures that span a large number of boxes. This means that iterating over arrays is still feasible provided that the array is encoded in a single box; instead, iterating over dynamic data structures such as key–value maps is quite problematic. Furthermore, working with multiple maps is tricky: since box storage maintains a single key–value store, making it appear as multiple maps requires the developer to manually handle the partitions. This issue, together with the varying minimum balance on the insertion of new key–value pairs, makes the use of key–value maps in Algorand quite burdensome.

Randomness Some use cases require randomly-generated values (e.g., in lotteries and other games to choose a winner or to draw a card). Randomness has also proven effective in mitigating the threats posed by criminal smart contracts [43]. Although some centralized randomness beacons are available, their use is not considered secure, since dishonest providers can bias their outputs [44,45]. A secure alternative is given by *commit-reveal-punish* schemes, which construct random values by combining values independently provided by users. To ensure that no one can observe the others' values to craft their own (which would easily lead to attacks), these schemes force users to commit the hashes of the chosen values before revealing them and use collaterals to rule out dishonest users who do not reveal (see e.g. the HTLC use case). A drawback of these schemes is that they become quite complex when many users are involved. A viable alternative is given by Verifiable Random Functions [46], a cryptographic primitive that allows users to generate publicly verifiable random values. Among the platforms considered here, only Algorand offers this feature natively, by combining a randomness seed beacon and a special opcode to verify the correct generation of values. We note that the analysis and formal verification of randomized smart contracts is currently an under-explored research field, with limited tool support [47].

5. Related work

In recent years, there has been a surge in advancements within decentralized, permissionless blockchain technologies, with a particular emphasis on smart contracts. However, despite this remarkable progress, there remains a substantial gap in our understanding of the fundamental principles and programming paradigms that underpin smart contracts. While numerous studies have examined specific applications and challenges associated with smart contracts, there has been a glaring absence in the exploration of their programming principles and languages.

Recent literature reviews, such as [48,49], aimed to provide systematic overviews of technical challenges in smart contract development. Sharma et al. [48] delved into aspects such as consensus algorithms, permission policies, Turing completeness, and data models. They identified crucial challenges such as readability, code correctness, execution efficiency, privacy concerns, and gas exceptions. Similarly, Zheng et al. [49] performed a comparative analysis of platforms and applications, examining essential aspects such as creation, readability, execution efficiency, transaction ordering, deployment, and privacy-preserving mechanisms. They also categorized applications and outlined common use cases across diverse domains. In contrast to [48,49], which primarily rely on comparing results from existing literature on smart contract languages, our comparisons is based on our practical experience developing a benchmark of use cases, where we contrast different platforms/languages by implementing a range of smart contracts in each language. This benchmarking methodology enables us to perform a comprehensive comparative analysis, offering insights into programming styles, readability and usability, compile-time checks, on-chain/off-chain interactions, as well as security considerations across different platforms/languages.

The impact of smart contracts on industry has spurred a wealth of research, see, e.g., [50,51]. Varela-Vaca et al.' work [50] categorized smart contract languages from both academic and industrial perspectives, with an emphasis on improving developer experiences for creating more human-readable smart contracts. Similarly, Dhaiouir et al.' literature review [51] compared distributed platforms, aiming to assist businesses in selecting suitable platforms for blockchain-based applications, thus facilitating informed decision-making. Vacca et al. [52] reviewed methods, techniques, and tools for improving the design, construction, testing, maintenance, and overall quality of smart contracts and DApps. Similarly, Zou et al. [53] performed an empirical study on developers' challenges and practices in smart contract development, with a focus on Ethereum. They collected valuable insights into the current state of the art through interviews and surveys with industry practitioners.

A few works address smart contract languages for UTXO blockchains, mainly focussing on Bitcoin and Cardano. Outside academic research, Bitcoin is quite unattractive as a layer-1 smart contract platform, because of the expressiveness limitations of its script language, its low throughput and high transaction fees. Still, a small subset of the use cases in our benchmark can be implemented also on Bitcoin [20], either using Bitcoin script of higher-level languages such as BitML [54]. In the Cardano literature, the work [55] draws an interesting comparison between the account-based and the UTXO model based on the implementation of a token use case in Solidity and in Plutus. In this paper we extend the comparison in [55], by implementing a large set of use cases in six smart contract languages. The relation between transaction redeem scripts and structured contracts in Cardano is explored in recent research [56].

Several surveys address the challenges related to security vulnerabilities. Hu et al. [57] categorized schemes and tools aimed at improving secure smart contract development. Additionally, they addressed challenges like privacy breaches, execution inefficiencies, and contract complexity by categorizing extensions and alternative systems for contract execution. Rouhani et al. [58] conducted an extensive review focusing on smart contract platforms and domain-specific programming languages, focussing on security vulnerabilities and performance optimization. Their study explored methods and tools for mitigating vulnerabilities. Hewa et al. [59] undertook a comprehensive survey on smart contracts, emphasizing aspects like security, privacy, gas cost, and concurrency. In particular, they explored the integration of smart contracts with emerging technologies such as artificial intelligence and game theory.

The works [60,61], which compare smart contract languages in terms of usability and security, are the most closely aligned with ours. In [60], Voloder et al. conducted a comparative analysis of

Table 5
Strengths and weaknesses of smart contract languages.

Language	Strengths	Weaknesses
Solidity Ethereum	Familiar procedural programming style Extensive documentation Rich ecosystem of analysis tools third-party libraries	EVM design induces vulnerabilities (e.g. reentrancy) Low-level interferences with the semantics (e.g., fees) Transaction-ordering dependencies
Rust/Anchor Solana	Powerful strongly typed language Good parallelization/transaction throughput Rich ecosystem and community	Rust ownership system not really used Verbose programming model Need to serialize/deserialize data manually
Aiken Cardano	Strongly-typed functional paradigm No transaction-ordering dependencies (UTXO) Arbitrary-precision arithmetic	Requires reasoning about the structure of transactions No contract calls Transactions must specify the *whole* contract state
TEAL/PyTeal Algorand	Rich set of functionalities (transaction batches, Verifiable Random Functions, contract updates) Predictable transaction fees	Cumbersome handling of memory (local, global, boxes) Cumbersome constraint on minimum accounts balance Weaker typing guarantees w.r.t. other languages
Move Aptos	Linear types prevent errors (e.g. double spending) Static prover enforces semantic properties Extensive stdlib and framework	Requires good understanding of linear types Lack of community and third-party libraries
SmartPy Tezos	Strong typing and type inference on top of Python Queued method call to avoid reentrancy attacks Allows Python meta-code	A strongly typed Python is a little awkward Lack of dedicated third-party libraries and tools

five platforms focusing on developers' perspectives. Their comparison examines critical features such as documentation availability, ease of installation, automated testing capabilities, implementation efforts, and the required level of expertise for specific use cases and chains. Parizi et al. [61] analyzed the usability and security aspects of three smart contract languages: Solidity, the Pact language for Kadena (which is Turing-incomplete), and Liquidity for Tezos (a typed functional language). The paper offers a comparative analysis of these languages, demonstrating sample contract implementations and evaluating them in terms of usability and security. In contrast to [61], we opt to exclude Pact and Liquidity from our selection of smart contract language. This choice is based on our paper's emphasis on Turing-complete languages (which is also the reason why we neglect contract languages on Bitcoin), as well as the recognition that Liquidity is no longer actively maintained.

Differing from prior research, this paper offers a unique perspective by providing a detailed technical comparison of smart contract languages from the standpoint of programming language theory. We delve into programming styles, language constructs, and typing considerations, complemented by a qualitative assessment derived from hands-on experience in crafting a standardized benchmark for smart contracts. Marking a pioneering effort, this work provides the first extensive hands-on evaluation, facilitating both comparison between smart contract languages and analysis of development and execution costs.

6. Conclusions

We have compared the smart contract languages of some of the most widespread blockchains. The comparison, which was performed along different axes, is based both on the literature and on hands-on knowledge derived from the construction of a common benchmark of smart contracts. Table 5 summarizes the main findings of our comparison: we conclude by discussing the lessons learned in our work.

Lesson learned #1: language abstractions Our analysis highlights the need for high-level abstractions over the low-level details of the underlying blockchain. Clean abstractions are crucial to simplifying reasoning about the correctness and security of contracts. Not all languages considered in this paper have such clean abstractions. For instance, the lack of good abstractions for tokens and contract-to-contract interactions is one of the main causes of vulnerabilities in Solidity/Ethereum contracts (see Section 4.4). The lack of good abstractions over the transactions level in Aiken/Cardano induces a burdensome programming style for contracts in these languages, with potentially harmful consequences on their security (see Sections 4.2 and 4.4). Furthermore, the interference between the low-level fee mechanisms and the contract semantics is not always hidden from programmers, who must have a good understanding of these mechanisms to avoid writing inefficient or vulnerable contracts (see Section 4.8).

Lesson learned #2: typing assets Assets deserve special treatment at the type level in order to prevent programmers from making financial mishaps when manipulating crypto-assets. This can be enforced to varying degrees. The loosest form is to represent assets by means of a custom datatype (distinct from the plain integer type), which prevents programmers from performing unwanted arithmetic operations on assets. By limiting the number of possible operations for the asset datatype, and providing only a minimal set of primitives for transferring assets, account-based platforms can reduce error-proneness when handling valuable tokens. Disciplining assets and transactions in UTXO platforms is not as straightforward, though. In Aiken, for instance, asset transfers are implemented as record field updates where arithmetic operations are required to manipulate amounts. A special asset datatype with its own set of functions would make things harder and verbose for the programmer. The strictest form of control among the languages reviewed in this paper is Move's linear types, which push the envelope by forbidding duplication and loss of assets at compile-time (see Sections 4.2 and 4.5). Although such a strict type discipline is hard to digest for a casual programmer, from our experience it does not come without its own merits. Move contracts seem less susceptible to asset-related issues (e.g. double spending and financial loss) compared to other platforms, underlining that smart contract languages ought to dare more than general-purpose languages when it comes to the discipline imposed on types, especially on the type representing assets.

Lesson learned #3: native vs. programmable functionalities The smart contract languages considered are characterized by different sets of native functionalities, as displayed in Table 4. The absence of some functionality could be detrimental to the implementation of certain use cases, making it either impossible, or possible only through complex workarounds and adaptations of the requirements. We have directly experienced the lack of native functionalities in our benchmark, where some implementations required such adaptations and workarounds. Although in principle all the languages considered in this paper are Turing-powerful (up-to computation bounds due e.g. to transaction

fees), some workarounds could be extremely impractical due to the high costs of on-chain computation and storage, besides the computation bounds. For instance, implementing arbitrary-precision arithmetic via Church encodings would make little sense. Improper workarounds could affect security and decentralization. This is the case, e.g., of generating randomness via block timestamps or external oracles. In general, the availability of specific native functionalities could be an important factor in the decision-making process to choose a blockchain platform, among others [62].

Lesson learned #4: procedural vs. approval style As we have seen in Section 4.2, smart contract languages can be partitioned into two classes based on the programming style they support: the *procedural style*, where contracts react to transactions by updating their state and triggering effects (e.g., token transfers), and the *approval style*, where transactions already contain their effect, and the contract reacts by deciding whether to approve a transaction or not, depending on its state and by the environment. In Sections 4.3 and 4.4 we have seen that the programming style has deep implications on the readability of contracts and on their security: roughly, the approval style is less readable and more error-prone, since the programmer must ensure that the new state is a correct update of the old one, which might involve multiple checks on the transactions fields. Based on the implementation of our benchmark, we argue that the procedural style is overall the most practical, even though in some of its incarnations we note that the produced code is burdened with boilerplate code (e.g., in Rust/Solana), or with type-based manipulations of resources that may look unfamiliar to average programmers (e.g., in Move/Aptos). An open question is whether it is possible to reconcile the procedural style with the UTXO-based model, so to program smart contracts *à la* Solidity while preserving the key strengths of UTXO blockchains like Cardano (e.g., the absence of transaction-ordering dependencies and the parallelizability of transactions).

CRediT authorship contribution statement

Massimo Bartoletti: Writing – review & editing, Supervision, Software, Methodology, Conceptualization. **Lorenzo Benetollo:** Writing – review & editing, Software, Methodology. **Michele Bugliesi:** Writing – review & editing, Methodology, Conceptualization. **Silvia Crafa:** Writing – review & editing, Methodology, Conceptualization. **Giacomo Dal Sasso:** Software. **Roberto Pettinau:** Software. **Andrea Pinna:** Writing – review & editing, Software, Methodology. **Mattia Piras:** Software. **Sabina Rossi:** Writing – review & editing, Methodology, Conceptualization. **Stefano Salis:** Writing – review & editing, Software. **Alvise Spanò:** Writing – review & editing, Software, Methodology, Conceptualization. **Viacheslav Tkachenko:** Writing – review & editing, Software. **Roberto Tonelli:** Writing – review & editing. **Roberto Zunino:** Writing – review & editing, Methodology, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Massimo Bartoletti reports financial support was provided by European Union. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Work partially supported by the Project PRIN 2020 “Nirvana - Noninterference and Reversibility Analysis in Private Blockchains” and by projects PRIN 2022 DeLiCE (F53D23009130001) and SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU. The authors declare that they had no investment or advisory relationships with any of the blockchain companies/foundations cited in this research.

Data availability

No data was used for the research described in the article.

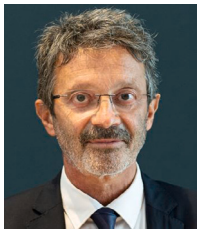
References

[1] Rosetta Smart Contracts, 2024, https://github.com/blockchain-unica/rosetta-smart-contracts.
[2] V. Capocasale, D. Gotta, G. Perboli, Comparative analysis of permissioned blockchain frameworks for industrial applications, Blockchain: Res. Appl. 4 (1) (2023) 100113, http://dx.doi.org/10.1016/j.bcra.2022.100113.
[3] T. Neudecker, H. Hartenstein, Network layer aspects of permissionless blockchains, IEEE Commun. Surv. Tutor. 21 (1) (2019) 838–857, http://dx.doi.org/10.1109/COMST.2018.2852480.
[4] M. Kelkar, S. Deb, S. Kannan, Order-fair consensus in the permissionless setting, in: ACM on ASIA Public-Key Cryptography Workshop, ACM, 2022, pp. 3–14, http://dx.doi.org/10.1145/3494105.3526239.
[5] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, A. Juels, Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability, in: IEEE Symp. on Security and Privacy, 2020, pp. 910–927, http://dx.doi.org/10.1109/SP40000.2020.00040.
[6] K. Qin, L. Zhou, A. Gervais, Quantifying blockchain extractable value: How dark is the forest? in: IEEE Symp. on Security and Privacy, IEEE, 2022, pp. 198–214, http://dx.doi.org/10.1109/SP46214.2022.9833734.
[7] L. Heimbach, R. Wattenhofer, SoK: Preventing transaction reordering manipulations in decentralized finance, in: ACM Conference on Advances in Financial Technologies, (AFT), 2022, pp. 47–60, http://dx.doi.org/10.1145/3558535.3559784.
[8] IOHK, How to write a scalable Plutus app, 2022, https://plutus-apps.readthedocs.io/en/stable/plutus/howtos/writing-a-scalable-app.html.
[9] Sundae Labs Team, Concurrency, state & cardano, 2021, https://sundae.fi/posts/concurrency-state-cardano.
[10] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on Ethereum smart contracts (SoK), in: Principles of Security and Trust (POST), in: LNCS, 10204, Springer, 2017, pp. 164–186, http://dx.doi.org/10.1007/978-3-662-54455-6_8.
[11] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, W.J. Knottenbelt, SoK: Communication across distributed ledgers, in: Financial Cryptography and Data Security, in: LNCS, 12675, Springer, 2021, pp. 3–36, http://dx.doi.org/10.1007/978-3-662-64331-0_1.
[12] R. Belchior, A. Vasconcelos, S. Guerreiro, M. Correia, A survey on blockchain interoperability: Past, present, and future trends, ACM Comput. Surv. 54 (8) (2022) 168:1–168:41, http://dx.doi.org/10.1145/3471140.
[13] K. Ren, N. Ho, D. Loghin, T. Nguyen, B.C. Ooi, Q. Ta, F. Zhu, Interoperability in blockchain: A survey, IEEE Trans. Knowl. Data Eng. 35 (12) (2023) 12750–12769, http://dx.doi.org/10.1109/TKDE.2023.3275220.
[14] S. Crafa, M. Di Pirro, E. Zucca, Is Solidity solid enough? in: Financial Cryptography Workshops, in: LNCS, 11599, Springer, 2019, pp. 138–153, http://dx.doi.org/10.1007/978-3-030-43725-1_11.
[15] S. Team, Solidity documentation – language description – function calls – expressions and control structure – external function calls, 2024, https://docs.soliditylang.org/en/v0.8.27/control-structures.html#external-function-calls.
[16] S. Team, Solidity documentation – language description – inline assembly – memory safety, 2024, https://docs.soliditylang.org/en/latest/assembly.html#memory-safety.
[17] M. Bartoletti, L. Galletta, M. Murgia, A theory of transaction parallelism in blockchains, Log. Methods Comput. Sci. 17 (4) (2021) http://dx.doi.org/10.46298/LMCS-17(4:10)2021.
[18] S. Cui, G. Zhao, Y. Gao, T. Tavu, J. Huang, VRust: Automated vulnerability detection for Solana smart contracts, in: ACM SIGSAC Conference on Computer and Communications Security (CCS), ACM, 2022, pp. 639–652, http://dx.doi.org/10.1145/3548606.3560552.
[19] M.M.T. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M.P. Jones, P. Wadler, The extended UTXO model, in: Financial Cryptography Workshops, in: LNCS, 12063, Springer, 2020, pp. 525–539, http://dx.doi.org/10.1007/978-3-030-54455-3_37.
[20] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, R. Zunino, SoK: Unraveling Bitcoin smart contracts, in: Principles of Security and Trust, in: LNCS, 10804, Springer, 2018, pp. 217–242, http://dx.doi.org/10.1007/978-3-319-89722-6_9.
[21] L. Rosa, 2023, https://cardanofoundation.org/en/news/aiken-the-future-of-smart-contracts/.
[22] P. Vinogradova, O. Melkonian, Message-passing in the extended UTxO ledger model, in: Financial Cryptography Workshops, 2024, To appear.
[23] M. Bartoletti, A. Bracciali, C. Lepore, A. Scalas, R. Zunino, A formal model of Algorand smart contracts, in: Financial Cryptography and Data Security, in: LNCS, 12674, Springer, 2021, pp. 93–114, http://dx.doi.org/10.1007/978-3-662-64322-8_5.
[24] L. Luu, D. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: ACM SIGSAC Conference on Computer and Communications Security (CCS), ACM, 2016, pp. 254–269, http://dx.doi.org/10.1145/2976749.2978309.

[25] P. Caversaccio, A historical collection of reentrancy attacks, 2024, https://github.com/pcaversaccio/reentrancy-attacks.
[26] D. Muhs, Smart contract security field guide – reentrancy, 2023, https://scsfg.io/hackers/reentrancy/.
[27] S. Smolka, J. Giesen, P. Winkler, O. Draissi, L. Davi, G. Karame, K. Pohl, Fuzz on the beach: Fuzzing Solana smart contracts, in: ACM CCS, 2023, pp. 1197–1211, http://dx.doi.org/10.1145/3576915.3623178.
[28] S.S. Kushwaha, S. Joshi, D. Singh, M. Kaur, H.-N. Lee, Ethereum smart contract analysis tools: A systematic review, IEEE Access 10 (2022) 57037–57062, http://dx.doi.org/10.1109/ACCESS.2022.3169902.
[29] N. Ivanov, C. Li, Q. Yan, Z. Sun, Z. Cao, X. Luo, Security threat mitigation for smart contracts: A comprehensive survey, ACM Comput. Surv. 55 (14s) (2023) 326:1–326:37, http://dx.doi.org/10.1145/3593293.
[30] I. Garfatta, K. Klai, W. Gaaloul, M. Graiet, A survey on formal verification for Solidity smart contracts, in: Australasian Computer Science Week, 3, ACM, 2021, pp. 1–10, http://dx.doi.org/10.1145/3437378.3437879.
[31] Z. Sun, X. Luo, Y. Zhang, Panda: Security analysis of algorand smart contracts, in: USENIX Security Symposium, 2023, pp. 1811–1828.
[32] M. Bartoletti, F. Fioravanti, G. Matricardi, R. Pettinau, F. Sainas, Towards benchmarking of Solidity verification tools, in: Workshop on Formal Methods for Blockchains, in: Open Access Series in Informatics (OASIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, To appear.
[33] S. Wesley, M. Christakis, J.A. Navas, R.J. Trefler, V. Wüstholz, A. Gurfinkel, Verifying Solidity smart contracts via communication abstraction in SmartACE, in: Verification, Model Checking, and Abstract Interpretation (VMCAI), in: LNCS, 13182, Springer, 2022, pp. 425–449, http://dx.doi.org/10.1007/978-3-030-94583-1_21.
[34] J. Park, T. Zhang, W. Grieskamp, M. Xu, G. Di Giacomo, K. Chen, Y. Lu, R. Chen, Securing Aptos framework with formal verification, in: Workshop on Formal Methods for Blockchains, in: Open Access Series in Informatics (OASIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, To appear.
[35] B. Bernardo, R. Cauderlier, G. Claret, A. Jakobsson, B. Pesin, J. Tesson, Making Tezos smart contracts more reliable with Coq, in: ISoLA, Springer, 2020, pp. 60–72.
[36] M. Milo, E.H. Nielsen, D. Annenkov, B. Spitters, Finding smart contract vulnerabilities with ConCert's property-based testing framework, in: Workshop on Formal Methods for Blockchains, in: Open Access Series in Informatics (OASIcs), 105, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 1–13, http://dx.doi.org/10.4230/OASIcs.FMBC.2022.2.
[37] Y. Nishida, H. Saito, R. Chen, A. Kawata, J. Furuse, K. Suenaga, A. Igarashi, Helmholtz: A verifier for Tezos smart contracts based on refinement types, New Gener. Comput. 40 (2022) 507–554, http://dx.doi.org/10.1007/s00354-022-00167-1.
[38] L. Olivieri, L. Negrini, V. Arceri, T. Jensen, F. Spoto, Design and implementation of static analyses for Tezos smart contracts, Distrib. Ledger Technol. (2024) http://dx.doi.org/10.1145/3643567.
[39] G. Bau, A. Miné, V. Botbol, M. Bouaziz, Abstract interpretation of Michelson smart-contracts, in: ACM SIGPLAN Workshop on the State of the Art in Program Analysis, 2022, pp. 36–43, http://dx.doi.org/10.1145/3520313.3534660.
[40] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M.H. Au, X. Zhang, An adaptive gas cost mechanism for Ethereum to defend against under-priced DoS attacks, in: Information Security Practice and Experience, in: LNCS, Springer, 2017, pp. 3–24.
[41] D. Perez, B. Livshits, Broken metre: Attacking resource metering in EVM, in: Network and Distributed System Security Symposium (NDSS), The Internet Society, 2020.
[42] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, X. Zhang, TokenScope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in Ethereum, in: ACM CCS, 2019, pp. 1503–1520, http://dx.doi.org/10.1145/3319535.3345664.
[43] Y. Wang, A. Bracciali, T. Li, F. Li, X. Cui, M. Zhao, Randomness invalidates criminal smart contracts, Inform. Sci. 477 (2019) 291–301, http://dx.doi.org/10.1016/J.INS.2018.10.057.
[44] G. Blaut, X. Ma, K. Wolter, Exploring randomness in blockchains, in: IEEE Int. Conf. on Blockchain and Cryptocurrency, IEEE, 2023, pp. 1–5, http://dx.doi.org/10.1109/ICBC56567.2023.10174962.
[45] P. Qian, J. He, L. Lu, S. Wu, Z. Lu, L. Wu, Y. Zhou, Q. He, Demystifying random number in Ethereum smart contract: Taxonomy, vulnerability identification, and attack detection, IEEE Trans. Softw. Eng. 49 (7) (2023) 3793–3810, http://dx.doi.org/10.1109/TSE.2023.3271417.
[46] S. Micali, M.O. Rabin, S.P. Vadhan, Verifiable random functions, in: Symposium on Foundations of Computer Science, (FOCS), IEEE Computer Society, 1999, pp. 120–130, http://dx.doi.org/10.1109/SFFCS.1999.814584.
[47] L. Mazurek, EthVer: Formal verification of randomized ethereum smart contracts, in: Financial Cryptography and Data Security Workshops, in: LNCS, 12676, Springer, 2021, pp. 364–380, http://dx.doi.org/10.1007/978-3-662-63958-0_30.
[48] P. Sharma, R. Jindal, M.D. Borah, A review of smart contract-based platforms, applications, and challenges, Clust. Comput. 26 (1) (2023) 395–421, http://dx.doi.org/10.1007/S10586-021-03491-1.
[49] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, M. Imran, An overview on smart contracts: Challenges, advances and platforms, Future Gener. Comput. Syst. 105 (2020) 475–491, http://dx.doi.org/10.1016/j.future.2019.12.019.
[50] Á.J. Varela-Vaca, A.M.R. Quintero, Smart contract languages: A multivocal mapping study, ACM Comput. Surv. 54 (1) (2021) 1–38, http://dx.doi.org/10.1145/3423166.
[51] S. Dhaiouir, S. Assar, A systematic literature review of blockchain-enabled smart contracts: platforms, languages, consensus, applications and choice criteria, in: Research Challenges in Information Science (RCIS), Springer, 2020, pp. 249–266, http://dx.doi.org/10.1007/978-3-030-50316-1_15.
[52] A. Vacca, A. Di Sorbo, C.A. Visaggio, G. Canfora, A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges, J. Syst. Softw. 174 (2021) 110891, http://dx.doi.org/10.1016/j.jss.2020.110891.
[53] W. Zou, D. Lo, P.S. Kochhar, X.D. Le, X. Xia, Y. Feng, Z. Chen, B. Xu, Smart contract development: Challenges and opportunities, IEEE Trans. Softw. Eng. 47 (10) (2021) 2084–2106, http://dx.doi.org/10.1109/TSE.2019.2942301.
[54] N. Atzei, M. Bartoletti, S. Lande, N. Yoshida, R. Zunino, Developing secure Bitcoin contracts with BitML, in: ACM ESEC/SIGSOFT FSE, ACM, 2019, pp. 1124–1128, http://dx.doi.org/10.1145/3338906.3341173.
[55] L. Brünjes, M.J. Gabbay, UTxO- vs account-based smart contract blockchain programming paradigms, in: ISoLA, in: LNCS, 12478, Springer, 2020, pp. 73–88, http://dx.doi.org/10.1007/978-3-030-61467-6_6.
[56] P. Vinogradova, O. Melkonian, P. Wadler, M. Chakravarty, J. Krijnen, M.P. Jones, J. Chapman, T. Ferariu, Structured contracts in the EUTxO ledger model, in: Workshop on Formal Methods for Blockchains, in: Open Access Series in Informatics (OASIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, To appear.
[57] B. Hu, Z. Zhang, J. Liu, Y. Liu, J. Yin, R. Lu, X. Lin, A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems, Patterns 2 (2) (2021) 1–51, http://dx.doi.org/10.1016/J.PATTER.2020.100179.
[58] S. Rouhani, R. Deters, Security, performance, and applications of smart contracts: A systematic survey, IEEE Access 7 (2019) 50759–50779, http://dx.doi.org/10.1109/ACCESS.2019.2911031.
[59] T.M. Hewa, Y. Hu, M. Liyanage, S.S. Kanhere, M. Ylianttila, Survey on blockchain-based smart contracts: Technical aspects and future research, IEEE Access 9 (2021) 87643–87662, http://dx.doi.org/10.1109/ACCESS.2021.3068178.
[60] A. Voloder, M. Di Angelo, Comparison of smart contract platforms from the perspective of developers, in: ICBC, in: LNCS, 14206, Springer, 2023, pp. 104–118, http://dx.doi.org/10.1007/978-3-031-44920-8_7.
[61] R.M. Parizi, Amritraj, A. Dehghantanha, Smart contract programming languages on blockchains: An empirical evaluation of usability and security, in: International Conference on Blockchain (ICBC), Springer, 2018, pp. 75–91, http://dx.doi.org/10.1007/978-3-319-94478-4_6.
[62] S. Farshidi, S. Jansen, S. España, J. Verkleij, Decision support for blockchain platform selection: Three industry case studies, IEEE Trans. Eng. Manage. 67 (4) (2020) 1109–1128, http://dx.doi.org/10.1109/TEM.2019.2956897.

Massimo Bartoletti leads the blockchain research group at the University of Cagliari, Italy. His research activity concerns the development of tools and techniques for the specification, analysis and verification of smart contracts, with a special emphasis on security and formal methods. He is principal investigator of R&D projects on blockchain technologies, and program committee member of top-tier conferences on blockchain technologies, including ACM CCS - blockchain track and Financial Cryptography. He is the organizer of the DLT Workshop series, of the Scientific School on Blockchain & DLT series, and of the Workshop on Distributed Ledger Technologies and Formal Methods series. Massimo Bartoletti has published over 40 scientific papers on blockchain technologies since 2016, and it has presented his research in top-tier conferences like ACM CCS, Financial Cryptography, Computer Security Foundations, IEEE Security and Privacy Europe, and ESEC/SIGSOFT FSE.

Lorenzo Benetollo is a Ph.D. student of the Italian National Ph.D. Program in Blockchain and Distributed Ledger Technology coordinated by the University of Camerino. Previously, he got his Master's Degree in Computer Science at Ca' Foscari University. He is interested in blockchain technology, programming languages and security. Currently, his research is focused on the security and safety of smart contract programming languages for different blockchains. He has experience programming Solidity smart contracts on EVM blockchains and developing Web3 applications.



Michele Bugliesi is Professor of Computer Science at Ca 'Foscari University of Venice since 2006. At Ca' Foscari he has held several institutional roles, as Department Head (2009-2014), Member of the Academic Senate (2006-2009,2012-2014), Rector (2014 – 2020). Member of the scientific committee of major international conferences, he has coordinated several research projects at national and European level. In 2013, he was the co-recipient of the award for the Best EATCS Theory Paper at the European Joint Conferences on Theory and Practice of Software (ETAPS 2013). His academic research has always centered on the foundational aspects of programming languages as well as on the analysis and formal verification of software and programming systems, with specific focus on safety and security. He is the author of over 100 publications in top international journals and refereed conference proceedings on these topics.

Silvia Crafa is associate professor in mathematical logics since 2021 and formerly researcher in computer science at Universita' di Padova, Italy (2005-2021). She is member of IFIP Working Group n.1.8: Concurrency Theory since September 2012. She has been visiting professor at the University Paris Diderot in February 2013 and at University Paris Est-Creteil in May 2017. In 2014 she collaborated with a team of IBM Research, USA, to the study of the formal semantics of the X10 programming language for High-Performance Computing. She has been member of the Working Group on Ethics of Informatics Europe, ACM Europe Council and ACM Europe Policy Committee to write the white paper "When Computers Decide: European Recommendations on Machine-Learned Automated Decision Making", presented to the European Commission in March 2018. Silvia Crafa's research is characterized by three complementary lines: foundational aspects and formal methods for distributed systems, type systems and theory of programming languages and the interdisciplinar study of epistemic aspects of computer science with applications in computational law. Finally, she explores the social impact of new digital technologies, both by means of research collaborations and activities of knowledge transfer.

Giacomo Dal Sasso received the Master degree in computer science in February 2024 with the thesis "Linear typing for resource-aware programming" at Universita' di Padova, Italy. Formerly he received the degree in computer engineering in 2020 at Universita' di Padova, Italy. He has long working experience in the field of embedded systems programming.

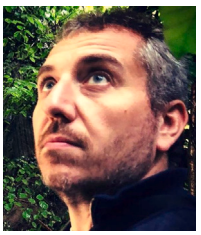
Roberto Pettinau is a Master's student in Computer Science & Engineering at the Technical University of Denmark. During his Bachelor studies at the University of Cagliari he designed and developed AlgoML, a novel smart contract language for building smart contracts that focuses on simplicity and security. The project not only secured first place at the Schelling Point Virtual Hackathon 2022 but has also garnered the attention of the Algorand Foundation. Roberto has later participated in multiple workshops and hackathons including the Blockchain and Distributed Ledger Technology School and International School on Algorand Smart Contracts. After his bachelor studies, he has worked in the industry as a Lead Blockchain Engineer at DeCash, in which he developed and audited multiple secure in-production smart contracts. Building upon his practical experiences, he resumed his studies, and shifted his focus to Ethereum and formal methods, studying techniques for the analysis and formal verification of programs and smart contracts. Roberto is currently writing his thesis on techniques for formally reasoning about probabilistic programs.

Andrea Pinna received the M.S. degree in electronic engineering from the University of Cagliari in 2012 and the Ph.D. degree in computer engineering from the University of Cagliari in 2018. Since 2023, he has been an assistant professor at the Department of Mathematics and Computer Science of the University of Cagliari where he teaches blockchain technology for master's degree students in computer science. He is the author of over 40 research papers and his research interests concern the study of blockchain technology and its applications. His topics of interest include the study of smart contracts, the engineering aspects of the development of decentralized applications and their interoperability, and the enhancement of software sustainability thanks to blockchain technology. He also dealt with the study of data stored inside the blockchain, network features, and users' behaviors.

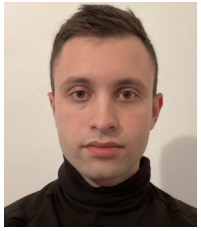
Mattia Piras graduated in 2023 from the University of Cagliari with a bachelor's degree in Computer Science. He is currently a master's student in Computer Science, specializing in Cloud and Security. His research interests, which began with his thesis, revolve around Blockchain technology. In particular, he focuses on the Tezos ecosystem with the goal of developing and executing reports on the costs of smart contracts on the platform. He aims to unveil all the functions and mechanisms of this platform using high-level smart contract programming languages.

Sabina Rossi received the Ph.D. degree in computational mathematics and informatics from the University of Padova, in 1994. She has been a Visiting Professor with Universit'e Paris 7, in 2007, and a Research Fellow with the Universit'e Catholique de Louvain-la-Neuve, Belgium, in 1997. She is currently a Full Professor in computer science with the University Ca' Foscari of Venice. Sabina Rossi is head of the research unit of Venive for the PRIN Project, Call 2020: "Noninterference and Reversibility Analysis in Private Blockchains (NiRvAna)" Prot. 20202FCJMH. Her research interests lie primarily in the area of the analysis and verification of programs and computer systems through the use of formal methods and mathematical models. Her expertise primarily focuses on utilizing theoretical models to ensure the accuracy and assess the efficiency of various systems, including ad hoc mobile wireless networks, systems incorporating fork-join constructs, distributed systems featuring load balancing, and blockchain systems.

Stefano Salis is a Master's student in Computer Science at Università degli Studi di Cagliari, Italy. He earned his bachelor's degree in computer science in September 2020 with a thesis titled "An approach to the Application of Business Intelligence" where he applied the business intelligence and data science methodologies within the blockchain environment, investigating some of the most prominent cryptocurrency networks of that period. In addition to blockchain technology, he is particularly interested in graphics engines (such as Unity and Unreal Engine).

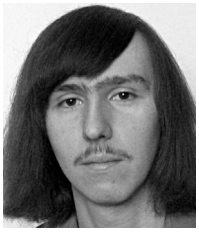


Alvise Spanò is a Researcher in Computer Science at Ca' Foscari University of Venice in Italy. He has a strong interest in programming languages, with a particular focus on functional programming, compilers, type systems, software validation, and correctness. Recently, he has been exploring smart contract languages for the blockchain, with a special emphasis on the implications of advanced type systems on asset management and security. Besides these research topics, he has experience in various areas of software engineering, including IoT, library design, software architectures, and type-disciplined programming methodologies. Prior to pursuing an academic career, he worked as a senior software developer for over two decades in the industry. Among his major open-source contributions, he is the creator of the functional programming language Lw and the text generator Polygen.



Viacheslav Tkachenko is a Master's student in Computer Science at the University of Cagliari in Italy. After graduating in 2022 with a Bachelor's degree in computer science, he developed a background in public administration with an emphasis on tenders and procurement. Currently, Viacheslav works closely with a research team at his university, focused on blockchain technology, cryptocurrencies, and smart contracts. His particular interest orbits around the landscape of smart contract development within the Solana blockchain ecosystem. Cloud computing technology is another area of interest for Viacheslav, which he is actively utilizing in his current endeavors. His Master's thesis delves into the intricacies of blockchain distributed and decentralized systems, with a focus on the Solana blockchain. Through his research, he explores the development and functionalities of smart contracts, conducting comprehensive analyses of deployment costs, interactions, and vulnerabilities associated with such contracts.

Roberto Tonelli is a Full Professor at the University of Cagliari's Department of Mathematics and Computer Science, where he's been Vice-Dean and Dean of the Ph.D. school for about 6 years. He is specialized in Blockchain Software Engineering. He has been awarded for his influential blockchain research (50 Topmost influential paper in 2018) by the Blockchain Connect Conference - SanFrancisco Jan 2019, with the participation of Vitalik Buterin. He is a leading author, recognized among the first six authors in the world, on Blockchain Oriented Software Engineering (BOSE), an acronym he coined in 2017 with other authors. Besides running international conferences and workshops co-located with ICSE and SANER, he founded the academic spin-off "Agile By Chain", focusing on blockchain technologies and applications, and organizes an annual international and well-recognized blockchain technology summer school. As a National Appointee for MISE (former Italian Ministry of Economical Development), he works on the European Blockchain Partnership and manages nodes of the BESU permissioned blockchain for the University of Cagliari's Italian Blockchain Service Infrastructure (IBSI). He has got two Ph.D. titles, one in Physics and the other in Software Engineering. He has been visiting researcher at the EECS of Berkeley University, California, in 2000 and 2001 and later on in 2006/07. He authored more than 150 papers.



Roberto Zunino is associate professor in computer science at the University of Trento, Italy. His main research line is the development of formal languages to model and verify distributed systems, leveraging techniques from programming languages theory and theoretical computer science. In particular, he focused on how to achieve and prove the security of systems working in distributed untrusted environments. He started his research on blockchain technologies in 2017, focusing on protocols for the safe and efficient execution of smart contracts in the UTXO model. He has published over 70 papers in international conferences and journals, including 14 papers on blockchain technologies. He was awarded the first Smart Contract Research Forum Impact Award in 2021 for the BitML language to express smart contracts on top of Bitcoin. He is a core member of the national DLT working group.