AlgoMove – A Move Embedding for Algorand

Lorenzo Benetollo
Ca' Foscari University of Venice, Italy
University of Camerino, Italy
lorenzo.benetollo@unive.it

Michele Bugliesi
Ca' Foscari University of Venice, Italy
bugliesi@unive.it

Silvia Crafa
University of Padua, Italy
silvia.crafa@unipd.it

Sabina Rossi
Ca' Foscari University of Venice, Italy
sabina.rossi@unive.it

Alvise Spanò
Ca' Foscari University of Venice, Italy
alvise.spano@unive.it

***Abstract*—As applications based on Distributed Ledger Technology (DLT) gain popularity, the wide range of vulnerabilities that have affected existing blockchains (most notably Ethereum and Solidity-based applications) has motivated renewed interest in the design of programming languages capable of providing more adequate abstractions for managing digital assets and their access control policies. These mechanisms are crucial to certify that applications are safe and secure before deploying them on the target blockchains.**

Venturing into this challenge, we focus on Move, currently one of the most promising programming languages for resources and digital assets management with the aim to investigate its effectiveness in the realm of general-purpose smart contract development, and the long-term goal to identify the design principles and language-based techniques for the safe and secure development of asset-based DLT applications. As a first step in that direction, in the present paper, we develop AlgoMove, a Move embedding on Algorand. In addition to providing new insight into the nature of digital assets, the embedding is noteworthy in its own right. It provides Algorand/TEAL developers with a friendly API that aligns with their familiar programming patterns, while at the same time leveraging Move's mechanisms of static typing and security verification to offer certified, language-level protection against double spending and other resource-related pitfalls commonly encountered in DLT applications.

***Index Terms*—Smart contract development, language-based techniques, digital-asset management.**

I. Introduction

Blockchain platforms have evolved significantly over the past decade. Initially intended as a support for distributed micropayments and other money-transfer applications, they have progressively turned into expressive frameworks providing more and more powerful tools for asset management and access control with smart contracts.

Along this evolution, a steadily growing number of security incidents have affected existing blockchains, causing significant damage. According to the SlowMist Annual Report 2022 (cf. Figure 1 below) the major causes for the 303 incidents that occurred in 2022 were smart contract vulnerabilities and other design flaws and loopholes, with a total cost of nearly US$1.1 billions.

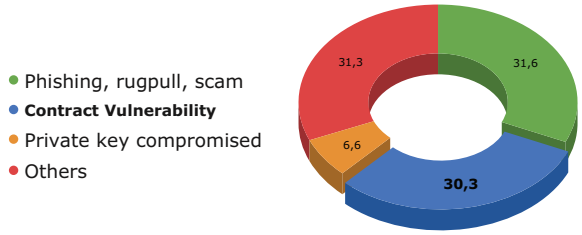

Fig. 1. Blockchain attack vectors (source: [9])

As a result, a new wave of awareness is gaining ground that more adequate tools are needed to build safe and secure smart contracts and web3 applications, leveraging the advances from programming language theory and implementation.

The Move programming language [1] developed by the (subsequently dismissed) Libra/Diem project represents one, perhaps the most significant step in that direction. Motivated by the vision that "providing first-class abstractions for the key asset management concepts would significantly improve both the safety of smart contracts and the productivity of smart contract programmers" [8], Move is designed as a cross-platform, embedded language with a simple core built around generic concepts such as *structs* integers and references, specific abstractions for resources and digital assets, but no blockchain-specific concepts like accounts, transactions, time, cryptography, etc. These features, instead, must come with the blockchain platform that integrates Move, so that developers may largely count on the core language elements and toolchain to add blockchain-specific features via code that builds on top of the core components. This kind of development has gained popularity recently, as witnessed by the number of Move-based chains under active development, among which, most notably, Aptos [6] and Sui [13].

This research was funded by Ministero dell'Università e della Ricerca (MUR), issue D.M. 351/2022, under the National Recovery and Resilience Plan (NRRP), and partially supported by the PRIN project NiRvAna – Noninterference and Reversibility Analysis in Private Blockchains, and by the project SERICS (PE00000014) under the MUR NRPP funded by the European Union - NextGenerationEU.

2834-9946/23/$31.00 ©2023 IEEE
DOI 10.1109/Blockchain60715.2023.00019

Our standpoint is that this new wave of interest in safe, platform-independent programming abstractions for asset management is much welcome, and that further research and effort should be invested to promote the widest possible adoption of programming languages supporting a principled design for the variety of DLT applications – tokens, NFTs, DeFi, legal and real estate applications, ... etc. – that blockchains are progressively meant to support.

Main Contributions

Motivated by that vision, in the present paper we take Move as our reference programming language and develop ALGOMOVE, a new blockchain embedding for the core language. Unlike the recent work carried out by Aptos and Sui, which are both Move-based platforms, our present effort is targeted at Algorand [2], the well-known blockchain originated by Silvio Micali's "Pure Proof-of- Stake" consensus mechanism, highly reputed for its security and performance effectiveness.

Our purpose and contribution to this endeavour are two-fold.

- By making Move's high-level programming abstractions for asset management and smart contracts available for Algorand, we provide a new, effective ground for safe DApp design and development over the Algorand blockchain. Specifically, ALGOMOVE offers Algorand/TEAL developers a new, friendly API to design applications based on their familiar programming patterns while at the same time being supported by Move's mechanisms of static typing and security verification. As a result, ALGOMOVE smart contracts enjoy Move's certified guarantees against double-spending and strong language-level protection against other resource-related pitfalls often found in DLT applications. This represents a significant step forward over the current TEAL and PyTEAL programming practice available for Algorand.
- By developing our experiment with a new, independently developed blockchain, we aim to assess the effectiveness of Move's core abstractions for digital asset management and to gather deeper insight into the different mechanisms available on existing platforms. Based on that, the ultimate goal of our present effort is to isolate the design principles and language-based techniques for the safe and secure development of asset-based DLT applications.

Related Work

Blockchain programming languages. The substantial body of existing R&D work on blockchain programming languages is only loosely related to our present endeavour. Solidity [4], the first and still dominant smart contract language, is all but cross-platform, being specifically tailored for the Ethereum Blockchain as the result of an explicit design choice. Also, the fragility of the Solidity design, as it emerged from the wealth of incidents experienced since its initial adoption (e.g., the reentrancy exploit and many other attacks), is in straight contrast with the foundational nature distinctive of Move. Indeed, our contention is that the development of sound, cross-platform programming languages like Move will be the best answer to the shortcomings suffered by Solidity.

Rust [10] is a safer alternative to Solidity. Launched in 2011, Rust was conceived as a general-purpose programming language offering fast, efficient execution and strong memory safety guarantees thanks to its ownership model and expressive type system. Its use for smart contract programming came only later, with its adoption as the primary programming language for Solana [14]. Build on Rust, from which it inherits a number of the benefits of the original Rust design, Move provides substantially stronger support for safety and composability thanks to its simpler programming model and its enhanced type and bytecode verification systems (cf. [11] for a thorough comparison between Move and Rust).

PyTeal [5] is the new tool provided by the Algorand development environment to encode smart contracts using Python. Though interesting in that it provides a more convenient and familiar syntax for developers, PyTeal is simply a Python library for generating programs in Algorand's native language TEAL and, as such, it retains the inherent fragility of TEAL's assembly-level abstractions for data representation and control.

Move Blockchain Embeddings. Several blockchain platforms have adopted Move as their primary programming language, including 0L [3], and Starcoin [7] besides the already mentioned Aptos and Sui. As noted earlier on, our purpose is orthogonal to these experiments, as they are intended to build an integrated Move-based blockchain from the ground up, whereas ALGOMOVE targets an existing and fully developed blockchain. As a consequence, though ALGOMOVE does share some similarities with these platforms – notably, the presence of an API framework to inter-operate with the underlying blockchain – the resulting technical development is inherently different and more challenging in various respects, as it requires a non-trivial effort to reconcile the many design gaps between the Move and Algorand Virtual Machines.

Paper Plan

In Section II we introduce the background material on Move and Algorand required to make the paper self-contained. In Section III we give an overview of the ALGOMOVE programming model and show its effectiveness for smart contract development on Algorand. In Section IV we discuss the structure of the ALGOMOVE framework. We conclude in Section V with final remarks and a brief discussion of our plans for future work. For the sake of clarity and improved readability, some of the technical aspects of the ALGOMOVE framework are omitted from the main body of the paper and reported in the Appendix.

II. BACKGROUND

A. Programming with resources in Move

The Move programming model is centered around a few, simple principles. Smart contracts are published as modules consisting of `struct` definitions and functions: the former are

the fundamental building block for representing data; the latter provide the only interface for the module clients to create, access and/or modify the module data structures, thus giving modules full control over how their datatypes are operated with. The module functions are always executed on behalf of an authenticated blockchain account represented inside Move as the `signer` of the account. The combination of these *access control* mechanisms is one of the two fundamental components of Move's powerful support for resource management. The second component is the language static typing system based on *linear types*, which ensures *scarcity*: once created (by a privileged module function), the type system construes structs as first-class resources that cannot be copied or implicitly discarded, only moved between program storage locations or passed around between functions.

The linear restrictions imposed by the type system are lifted for structs enhanced with *abilities*, to qualify them as *values* that can be duplicated (the `copy` ability) and/or discarded (the `drop` ability), i.e. not transferred, hence effectively destroyed as the execution flows through the program. Two additional abilities, `key` and `store`, allow structs to be stored persistently (on the underlying blockchain) and, respectively, to exist inside structs held in persistent storage.

Different Move variants provide different persistent storage representations, quite naturally so given that storage representation is inherently blockchain specific. For our present purposes, we assume the original Diem-Move (sometimes called *Core Move*) representation defined in terms of the *global storage* abstraction, that is as a table of *(acct, struct)* pairs, keyed by *acct*, the addresses of the accounts holding the resources encoded by *struct*. A resource can be created or modified on the global storage under an account only by the account's signer, by invoking the `move_to` primitive. Global storage resources, in turn, can be accessed by requesting (`borrow`ing) a reference to the resource (via the account address under which the resource is stored).

B. Managing digital assets in Algorand

Initially conceived as simple TEAL scripts encoding authorization policies for micro-payments and other transfer protocols for Algos, the Algorand native currency, today Algorand Smart Contracts support fully general digital-asset management applications. The Algorand programming model revolves around two components: (Py)TEAL scripts encoding the application logic and an API providing access to the underlying blockchain system of transactions and Algorand Standard Assets (ASAs). Unlike Move resources, which are first class (i.e. values that can be passed as arguments to functions and returned from functions as results), Algorand assets are layer-1 entities that may only be manipulated by way of the system of transactions which are provided by the platform and are made available for invocation from within the (Py)TEAL scripts by the API.

The Algorand storage model is organized in two areas of persistent state. A first area, which we refer to as *account storage*, is structured around accounts much in the same way as in Move, though without the access restrictions imposed by the authentication mechanisms underlying Move's signer values. A second area, referred to as *contract storage* is instead associated with smart contracts[1]. Thus, unlike Move modules, Algorand Smart Contracts are stateful and may therefore count on the existence of their own local state much like Solidity smart contracts and more generally, like objects in object-oriented languages.

III. Programming in AlgoMove

We introduce AlgoMove and its programming model with a smart contract implementing a simple auction. We first look at a Move implementation that uses the `coin` module provided by the standard library available on existing Move platforms (e.g. Aptos).

```
module coin;

struct Coin<CoinType> has store {
  value: u64
}

public fun deposit<CoinType>(account_addr: address,
                             coins: Coin<CoinType>)
```

The module collects the `Coin` parametric struct type that represents fungible assets, and the functions available to programmers to exchange assets among their accounts. Our Move implementation, below, is itself parametric over the `CoinType` generic type parameter so that different auctions are free to accept bids in their custom assets.

```
module auction;

struct Auction has key {
  auctioneer: address,
  top_bidder: address,
  expired: bool
}

struct Bid<CoinType> has key {
  coins: Coin<CoinType>
}

public fun start_auction<CoinType>(acc: &signer,
                                   base: Coin<CoinType>) {
  let auctioneer = signer::address_of(acc);
  let auction = Auction {
      auctioneer,
      top_bidder: auctioneer,
      expired: false };
  move_to(acc, auction);
  move_to(acc, Bid { coins: base });
}

public fun bid<CoinType>(acc: &signer,
                         auctioneer: address,
                         coins: Coin<CoinType>)
    acquires Auction, Bid {
  let auction = borrow_global_mut<Auction>(auctioneer);
  let Bid { coins: top_bid } =
    move_from<Bid<CoinType>>(auction.top_bidder);
  assert!(!auction.expired);
  assert!(coin::value(&coins) > coin::value(&top_bid));
  coin::deposit(auction.top_bidder, top_bid);
  auction.top_bidder = signer::address_of(acc);
  move_to(acc, Bid { coins });
}
```

[1] Algorand adopts a different naming scheme: what we call account storage is Algorand *local state*, while our contract storage corresponds to the *global state*. Choosing alternative names is simply meant to clarify the role of each storage area.

```
public fun finalize_auction<CoinType>(acc: &signer)
    acquires Auction, Bid {
  let auctioneer = signer::address_of(acc);
  let auction = borrow_global_mut<Auction>(auctioneer);
  assert!(auctioneer == auction.auctioneer);
  auction.expired = true;
  let Bid { coins: top_bid } =
    move_from<Bid<CoinType>>(auction.top_bidder);
  coin::deposit(auctioneer_addr, top_bid);
}
```

The `auction` module defines the `Auction` and `Bid` datatypes together with three functions to start the process, make bids and close the auction. The `Auction` datatype encodes the auction state, including the address of the user who last made the bid and the address of the auctioneer. The current top bid is stored separately on the `Bid` struct, allowing linearity checks performed by the Move compiler to keep track of movements of fungible assets (represented by the `Coin` datatype) between bidders and the blockchain, ensuring no double spending occurs.

The auctioneer starts the bidding by invoking `start_auction`, which initializes the auction state as well as the base bid on the blockchain storage under the auctioneer account. Participants may join in by invoking the `bid` function with their bid amount: the auctioneer address is passed on to `bid` in order to request a reference to the auction state and update it as needed. The current top bid is updated separately by retrieving the previous `Bid` struct from the blockchain through a `move_from`, and then storing the new bid under the new bidder account through a `move_to`. To pay back participants whose bids have been outbid, the `deposit` function is invoked. Finally, the auctioneer may close the bids by invoking `finalize_auction`.

The ALGOMOVE structure of the auction module is similar to the Move implementation we just illustrated, with few key differences that arise from the different nature of the programming and storage models distinctive of the underlying Algorand/TEAL architecture.

First, as noted earlier, unlike Move modules, Algorand smart contracts may have a state. As a result, the auction state in our example may directly be associated with the auction smart contract and held on the auction contract storage rather than being associated with the auctioneer account used by the Move implementation. In other words, the Move auction module and the auctioneer's account can be combined within ALGOMOVE and integrated into the auction smart contract, which can therefore receive and store the bids deposited by the auction participants.

Secondly, while assets are first-class Move values (in fact, resources), Algorand provides its Standard Assets as layer-1 entities that may only be manipulated by corresponding layer-1 transactions. As a consequence, all the logic, and security checks associated with the management of assets are accounted for at layer-1, while at the language level there is no room for any form of flow and/or access control. ALGOMOVE rectifies this by providing Move-based mechanisms to introduce first-class datatypes to act as the language-based counterpart of Algorand layer-1 assets. Furthermore, ALGOMOVE provides a set of functions for handling assets as Move resources and at the same performing the corresponding layer-1 asset operations by executing the associated transactions. As a payoff, ALGOMOVE enjoys all the support from Move's static typing discipline for linear types and safe resource management.

The ALGOMOVE auction module is given below. The code should be easily understood based on its Move companion and the explanations we just provided.

```
module algomove_auction;

struct Auction<AssetType> has key {
  auctioneer: address,
  top_bid: Asset<AssetType>,
  top_bidder: address,
  expired: bool
}

public fun start_auction<AssetType>(base:Asset<AssetType>){
  let sender = get_sender();
  let auction = Auction<AssetType> {
    auctioneer: sender,
    top_bid: base,
    top_bidder: sender,
    expired: false };
  app_global_put(sender, auction);
}

public fun bid<AssetType>(auctioneer: address,
                          assets: Asset<AssetType>) {
  let Auction {auctioneer, top_bid, top_bidder, expired} =
    app_global_get<Auction<AssetType>>(auctioneer);
  assert!(!expired);
  assert!(get_amount(&assets) > get_amount(&top_bid));
  let sender = get_sender();
  transfer(top_bid, top_bidder);
  let new_auction = Auction<AssetType> {
    auctioneer,
    top_bid: assets,
    top_bidder: sender,
    expired: expired};
  app_global_put(auctioneer, new_auction);
}

public fun finalize_auction<AssetType>() {
  let sender = get_sender();
  let auction = app_global_get<Auction<AssetType>>(sender);
  assert!(sender == auction.auctioneer);
  let Auction { auctioneer, top_bid,
                top_bidder: _, expired: _ } = auction;
  transfer(top_bid, auctioneer);
  let new_auction = Auction<AssetType> {
    auctioneer,
    top_bid: assets,
    top_bidder: sender,
    expired: true };
  app_global_put(auctioneer, new_auction);
}
```

All function calls appearing in green are part of the ALGOMOVE framework, that is the Algorand wrapper on top of which we develop our Move embedding. A tiny excerpt of this layer, which we describe in more detail in Section IV, is given below.

```
struct Asset<AssetType> has store {
  id: u64,
  amount: u64
  owner: address
}

public fun transfer<AssetType>(asset: Asset<AssetType>,
                               receiver: address)
public fun get_amount<AssetType>(a: &Asset<AssetType>): u64
```

```
public fun get_sender(): address
public fun app_global_get<T: key>(k: vector<u8>): T
public fun app_global_put<T: key>(k: vector<u8>, data: T)
```

The `Asset` datatype provides the language-level counterpart of Algorand's layer-1 assets. This is where the Move type system comes into play: once a resource of type `Asset` has been minted or withdrawn, it can only be moved, not copied or dropped. The `id` field in the `Asset` datatype hosts the Algorand unique layer-1 identifier of the created asset. Notice that ALGOMOVE assets cannot be directly stored on-chain as they miss the `key` ability, but can be embedded into other structs (specifically, within the `Auction` struct) thanks to the `store` ability. The `transfer` function takes an asset and moves it to the receiver address invoking an Algorand transaction. The boundary between Move and Algorand is subtle here. The Move type system is in action for checking, statically, that assets are not copied: as a result no double spending may ever take place at the code level. While Algorand still lies beneath and performs transactions, including asset transfers, that are subject to run-time checking to prevent double-spending, using ALGOMOVE renders the checks performed by Algorand redundant.

The two `app_global_*` functions are plain stubs of the respective Algorand instructions for reading and writing to the global state, here turned into functions exhibiting a Move-like flavour in the prototype. Although Algorand does not support abilities and allows all datatype to be read or written on-chain, our framework enforces the Move tradition by putting the `key` constraint on the generic type parameter `T`, in the likes of the `move_to` and the `move_from` primitives. Most notably, though, the `app_global_get` in Algorand performs a copy of the data being read from the blockchain, thus its overall semantics substantially differ from a `borrow` or a `move_from`. That is the reason why, in the auction example, the `bid` function performs an `app_global_get` followed by a `app_global_put` for updating the state of the auction.

IV. THE ALGOMOVE FRAMEWORK

We may characterize the relationship between ALGOMOVE and Algorand along three dimensions: data model, computational model, and access control mechanisms. The following table provides as synthetic picture of the distinctive features of the two frameworks:

	Algorand	ALGOMOVE
Data Model	Assets as layer-1 objects	Assets as Move first-class objects (structs with abilities)
Computational Model	Transactions as the only computational device to operate with assets Opt-in as a computational enabler for asset exchanges among accounts	Move (primitives and user-defined functions) to compute with assets Opt-in and other ad-hoc primitives to store assets under accounts
Access Control	Account-based asset ownership Transaction checking	Account-based ownership (via the signer abstraction) Module-based scope restrictions (entry, public, friend qualifiers) Linear typing

In principle, ALGOMOVE appears to outperform Algorand/TEAL both as a platform for program design and as a development environment equipped with full support for static typing and security verification. In fact, by supporting assets as language-level objects, ALGOMOVE provides the full expressivity of Move to design complex asset management policies and asset-based applications. Algorand, instead, has only its (somewhat limited) transaction suite to offer to support the development of such applications. Also, ALGOMOVE access control mechanisms provide static protection against double spending and other safety & security threats affecting asset-based applications, whereas Algorand addresses such threats only with the run-time transaction-checking mechanisms supported by the underlying blockchain.

A. Framework Architecture

The proposed framework consists of two major components: a Move library, and a translation subsystem. The library (which we often refer to as the *framework* following the terminology adopted by other Move embeddings, like Aptos, Sui and StarCoin), provides the API for interoperating with Algorand, bundling a collection of Move modules that offer a variety of services specifically tailored to the underlying chain. The translation subsystem, in turn, converts Move bytecode into TEAL bytecode to be deployed for execution on Algorand. Due to space limitations, though, we are not delving into its details.

The framework can be logically split into three layers, implemented by three Move modules each relying on the underlying layer, as shown in Figure 2. The highest layer provides an abstraction for Algorand assets; the middle layer provides an API for performing transactions; the lowest layer consists of native functions that serve as stubs for TEAL instructions. Programs can freely call any layer from the application code.

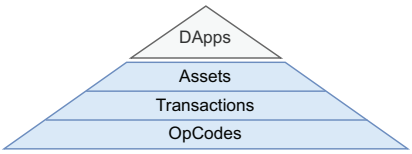

Fig. 2. A diagram of the ALGOMOVE framework layers. Each layer is built on top of the one below. However, applications are allowed to directly call any layer, for maximum control.

A breakdown of the library layers follows, from the lowest level to the highest.

The OpCodes Layer: the lowest layer consists of a Move module declaring function prototypes tagged with the *native* keyword, thus having no implementation. Each native

function here represents an opcode in TEAL that goes under the same name. Native functions have no implementation and serve as *stubs* mapping TEAL instructions: whenever a call to such natives is encountered during the translation process, the respective TEAL opcode is emitted. Function parameters stand for arguments that are expected to be found on the stack by the TEAL opcode, whereas the function result represents the output of the opcode pushed on the top of the stack after its execution. See the TEAL opcode reference for a full list [12].

The Transactions Layer: the middle layer provides an API for performing transactions, inner transactions and transaction groups. To create a transaction in TEAL a programmer has to fill a number of fields with the desired information and eventually submit the transaction for executing it. Our transaction layer abstracts this procedure by providing an easy-to-use API for a Move programmer to perform a transaction by calling a function and passing the relevant arguments.

The Assets Layer: the topmost layer provides datatypes and functions for creating and manipulating Algorand assets. A tiny excerpt of it has been shown in the previous Section. Assets are parametric over a generic `AssetType`, reproducing the same programming pattern provided by the `Coin` type in the Move standard library. In Algorand, assets are layer-1 entities that can be created and transferred using the transaction system. The `Asset` type in ALGOMOVE, instead, represents first-class structured values that cannot be copied or dropped, which makes them subject to linearity checks by the Move compiler. Transferring assets using the `transfer` function triggers a transaction in Algorand, factually performing a payment from a sender to a receiver.

V. FINAL REMARKS

The growing influence and worldwide adoption of DLT applications across various critical production and societal sectors calls for urgent action to provide developers with principled tools and programming language techniques.

The experiment we have reported in the present paper aligns with this call to action as a first attempt to shed new insight into what such tools and techniques should provide and how. From this standpoint, the ALGOMOVE framework emerges as an interesting experiment, whose significance extends beyond its practical implications which involve providing innovative tools for secure DApp design on the Algorand blockchain. In fact, equally important is its role in forging a conceptual link between two well-developed, but previously unrelated approaches to smart contract development.

Needless to say, gaining a full understanding of the essence of digital assets and of the principles for their management on distributed ledger platforms will require substantially more effort both in theoretical research and in engineering work. Our own plans for future work include both further engineering experiments (of Move embeddings) with other platforms (e.g. Sui, Solana, Ethereum, UTXO frameworks), and theoretical work to assess the formal properties of the embeddings and to contribute to the construction of a solid semantic and type-theoretic framework for digital assets. We have already explored the feasibility of what we believe to be the main challenges of the Move embedding for Algorand, and we will discuss them in a future work, going into the details of the framework and the translation subsystem.

REFERENCES

[1] Sam Blackshear, David L. Dill, Shaz Qadeer, Clark W. Barrett, John C. Mitchell, Oded Padon, and Yoni Zohar. Resources: A safe language abstraction for money. *CoRR*, abs/2004.05106, 2020.

[2] Jing Chen and Silvio Micali. Algorand. Available at: https://arxiv.org/pdf/1607.01341, 2017.

[3] The 0L Network Community. Open, transparent & community driven. Available at: https://0l.network/, 2022.

[4] Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, USA, 1st edition, 2017.

[5] Algorand Foundation. PyTeal: Algorand Smart Contracts in Python. Available at: https://pyteal.readthedocs.io/en/stable/, 2022.

[6] Aptos Foundation. The aptos white paper. Available at: https://aptos.dev/aptos-white-paper, 2023.

[7] Starcoin Foundation. Introduction to Starcoin. Available at: https://starcoin.org/downloads/Starcoin_Whitepaper.pdf, 2022.

[8] Sui Foundation. Why we created sui move. Available at: https://blog.sui.io/why-we-created-sui-move/, 2023.

[9] SlowMist Inc. 2022 Annual Blockchain Security and AML Analysis Report. Available at: https://www.slowmist.com/report/2022-Blockchain-Security-and-AML-Analysis-Annual-Report(EN).pdf, 2022.

[10] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[11] Krešimir Klas @ Medium.com. Smart Contract Development: Move vs Rust. Available at: https://medium.com/@kklas/smart-contract-development-move-vs-rust-4d8f84754a8f, 2022.

[12] Algorand Developer Portal. Algorand developer docs. Available at: https://developer.algorand.org/docs/.

[13] The MystenLab Team. The Sui Smart Contracts Platform. Available at: https://github.com/MystenLabs/sui/blob/main/doc/paper/sui.pdf, 2023.

[14] Anatoly Yekovenko. Solana: A new architecture for a high performance blockchain v0.8.13. Available at: https://solana.com/solana-whitepaper.pdf, 2022.