Received 20 September 2024, accepted 7 November 2024, date of publication 11 November 2024, date of current version 20 November 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3495535

SURVEY

General-Purpose Languages for Blockchain Smart Contracts Development: A Comprehensive Study

LUCA OLIVIERI[1], VINCENZO ARCERI[2], BADARUDDIN CHACHAR[1], LUCA NEGRINI[1], FABIO TAGLIAFERRO[3], FAUSTO SPOTO[4], PIETRO FERRARA[1], AND AGOSTINO CORTESI[1]
[1]Department of Environmental Sciences, Informatics, and Statistics, Ca' Foscari University of Venice, 30123 Venice, Italy
[2]Department of Mathematical, Physical and Computer Sciences, University of Parma, 43121 Parma, Italy
[3]Equixly Srl, 37135 Florence, Italy
[4]Department of Computer Science, University of Verona, 37129 Verona, Italy

Corresponding author: Vincenzo Arceri (vincenzo.arceri@unipr.it)

This work was supported in part by the Bando di Ateneo per la Ricerca 2022, funded by the University of Parma under Grant MUR_DM737_2022_FIL_PROGETTI_B_ARCERI_COFIN, CUP: D91B210 05370003; in part by the Formal Verification of GPLs Blockchain Smart Contracts; in part by SEcurity and RIghts In the CyberSpace (SERICS) under Grant PE00000014—CUP H73C2200089001; and in part by Interconnected Nord-Est Innovation Ecosystem (iNEST) Projects funded by PNRR NextGeneration EU under Grant ECS00000043—CUP H43C22000540006.

ABSTRACT Smart contracts are undoubtedly one of the most successful and popular applications of the blockchain industry. They consist of computer programs that are stored in blockchain, typically immutable, allowing the creation of decentralized applications (DApps). Their source code describes how the blockchain's global state should evolve as a consequence of input received from transaction requests. There are two categories of programming languages for writing smart contracts: domain-specific languages (DSLs) and general-purpose languages (GPLs). The research community has spent a great effort for proposing, studying, and verifying DSLs, while the same is not true for GPLs applied to blockchain, although the most popular blockchains adopt them at different levels of the software development. This paper investigates the use of the most popular GPLs in smart contracts and DApps development. It first overviews blockchains that use GPLs for writing smart contracts. Then, it provides a taxonomy of how GPLs are used to develop smart contracts, highlighting limitations and challenges for each type of GPL. The goal of this study is to provide blockchain practitioners with a better comprehension of GPLs, while shedding light on this class of programming languages that are widespread in blockchain software.

INDEX TERMS General-purpose programming languages, smart contracts, decentralized applications, blockchain, distributed ledgers.

I. INTRODUCTION

Blockchain technology is no longer in its infancy. Bitcoin [1] provided the first successful solution for the double-spending problem, effectively implementing a network of programmable money. Later on, Ethereum [2] raised the interest in enforcing the execution of code through consensus rules with smart contracts and DApps, the most important applications of blockchain technology. In a nutshell, they are computer programs on the blockchain network that can receive specific inputs from transaction requests, process them according to the instructions contained within them, and possibly modify the global state of the blockchain. There are several languages for writing smart contracts and DApps and they can be divided into two classes: domain-specific languages (DSLs) and general-purpose languages (GPLs). DSLs are programming languages specifically designed and restricted for a given problem domain. An example is Solidity for the Ethereum blockchain. Instead, as the name suggests, GPLs are programming languages that are typically applicable across different software domains. Therefore, they are not strictly related to the development of smart contracts and DApps. Examples are Rust, Java, C++, and Go. In recent years these have become widespread in the programming of

The associate editor coordinating the review of this manuscript and approving it for publication was Barbara Guidi.

smart contracts. In fact, although the majority of the total value locked (TVL) is in the Ethereum blockchain and is therefore managed by smart contracts written in Solidity, the remaining part of TVL is managed by contracts written in other languages including GPLs. For instance, as reported by DefiLlama [3], as of mid-2024, smart contracts written in Rust handled Defi worth more than $8 billion, while smart contracts written in C, C++, and Java for several million dollars [4].

A. RESEARCH SCOPE AND CONTRIBUTION

This paper overviews, analyzes, classifies, and discusses, at high-level, the GPLs used for developing smart contracts and DApps, taking into consideration their features, limitations, issues, and applicability. In particular, this paper will discuss, answer, and argue about the following research questions:

RQ1: How are GPLs involved in the development of smart contracts and DApps?
RQ2: Why are GPLs used for programming smart contracts and DApps? What reasons lead to their choice?
RQ3: What challenges arise from writing smart contracts using GPLs?
RQ4: What are the current limitations of developing smart contracts using GPLs?

The main contribution of this paper is:

- to survey the GPLs involved in the smart contract development for the most popular blockchains;
- to investigate the reasons behind the adoption of GPLs in the development of smart contracts and decentralized applications;
- to provide a taxonomy related to the usage of GPLs in smart contract development;
- to discuss challenges and current limitations for each classification proposed in the taxonomy.

The motivation of this study comes from the fact that, while the research community has spent a great effort on DSLs (for instance, a representative example is the huge research literature about Solidity), the same has not happened for GPLs, although, as argued in this paper, most popular blockchains rely on them.

Hence, the aim of this paper is to focus on GPLs, in different ways, and with different scopes, in smart contract and DApp development. We think that this helps reaching a better comprehension of GPLs in blockchain, highlighting benefits and potential risks in adopting them and thus helping practitioners in selecting the more appropriate GPL for their purposes.

B. PAPER STRUCTURE

The rest of this paper is organized as follows. Section II describes at high-level the methodology adopted in this study, the search strategy, and the search findings. Section III provides an overview of the blockchains that support the development of smart contracts and DApps through GPLs.

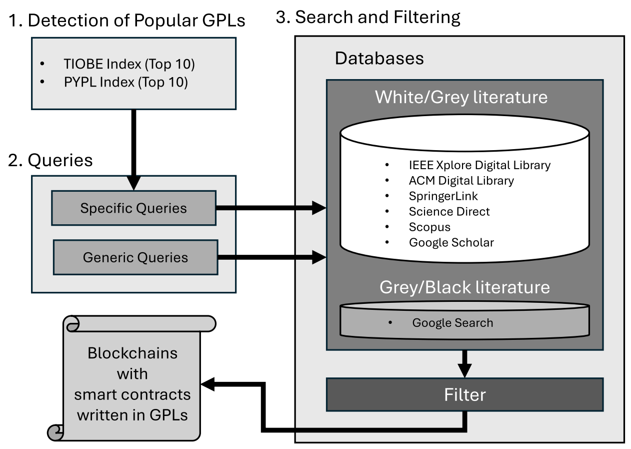


FIGURE 1. Workflow of the search strategy.

Section IV proposes a taxonomy of the different involvements of GPLs in popular blockchains. Section V investigates the reasons behind the popularity of this choice. Section VI analyzes the limitations and challenges of GPLs adoption. Section VII discuss the performance issues and the lack of empirical evaluations. Section VIII discusses selected related works. Finally, Section IX concludes the paper.

II. METHODOLOGY

We conducted a state-of-the-art investigation to identify blockchains supporting GPLs for the development and implementation of smart contracts and DApps.

We structured the search strategy as follow (see Figure 1): (i) detection of popular GPLs, (ii) definition of queries, (iii) search and result filtering. Each phase is described below.

A. DETECTION OF POPULAR GPLs

Given the rapid emergence of new programming languages, this study focuses exclusively on the most widely adopted and popular GPLs to ensure relevance and practical applicability. To determine these languages, we consider as reference two established indexes, namely TIOBE [5] and PYPL [6], as in June 2024 (see Table 1), that highlight the popularity of programming languages. In particular, the TIOBE index considers the number of search engine results for queries containing the name of a language. Instead, the PYPL index is computed by checking how often a programming language tutorial is searched, based on datasets extracted by using Google Trends.

Hence, we selected only GPLs excluding DSLs (e.g., SQL) from their ranks.

B. DEFINITION OF QUERIES

We structured our search strategy by querying both specific and generic keywords on popular academic databases, as we will explain later. The following keywords were used:

- specific keywords (one for each selected GPL):
  - `smart contracts <GPL>`, where `<GPL>` is the name of the GPL.

TABLE 1. Top 10 popular programming languages.

	TIOBE Index		*PYPL Index* (worldwide)	
Rank	**Language**	**Ratings**	**Language**	**Shares**
1	Python	15.38%	Python	29.06%
2	C++	10.03%	Java	15.97%
3	C	9.23%	JavaScript	8.70%
4	Java	8.40%	C#	6.73%
5	C#	6.65%	C/C++	6.40%
6	JavaScript	3.32%	R	4.75%
7	Go	1.93%	PHP	4.57%
8	SQL	1.75%	TypeScript	3.00%
9	Visual Basic	1.66%	Swift	2.76%
10	Fortran	1.53%	Rust	2.50%
-	Others languages	40.12%	Others languages	15.56%

- generic keywords:
  - `smart contract general purpose programming languages`
  - `smart contract any programming languages`
  - `multi programming language smart contracts`
  - `survey smart contract programming languages`

Regarding specific keywords, we performed the queries for the following GPLs: *Python*, *C++*, *C*, *Java*, *C#*, *Javascript*, *Go*, *Visual Basic*, *Fortran*, *R*, *PHP*, *TypeScript*, *Swift*, *Rust*.

Regarding generic keywords, we also considered `multi programming language` because, during the first iterations of the search, we realized that different blockchains supported more than one GPL.

C. SEARCH AND RESULT FILTERING

The literature review first involved systematical search in academic databases (i.e., IEEE Xplore Digital Library, ACM Digital Library, SpringerLink, Science Direct, Scopus, and Google Scholar) to collect white/grey literature. This required submitting 18 queries to each database. Then, the results have been filtered, starting from the year 2008, i.e., from the first appearance of the blockchain with Bitcoin, and sorted by relevance, resulting in 582455 references. However, when we started inspecting the first results, we discovered that the search engines introduced noise and lacked accuracy in the results after just a few records. Then, we decided to only consider the top 10 most relevant records for each query, i.e., 1019 references of white literature in total (some queries did return less than 10 records), then, since some queries reported common results, we removed duplicates, resulting in **471** references.

Additionally, we decided to collect relevant gray/black literature as well (e.g., websites, blog posts, and other non-academic sources of information) through Google Search, being blockchain technology primarily emerged from the efforts of practitioners and enthusiasts rather than academia. Therefore, much information on the state-of-the-art and new blockchain technologies can only be found through non-academic sources. Hence, as we previously did, we performed 18 queries, we filtered from the year 2008 and selected only the 10 highest ranked results. The result is **180** references.

Finally, blockchains were selected based on their relevance for the research questions, and coherence with the implementation of smart contracts and programming languages. The selection criteria requires that they have at least one development support of smart contracts or DApps through GPLs, coming from official sources, popular community teams, or industrial companies. Therefore, not all popular blockchains have been covered as they may not support development via GPLs, such as Bitcoin and Cardano [7] that only support DSLs [8], [9]. Moreover, we avoided also those blockchains that support frameworks written in GPLs only for deployment, such as Ethereum with its SDKs [10] or unofficial GPL libraries such as Jthereum [11].

Our findings are summarized in Table 2. Note that we found no results for the following GPLs: Fortran, PHP, R, Swift and Visual Basic. Specifically, Fortran and Visual Basic are largely used in legacy software, with little adoption in modern programming. Instead, the other languages are GPLs but are widely used in specific contexts, i.e., PHP is mainly involved for web application development, R for statistical computing and graphics purposes, and Swift for the development of Apple applications.

Moreover, for completeness, we reported all the GPLs supported for each blockchain in Table 2, although the queries were only made on the most popular ones.

III. BLOCKCHAIN PLATFORMS

This section provides a brief description of each blockchain reported in Table 2.

Algorand [12] is a blockchain that provides a common platform for building products and services in a fully decentralized, secure, and scalable way. It supports the execution of smart contracts and DApps through the Algorand Virtual Machine (AVM). Smart contracts and smart signatures are written in the Transaction Execution Approval Language (TEAL), an assembly-like language that is ultimately converted to AVM bytecode. Moreover, it is also possible to write smart contracts with Python, by using the PyTeal library [13], and also with the Reach DSL [14].

Casper Network [15] is a blockchain platform designed for enterprise and developer adoption, based on Proof-of-Stake consensus. Casper Network focuses on usability and upgradability, enabling developers to create upgradable smart contracts for enterprise applications. On the Casper platform, developers may write smart contracts and compile them to WebAssembly binaries.

Concordium [16] is a blockchain designed to balance privacy with regulatory compliance. It features built-in identity verification that ensures that users are accountable, while maintaining privacy through zero-knowledge proofs. The platform supports smart contracts and tokens, aiming for secure and scalable transaction processing. A standard library is provided for writing smart contracts in Rust.

TABLE 2. Blockchains platforms with smart contracts written in GPLs.

Platform	Framework/Library	Languages
Algorand	PyTeal	Python
	Reach	Reach
Casper Network	Casper Contract Library	Rust, TypeScript/AssemblyScript
Concordium	Concordium STD	Rust
Corda	Contract SDK	Java, Kotlin
Cosmos	Ignite (ex Tendermint Core)	Any language
	Cosmos SDK	Go
	CosmWasm	Rust
EOSIO	EOSIO Contract Development Toolkit	C++
Exonum	Exonum Binding	Java, Rust
Hotmoka	Takamaka	Java
Hyperledger Fabric	Hyperledger Fabric SDKs	Go, Java, Javascript
IOTA	IOTA	Rust, Go
Internet Computer Protocol (ICP) Network	Canister development kits (CDKs)	Rust, TypeScript, Python
Lisk	Lisk SDK	JavaScript
Neo	Neo SDK	Python, C#, Go, TypeScript, Java
Polkadot	Ink!	Rust
QANplatform	QANplatform	Any language
Solana	Solana SDK	Rust, C, C++
Stellar	SDK for Soroban contracts	Rust
Stratis Platform	.Net Core	C#
Tezos	SmartPy	Python, OCaml, TypeScript
	LIGO	Javascript, OCaml, Pascal, ReasonML
XPR Network	Contract SDK	TypeScript/AssemblyScript

Cosmos [17] is a public ecosystem of independent interconnected blockchains. The blockchain nodes are built by using *Tendermint Core* [18], recently rebranded as *Ignite* [19], a middleware that separates the application logic from the consensus and networking layers. This allows one to develop DApps written in any programming language that supports remote procedure calls, and replicate them on many machines [20]. However, the recommended development involves the use of *Cosmos SDK*, an open-source framework written in Go for the development of the application layer of the blockchain. A very peculiar instance of the Cosmos SDK module is *CosmWasm* [21], allowing the development of smart contracts in any language that can be compiled into WebAssembly (Wasm), a low-level language supported as compilation target by several GPLs [22]. In particular, the first and most used language for CosmWasm contracts is Rust.

EOSIO [23] is an open-source blockchain software protocol with industry-leading transaction speed and a flexible utility. It is designed to be customizable and provides both public and private blockchain deployments, winking at the enterprise context. It supports the execution of smart contracts through a WebAssembly Virtual Machine (WasmVM). Smart contracts can be developed at high-level in C++, by using EOSIO.CDT (EOSIO Contract Development Toolkit).

Exonum [24] is a blockchain framework designed by Bitfury for building decentralized applications. It offers enterprise-level features and custom modules. The smart contract are called *services* and can be developed in Java and Rust [25].

Hotmoka [26] is the abstract definition of a device able to store objects (data structures) in its persistent memory (its state) and can execute code on those objects. Although devices can be different from the nodes of a blockchain (for instance, they can be an Internet of Things (IoT) device), the most prominent application of Hotmoka nodes is, currently, the construction of blockchains whose nodes are Hotmoka nodes. Hence, such a device is called a Hotmoka node and such programs are known as smart contracts, taking that terminology from programs that run inside a blockchain. The language supported by Hotmoka's smart contract is a subset of Java called Takamaka [27].

Hyperledger Fabric [28] is a popular permissioned blockchain framework, designed to be adopted in the industrial context. It is supported by The Linux Foundation and other contributors such as IBM, Cisco, and Intel. In this blockchain, smart contracts and DApps are called *chaincode* and can be implemented by using the full set of instructions and features of popular GPLs such as Go, Node.js, and Java [29].

IOTA [30] is an open-source permissionless distributed ledger and cryptocurrency with the purpose of protecting the integrity and verifiability of data and values in the IoT context. It supports the execution of smart contracts through a virtual machine (VM) and VM plugins. The VM itself is a black box capable of executing deterministic code and can be extended dynamically by adding VM plugins. Currently, there are two such plugins, WasmVM and EthereumVM (EVM). WasmVM supports TinyGo (a subset of Go) and Rust. EthereumVM supports instead the Solidity programming language.

Internet Computer Protocol (ICP) [31] is a decentralized blockchain network developed by the DFINITY Foundation, designed to extend the functionality of the public Internet by enabling smart contracts and decentralized applications to run at web speed. In ICP, smart contracts are called *canisters* and they are WebAssembly programs with additional features that allow them to store persistent data, communicate with users and other canisters, and be managed by entities such as decentralized organizations.

Lisk [32] is a blockchain application platform to build interoperable blockchain applications and services with the *Lisk SDK*. The *Lisk SDK* is an open-source and modular SDK in JavaScript and TypeScript.

Neo [33], rebranded from *Antshares*, is an open-source decentralized blockchain with the purpose of creating a smart economy. That is, it leverages smart contracts to issue and manage digital assets over the blockchain network. As reported in its documentation [34], *Neo*'s tooling and infrastructure supports several GPLs for smart contracts. The compilation target of GPLs is the standard NEF (Neo Execution Format) that can be executed within NeoVM, a lightweight virtual machine.

Polkadot [35] is a fully-sharded blockchain built to connect and secure other blockchains, that can be public, permissionless, based on private consortium chains, or other Web3 technologies. Polkadot allows one to execute smart contracts written in WebAssembly. However, they can be developed by using *Ink!* [36], a subset of Rust, and then compiled into Wasm code.

QANplatform [37] is a blockchain platform focused on security and scalability. The platform leverages quantum-resistant cryptography, ensuring that it remains secure against future quantum computer threats. The blockchain infrastructure is hybrid and combines public and private blockchain features. Moreover, it proposes a new consensus algorithm called Proof-of-Randomness. Regarding smart contracts and DApps, it provides a QVM (QAN Virtual Machine) that allows the user to execute any kind of application, even a Linux Kernel [38].

Solana [39] is an open-source blockchain that proposes a new architecture based on a Proof-of-History (PoH) algorithm, i.e., a proof for verifying order and passage of time between events. It supports smart contracts and DApps written in Solana Berkeley Packet Filter bytecode (Solana BPF bytecode). As reported in the documentation [40], the foundation promotes the development through GPLs and officially supports SDKs for Rust, C, and C++.

Stellar [41] is a decentralized blockchain platform that aims to connect financial institutions, payment systems, and individuals globally through its native cryptocurrency called *Lumens*. Stellar operates on a consensus protocol called Stellar Consensus Protocol (SCP), which ensures fast transaction confirmation times and scalability. Smart contract support comes from Project Jump Cannon [42] and it targets WebAssembly [43].

Stratis Platform [44] is a blockchain-as-a-service (BaaS) solution designed to simplify the development, deployment, and management of blockchain applications. It is built on a blockchain network that is compatible with.NET.

TABLE 3. Target languages for each blockchain.

Blockchain	Target Languages
Algorand	Transaction Execution Approval Language (TEAL)
Casper Network	WebAssembly
Concordium	WebAssembly
Corda	Java Bytecode
Cosmos Tendermint (Ignite)	Go Any language
EOSIO	WebAssembly
Exonum	Java, Rust
Hotmoka	Java Bytecode
Hyperledger Fabric	Go, Java, Javascript
ICP Network	WebAssembly
IOTA	WebAssembly
Lisk	JavaScript
Neo	Neo Execution Format (NEF)
Polkadot	WebAssembly
QANplatform	ELF Linux binaries
Solana	Solana BPF Bytecode
Stellar	WebAssembly
Stratis Platform	Common Intermediate Language (CIL) for .Net
Tezos	Michelson language
XPR Network	WebAssembly

Stratis supports smart contracts development with a subset of the C# language because smart contracts must execute deterministically. Hence, the Stratis smart contracts suite includes a validation tool that checks for any non-deterministic elements in any smart contracts that one write.

Tezos [45] is an open-source blockchain based on a PoS consensus algorithm. It is designed to provide the safety and code correctness required for assets and other high-value components. In addition, it provides an upgrade mechanism based on the will of the community governance. It supports smart contracts written in the Michelson language, a low-level stack-based language, with high-level data types and primitives, and strict static type checking. The blockchain is also supported by smart contract development platforms such as *SmartPy* [46] and *LIGO* [47]. They allow one to write smart contracts using popular GPLs and compile them into Michelson.

XPR Network [48], also known as *Proton*, is a blockchain platform designed to simplify and enhance digital payments and identity verification. The network supports secure identity authentication, enabling users to link their real identities with their accounts while maintaining privacy. Transactions are without fees and XPR Network smart contracts can be written in Typescript/AssemblyScript and compiled to WebAssembly.

IV. TAXONOMY

Section III described several popular blockchains that make use of GPLs for the development of smart contracts and DApps. However, GPLs are involved in different ways, depending on the blockchain. To answer question **RQ1**, it is necessary to consider the language in which the code is written by developers. Indeed, many programming languages can be used at high-level to develop blockchain software.

TABLE 4. Taxonomy of GPLs related to the blockchains.

Blockchain	Framework/Platform	Taxonomy
Algorand	PyTeal	Meta-programming
	Reach	Meta-programming
Casper Network	Casper Contract Library	Meta-programming
Concordium	Concordium STD	Meta-programming
Corda	Contract SDK	Restricted language
Cosmos	Ignite (ex Tendermint Core)	Full language
	Cosmos SDK	Full language
	CosmWasm	Meta-programming
EOSIO	EOSIO Contract Development Toolkit	Meta-programming
Exonum	Exonum Binding	Full language
Hotmoka	Takamaka	Restricted language
Hyperledger Fabric	Hyperledger Fabric SDKs	Full language
IOTA	IOTA	Meta-programming
ICP Network	Canister development kits (CDKs)	Meta-programming
Lisk	Lisk SDK	Full language
Neo	Neo SDK	Meta-programming
Polkadot	Ink!	Meta-programming
QANplatform	QANplatform	Full language
Solana	Solana SDK	Meta-programming
Stellar	SDK for Soroban contracts	Meta-programming
Stratis Platform	.Net Core	Restricted language
Tezos	SmartPy	Meta-programming
	LIGO	Meta-programming
XPR Network	Contract SDK	Meta-programming

However, blockchains typically support only few target languages for executing programs, as depicted in Table 3.

In order to reason about that, it is possible to classify the various GPLs involved in the development of smart contracts and DApps into three macro categories:

- *Full languages*: the code is written in a GPL without restrictions and the blockchain uses the same GPL as target language during the code execution.
- *Restricted languages*: the code is written in a restricted subset of a GPL and the same subset is used by the blockchain during code execution.
- *Meta-programming languages*: the code is written in a language that generates a program in another language. Basically, this happens when a code is written in a GPL but, later, a framework or a compiler translates it into another language used by the blockchain as the target language.

Table 4 shows a classification of GPLs on the blockchain context based on these macro categories. For instance, the GPLs involved in the development of smart contracts and DApps in *Hyperledger Fabric* are classified as full languages, because they allow one to write and execute code by using the same language, without any restriction or modifications. While for *Hotmoka*, the GPL is Java but *Takamaka* allows one to use only a restricted subset of its features and instructions, in order to guarantee some properties to the code execution [49], such as determinism. Next, other blockchains such as *Tezos* allow one to write code in a high-level GPL and run a compiled version of it in another target language supported by the blockchain. There are also cases like *Polkadot* with *Ink!*, that uses a subset of Rust, i.e., a GPL that compiles into another target language, i.e., WebAssembly. In terms of behavior, this is much closer to the meta-programming language than to a restricted language.

V. GPLS IN BLOCKCHAIN SOFTWARE

To answer **RQ2**, it is possible to start by analyzing how widespread is the use of GPLs and DSLs. As shown in Table 1 and also by checking back the history of these indexes, the most popular languages are GPLs rather than DSLs.

It is not surprising that the popularity and widespread diffusion of a language have fundamental implications in the economic context. In general, it means reduced costs and time, because fluent developers in a certain GPL can quickly apply their expertise to a different domain, such as blockchain and smart contracts. For example, the choice of Brown et al. [50] to reuse an industry standard like the JVM for the Corda platform was to make it easier for banks to reuse their existing enterprise code inside smart contracts. However, these considerations are not enough to justify the adoption of GPLs for programming smart contracts and DApps. In fact, it is a weak assumption to argue that it is an advantage if developers write smart contracts in the language they already know. Nowadays, applications are implemented in multi-language solutions and senior developers are proactive in learning new languages. Moreover, according to Deursen et al. [51], DSLs can typically ensure some guarantee (restrictions, expressiveness, abstractions, etc.) on a specific problem domain, which might not be reached by using the full features of a GPL. In blockchain technology, the payoff is minimal in terms of cost, if the language exposes the company to severe economic consequences, such as the immutable deployment of a vulnerable contract that manages cryptocurrencies or financial transactions. Hence, a further question needs to be asked:

> *Considering the fact that there are custom DSLs for smart contracts that provide certain guarantees and properties to the code, how do you justify the fact that many smart contract frameworks make use of GPLs?*

The choice of the language should be based on the project to implement or on the type of functionality that the language makes available to extend, maintain, and evolve the code. According to Deursen et al. [51], the adoption of DSLs leads to benefits and disadvantages. A critical point is surely the difficulty of balancing between DSLs and GPLs constructs and guarantees.

Let us consider Solidity for the Ethereum blockchain, one of the most known DSLs in the blockchain context. The purpose of Solidity was to create a DSL that would be safer and more easily verifiable than a GPL. However, it tends not to deviate too much from traditional GPLs. Indeed, Solidity is a high-level language with syntax and semantics that are very close to programming languages such as Java, C++, and Go. However, Solidity has a low level of abstraction which does not differentiate it much from the latters. For instance, Solidity specialization can be easily replicated in GPLs, such as using Java [11], [27], [52], [53]. The main pitfall is that it requires developers to manage intricate details such as gas costs, storage optimization, and explicit memory management. Moreover, it also does not provide good abstractions in some cases [54]. This makes code development trickly and error-prone, leading to multiple vulnerabilities [55], that have been actually exploited to perform fraudulent actions, such as for the DAO attack [56].

Hence, if DSLs and GPLs are similar, it is easier for developers to choose the latter. In that sense, a widespread GPL is often supported by wider communities, which consequently leads to many studies, research, tools (debuggers, monitors, analyzers, etc.), libraries, software utilities, and IDEs. Nevertheless, as Sect. VI will show, this leads to other challenges.

Another key concept to highlight is that there is nothing similar to a standard DSL for programming smart contracts and DApps. For instance, in the database context, there are several relational database management systems, but most of them use the same SQL as the standard language for database development. Distinct development frameworks might contain variants of SQL (such as MySQL, PostgreSQL, etc.), which implement additional methods, instructions, and macros to facilitate the developer, yet maintain the common features of the language. However, by providing the same common instruction set of SQL, they make it easy to migrate the code to other systems, with minor changes. It is not uncommon that, in industrial realities, the code has to be reused or migrated from one system to another. The same occurs for blockchain-oriented software (BoS), i.e., software working with an implementation of a blockchain [57]. Indeed, the target blockchain could change in favor of another. This can happen for many reasons such as the increase in transaction fees or changes in the visibility of blockchain context (private to public or vice versa). Regarding transaction fees, the cost is typically not fixed and can swing over time. Hence, it may be convenient to change blockchain if the costs are too high. Regarding the visibility context of the blockchain, it drastically impacts the performance of a blockchain network. For instance, public blockchains are more secure than private ones due to their more decentralized and distributed nature, but have disadvantages such as lower transaction speed and higher energy consumption [58]. In the absence of standards, GPLs allow one to facilitate the portability and re-usability of code. Table 4 shows that there are many frameworks targeting different blockchains that are covered by the same GPL. Although each framework certainly has differences, the core logic of smart contracts is typically implemented in the same way by using the same programming language, or at most it requires only small tricks and fixes to be re-used in other frameworks.

The trend that leads to the use of GPLs for the development of smart contracts and DApps is therefore not imputable to a simple answer. Currently, the mix between the still low maturity of high-level DSLs in the blockchain context, the lack of standards, and the scarcity of supporting tools brings the cost-to-benefit ratio towards GPLs.

VI. LIMITATIONS AND CHALLENGES

Turning to **RQ3** and **RQ4**, despite the fact that GPLs are more mature in terms of development and support, the main problem is that they were not initially designed for the development of smart contracts and DApps, exposing developers to several limitations and challenges, that are discussed below.

A. FULL VS RESTRICTED LANGUAGES

The classification between *full* and *restricted* languages arises from two different schools of thought. The first one argues that it is the developer that should be in charge of the quality and safety of the code and, in turn, the implications it may have while using the language at its full expressiveness. Instead, the second one applies restrictions to the full language in such a way that the developer is protected from unexpected behaviors or known vulnerabilities that may arise when using the full language. However, these restrictions could limit the development by denying useful implementations. Full languages are typically involved in controlled environments such as permissioned blockchains, where only authorized users can perform only specific actions granted to them by the blockchain administrators. Instead, restricted languages can be involved also in permissionless blockchains, where peers can develop and deploy code without asking for any permission, since the restricted language forbids a priori specific features of the full language and more likely prevents to compromise the blockchain.

1) NEW ISSUES RELATED TO FULL LANGUAGES

The use of a full language in the frontier of smart contracts and DApps development leads to the risk of new and challenging issues, that usually do not affect DSLs because they have already been taken into consideration in their design phase. For instance, let us consider the problem of non-determinism. Intuitively, a non-deterministic computation involved during the update of the global state of the blockchain leads to a consensus issue. Specifically, the involved transaction fails because the blockchain validators are unable to agree on the same blockchain update. This is the case of code invocations that return the current time of the local machine, since this value may vary between the different blockchain validators, leading to a consensus failure. For instance, both *Ignite* and *Hotmoka* support the GPL Java, where several APIs can be non-deterministic. *Hotmoka* allows the usage just of a restricted subset of Java libraries, blocking the use of features not suitable for smart contracts [49], such as random value functions, disk writes and reads, or internet connections. These restrictions are applied a priori and developers will not be able to lift them. However, as argued in [59] and [60], non-determinism is unsafe in the blockchain context only if it is *global*, that is if it can affect the global state of the blockchain. This means that certain uses of non-determinism are still safe even in the blockchain software developoment. For instance, let us suppose that something must be logged on a blockchain node in your local machine, while the software is running. In the case of *Hotmoka* smart contracts, this will not be possible since the use of local time is not allowed due to the imposed restrictions, while it would be possible in the case of *Cosmos*, where developers are allowed to use the full expressiveness of the Go language. However, in *Ignite* and also in *Cosmos* DApps, developers have to worry about verifying, in a similar case, that the time reported in the log does not end up in the global state of the blockchain. Tools that use formal verification techniques might be able to detect this kind of problem [61].

2) CODE RE-ENGINEERING

Given the flexibility and maturity of GPLs, it is possible to re-engineer the code and standards already implemented using DSLs to obtain technological advantages and develop more efficient code in terms of performance and gas consumption in other blockchains. According to Crosara et al. [52], [53], a trend in blockchain is to apply standards from platform to platform, easing the design challenges with trusted and widely-used specifications. In particular, [52], [53] show how to re-engineer, efficiently, for the Java Virtual Machine, an implementation of the token standards from Solidity to Takamaka (*restricted* language based on Java). However, this approach is typically possible only using *restricted* or *full* languages, because, in *meta-programming*, the high-level optimizations have no effect unless supported by the target language.

3) CODE VERIFICATION

According to Destefanis et al. [62], smart contracts and blockchain software engineering are not yet sufficiently mature, in comparison to traditional software. Smart contracts and DApps rely on a non-standard software life-cycle [63], due to the main peculiarities of the blockchain (immutability, decentralization, and distributability) which make bug fixing and code patching more difficult. The risk of using unsuitable tools is to give a false sense of trust to the developers [59]. At best this will be detected in later stages, still creating delays and increasing the cost of the fix. In the worst case, the deployment in blockchain could occur and the vulnerabilities and inefficiencies could be exploited maliciously without any possibility of fixing them.

Furthermore, the blockchains reported in Table 2 do not boast a vast ecosystem like mainstream blockchains such as Ethereum. Therefore, the effort spent by the communities of practitioners, enterprises, and academics is lower and consequently the tools are often low-patch or absent.

a: VERIFICATION OF FULL LANGUAGES

The verification of smart contracts coming from blockchain classified as full languages means directly analyzing the code written in GPLs. Although GPLs have mature toolbelts, often these are designed for general use and do not include specific features for the blockchain context. In addition, the use of these tools in the blockchain context, without supporting its peculiar features, may lead to critical implications. If a verification tool does not properly model blockchain software, it cannot detect issues specific to the blockchain context, leading to incomplete or incorrect checks.

In Cosmos, the verification tool support is poor, and as stated in the documentation of Tendermint Core in the future they may work with partners to create new tools [64]. GoLiSA [59], [60] proposes an information flow static analysis based on formal methods to detect issues of non-determinism. Instead, Crypto.com [65] provides a query suite for scanning common bug patterns in Cosmos SDK-based applications checking a set of CodeQL code analysis rules. However, according to Surmont et al. [66], this suite requires improvements to reduce the false positive ratio.

In Hyperledger Fabric, the potential risks during the development of smart contract are several [67], [68], ranging from general issues related to GPLs such as non-determinism [60] to more specific ones that are framework-related such as phantom reads [69]. Both linters and more specific tools are available for this platform depending on the problem. For instance, Chaincode Analyzer [70] and Revive^CC [71] provide mainly syntax-based checks for quick code analysis based on an abstract syntax tree investigation. Instead, semantic-based tools such as GoLiSA [60], [69] and HFCCT [72] ensure a more in-depth investigation of the code thanks to abstract interpretation [73] and symbolic execution [74], respectively.

Regarding Exonum, Lisk and QANplatform, to the best of our knowledge, currently, we have found no evidence of specific analyzers or verifiers for these blockchain platforms.

b: VERIFICATION OF RESTRICTED LANGUAGES

In this case, the verification is carried out directly on a subset of the GPL used. Therefore typically the instructions to be covered during the analysis are fewer. Moreover, several critical security issues are defused by the restrictions.

For instance, in Hotmoka, currently, there is no support for standalone analyzers. However, the static analyzer embedded in the compiler of Takamaka smart contracts [27], [49] also performs black-listing to ensure determinism and constraint-based checks to avoid re-entrancy issues. On the other hand, exceptions are triggered at runtime in the case of numerical overflows.

Instead, as far as we know, currently, we have found no analyzers or verifiers that are specific for Corda and Stratis platforms.

B. META-PROGRAMMING LANGUAGES

Meta-programming is widely used in several blockchains, because it typically allows one to develop in different popular high-level languages and then compile them in a single target language, generally at low-level.

1) INFORMATION LOSS

The switch from high to low-level languages can imply a loss of information, making it difficult to understand, reverse engineer, analyze and verify blockchain software. High-level languages, for their nature, tend to approximate semantics through compact instructions, types, annotations, etc. Instead, low-level languages have restricted set of instructions and must make explicit all the operations to perform during the execution, therefore losing expressiveness and increasing the code verbosity. For instance, this is a well-known problem in WebAssembly, because the recovery of high-level function types from WebAssembly binaries is challenging [75].

2) COMPILATION AND SEMANTIC ISSUES

As already reported above, meta-programming involves at least two languages, a source and a target language. If these are different, then compilation problems may occur, as the semantics of certain operations may not be compiled from one language to another. In this scenario, an interesting case of study is SmartPy and the Michelson language. SmartPy is a framework to develop Tezos smart contracts in Python, constantly in development and improvement. While it supports many Python APIs, there is no direct compilation to Michelson from them. Hence, to overcome this problem within the compilation into Michelson, these functions are resolved at compile-time [76] and the results are hardcoded in the Michelson compiled program. Here, it is possible to note two main problems.

The first and more immediate one is that, while analyzing the Michelson source code, the use of Python APIs cannot be retrieved from the Michelson source code, since they are lost during the compilation process described above.

The second one is subtler and involves the stage of resolution described above. If the value of a function call cannot be computed at run time, but is hardcoded at compile time, then the developer should check if its return value is actually constant. Let us explain this by means of the example in Figure 2a. It is a smart contract written in SmartPy, that gets initialized with a numerical parameter `myParameter1` through function `__init__` at line 6. This value can later be changed through function `myEntryPoint` at line 11. The `myEntryPoint` function relies on `random.randint` at line 13, a standard Python API that cannot be compiled into Michelson, since the latter does not support instructions to generate random numerical values. However, the SmartPy framework silently compiles this Python contract without a single warning. Figure 2a shows the result of the compilation; `random.randint` has been evaluated at compile-time and, in this case, the value seven is hard-coded at its place, at line 8 of Figure 2b. Consequently, when running the Michelson code, the latter will not add a random value, as expected by the Python programmer, but rather, always, seven. Moreover, the compiled Michelson program may differ at each compilation, since the value chosen for the random function will vary.

3) TARGET LANGUAGE ISSUES

Another issue concerning meta-programming is related to the fact that the target language might not be a DSL designed for the blockchain context, hence it does not necessarily place guarantees on the execution of the code, which is not checked during the meta-programming phase and can lead to bugs and vulnerabilities. Let us consider the case of *WebAssembly*. As reported in the documentation [77], it is designed to enable high-performance applications on the web. It was later adopted to run on different blockchain platforms for the following key factors: a high-performance execution, a compact representation, and platform independence. Although it has several reasons for being adopted in the blockchain context, the language is still exposed to potential risks, such as non-determinism [78] or numerical overflow [79]. Another pitfall of meta-programming includes the fact that a certain program that is compiled from a GPL to the target language may not preserve the same semantics. For instance, suppose to use the Python3 language for meta-programming and WebAssembly or Ethereum bytecode as the target language. Python3 allows one to represent potentially unbound integers [80]. Instead, WebAssembly and Ethereum bytecode only support bounded integers [81], [82]. This means that the compilation of some instructions related to integer values could fail, even silently. Therefore, in the worst case, the compilation could be successful but the execution of arithmetic operations could be subject to vulnerabilities. For instance, this was the case of the EOSIO blockchain that was affected by such an integer overflow vulnerability [79].

4) CODE VERIFICATION

As for blockchains that support meta-programming, the software verification toolbelt typically includes verifiers that directly analyze the target language of the blockchain. The reasons for this choice are multiple [83]: (i) a more faithful analysis because it is the analyzed code that actually will be executed, (ii) it enables the analysis when the source code is not available, (iii) it avoids redundant work already performed by the compiler such as name resolution and type checking, (iv) the semantics of source code is expanded by the compiler in the target code. However, the program representation could be challenging, especially for low-level target languages [83].

In Algorand, MATH [84] performs static analyses targeting TEAL language to detect math exploits and byte subtraction vulnerabilities. Tealer [85] builds control-flow graphs from the TEAL code applying a sort of pattern-matching search based on regular expression to detect vulnerabilities such as missing validation issues and unprotected contacts. Instead, Panda [86] proposes a static analysis based on the symbolic execution of TEAL contracts.

In EOSIO, EOSFuzzer [87] applies black-boxing verification to detect smart contract vulnerabilities through fuzzing techniques. While EOSAFE [88] and WANA [89] involve

```python
import smartpy as sp

import random
# A class of contracts
class MyContract(sp.Contract):
    def __init__(self, myParameter1):
        self.init(myParameter1=myParameter1)

    # An entry point, i.e., a message receiver
    # (contracts react to messages)
    @sp.entry_point
    def myEntryPoint(self):
        self.data.myParameter1 += random.randint(0,10)
```

(a) Python code

```
parameter (unit %myEntryPoint);
storage int;
code
  {
    CDR;        # @storage
    # == myEntryPoint ==
    # self.data.myParameter1 += 7 # @storage
    PUSH int 7; # int : @storage
    ADD;        # int
    NIL operation; # list operation : int
    PAIR;       # pair (list operation) int
  };
```

(b) Michelson code

FIGURE 2. An issue related to meta-programming from SmartPy.

symbolic execution on Web Assembly to detect issues such as fake tokens and receipts.

In Solana, FuzzDelSol [90] proposes fuzzing verification. VRust [91] analyzes Rust's mid-level intermediate representation of smart contract code for validating untrustful input accounts. Instead, Solana Certora Prover [92] verifies conditions (i.e., logical formulas) with a solver based on satisfiability modulo theories from an LLVM-based intermediate representation of contracts written in Rust.

In Tezos, MichelsonLiSA [83], [93] and MOPSA [94] provide semantic checks based on abstractions targetting Michelson language for the static detection of smart contract issues. Mi-Cho-Coq [95] ensures the functional correctness of contracts through the Coq proof assistant. While, Helmholtz [96] proposes a refinement-type approach for the verification of Michelson programs, that discards verification conditions with an SMT solver.

Finally, to the best of our knowledge, we have found currently no evidence of specific analyzers or verifiers for Casper Network, Concordium, ICP-Network, IOTA, Neo, Stellar, and XPR Network.

VII. PERFORMANCE EVALUATION

According to Kruglik et al. [97] low performance and lack of scalability are important problems of modern blockchain systems. In smart contracts, they are crucial because have a direct impact on transaction costs, execution speed, and the overall efficiency of the blockchain, ensuring that DApps run smoothly and cost-effectively.

Choosing one programming language rather than another can have different implications at performance and scalability levels. This is especially true for blockchains that allow one to support multiple programming languages for smart contract development. Indeed, there may be differences for example in terms of throughput, latency, and memory depending on the language executed. Consider for instance Hyperledger Fabric that allows the execution of smart contracts written in Go, Javascript, and Java. According to Foschini et al. [98], the Go language is the most performing one, while Java is the worst with, in some cases, a performance degradation of almost double compared to other languages. The probable reasons are that Go is also the language used to implement the Hyperledger Fabric so it is probably optimized, while Java requires launching a virtualized environment (i.e. Java Virtual Machine) which typically takes more time and resources.

Regarding the meta-programming, the choice of programming language has less impact. In fact, the performance of the blockchain does not directly depend on the GPL chosen but on the target language which in most cases is always the same. Then, what really impacts performance is the compiler that can provide code optimizations in different ways. For instance, for Web Assembly, there are many compilers that support GPLs but some optimize more than others ensuring the better performances but there is always to consider the trade-off between optimizations, support, communities, and documentation [99].

However, currently, the existing literature does not provide empirical evaluations for most of the identified blockchains, so in-depth investigations regarding the intersection between the performances of languages and blockchains were not possible due to the lack of data.

VIII. RELATED WORK

This work classified and discussed GPLs for smart contracts and DApps. To the best of our knowledge, there are very few academic studies that classify blockchain programming languages, especially in relation to GPLs.

In particular, Varela-Vaca et al. [100] propose a mapping study and a general snapshot of the languages for smart contracts, emphasizing the importance of grey literature (i.e., white papers, reports, documentation, working papers, etc.). While, Fraga-Lamas et al. [101] investigate the blockchain impact in the Industry 5.0 also proposing a mapping of plaforms supporting smart contracts written in several languages including GPLs.

A. SURVEYS ON DSLS

Some research papers, reported below, investigate and survey DSLs, without explicitly considering GPLs. Alam et al. [102] survey the state-of-art of DSLs introduced in literature since 2015, also providing a comparative analysis based on their application target, development stage, and offered features.

TABLE 5. Summary of the main research questions and answers discussed in this paper.

Question	Answer
RQ1: How are GPLs involved in the development of smart contracts and DApps?	**A1**: GPLs are involved in different ways depending on the blockchain. They can be categorized in three macro categories: full languages (GPL without any restriction), restricted languages (subset of GPL), and meta-programming (development using GPL, compilation in another target language).
RQ2: Why are GPLs used for programming smart contracts and DApps? What reasons lead to their choice?	**A2**: The main factors are the popularity of GPLs, the reduced learning curve, code reuse, their wide toolbelts (debug and monitoring tools, libraries, etc.) and the lack of popular DSLs with a greater level of abstraction.
RQ3: What challenges arise from writing smart contracts using GPLs?	**A3**: The adoption of GPLs has introduced several challenges. For example: efficient and secure code development, software reengineering, and the development of new techniques that support analysis, verification, and reverse engineering of smart contracts and DApps.
RQ4: What are the current limitations of developing smart contracts by using GPLs?	**A4**: GPLs were not initially designed for blockchain software. This leads to several issues such as the emergence of new vulnerabilities, the loss of information related to compilation, limited features, etc. Furthermore, the tools that support GPLs are mature in terms of development, but typically cover only general issues, without supporting specific features for the blockchain context.

Parizi et al. [103] provide a survey about programming languages used for smart contract development, focusing on three DSLs, namely Solidity, Pact, and Liquidity, focusing on their usability and security aspects. Seijas et al. [104] overview scripting languages used in existing cryptocurrencies, focusing on those for Bitcoin, Nxt, and Ethereum. Zou et al. [105] analyze the current state and potential challenges developers are facing in developing smart contracts on blockchains, highlighting the limitations of DSLs such as Solidity. Bartoletti et al. [54] describe the programming languages related to smart contract development of the major permissionless blockchain commenting on the Python dialects used for Tezos and Algorand.

B. SURVEYS ON BLOCKCHAIN SECURITY

The research community has proposed several studies and surveys about the security of blockchain, even though they do not explicitly investigate programming languages used to build blockchain software. These are reported below. In [106], the authors discuss different technologies embedded in blockchains, such as consensus algorithms, public key cryptography, and hash functions used in blockchain, with a focus on their security aspects, providing a survey about the types of attacks that affected blockchains, the state-of-art analysis tools that have been proposed over the years to cope with those attacks, and a qualitative comparison between these tools, based on the number of vulnerabilities detected. Similarly, [107] surveys blockchain technology until the end of 2019, also presenting the most popular blockchain applications. Also [108] is a survey about security aspects of blockchain, such as availability and integrity, but it focuses on the use of blockchain within information systems. As far as information systems are concerned, it reviews blockchain security in this context at three different stages, at the process, data, and infrastructure levels, drawing future research directions on blockchain security, with a particular focus on business and industrial-related issues.

IX. CONCLUSION

To the best of our knowledge, this is the first survey about GPLs in blockchain software. In particular, it investigates which GPLs are officially involved in smart contract and DApps development, for the most popular blockchains, and discusses why they have been adopted by the blockchain community. Therefore, it provides a taxonomy capturing how GPLs are involved in blockchain software development, namely as full languages, restricted languages, or meta-programming languages. Based on this taxonomy, it presents challenges and current limitations of adopting GPLs for blockchain software development. Table 5 summarizes the research questions set at the beginning of this paper and the corresponding answers that it provides.

Future research directions may investigate other aspects of blockchain related to GPLs such as potential security risks, cross-blockchain communications and the support to external information sources. They may also focused on the collection and analysis of empirical data to quantitatively understand which GPLs are most used and widespread by blockchain communities, what are the best performing, etc. Furthermore, other blockchain-like technologies have emerged in recent years such as ledger databases [109] that combine some blockchain features with more traditional database techniques. While they do not currently include smart contract execution, they do take advantage from common database languages, such as SQL dialects, and not strictly specific DSLs to perform data operations.

REFERENCES

[1] S. Nakamoto. (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System*. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] V. Buterin, "A next-generation smart contract and decentralized application platform," White Paper, 2014. [Online]. Available: https://ethereum.org/en/whitepaper/

[3] DefiLlama. *Defillama and Our Methodology*. Accessed: Jun. 2024. [Online]. Available: https://docs.llama.fi/#our-methodology

[4] *Breakdown by Smart Contract Languages*. Accessed: Jun. 2024. [Online]. Available: https://defillama.com/languages

[5] TIOBE. *TIOBE Index*. Accessed: Jun. 2024. [Online]. Available: https://www.tiobe.com/tiobe-index/

[6] Pierre Carbonnelle. *PYPL Index*. Accessed: Jun. 2024. [Online]. Available: https://pypl.github.io/PYPL.html

[7] Cardano. *Cardano WebPage*. Accessed: Aug. 2024. [Online]. Available: https://cardano.org/

[8] A. M. Antonopoulos, *Mastering Bitcoin: Programming the Open Blockchain*, 2nd ed. Sebastopol, CA, USA: O'Reilly, 2017.

[9] Cardano. *Cardano Documentation—Programming Languages*. Accessed: Aug. 2024. [Online]. Available: https://developers.cardano.org/docs/smart-contracts/#programming-languages

[10] Ethereum Foundation. *Ethereum Documentation—Programming Languages*. Accessed: Jul. 2022. [Online]. Available: https://ethereum.org/en/developers/docs/programming-languages/

[11] Jthereum. *Jthereum—Documentation*. Accessed: Jun. 2024. [Online]. Available: https://jthereum.com/documentation

[12] J. Chen and S. Micali, "Algorand: A secure and efficient distributed ledger," *Theor. Comput. Sci.*, vol. 777, pp. 155–183, Jul. 2019, doi: 10.1016/j.tcs.2019.02.001.

[13] Algorand. *PyTeal Documentation*. Accessed: Aug. 2024. [Online]. Available: https://pyteal.readthedocs.io/en/stable/

[14] Reach.sh. *Reach Documentation*. Accessed: Jul. 2022. [Online]. Available: https://docs.reach.sh/

[15] Casper Network. *White Paper*. Accessed: Jun. 2024. [Online]. Available: https://backend.casper.network/assets/0f7730f2-c480-43be-b332-242567dd95bc?download

[16] Concordium. *Technology*. Accessed: Jun. 2024. [Online]. Available: https://www.concordium.com/technology

[17] J. Kwon and E. Buchman, "Cosmos whitepaper," Tech. Rep., 2019.

[18] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, Univ. Guelph, Guelph, ON, Canada, 2016.

[19] Ignite. *Ignite Documentation*. Accessed: Aug. 2024. [Online]. Available: https://docs.ignite.com/

[20] E. Buchman, "Byzantine fault tolerant state machine replication in any programming language," in *Proc. ACM Symp. Princ. Distrib. Comput.*, New York, NY, USA, 2019, p. 546, doi: 10.1145/3293611.3338023.

[21] CosmWasm. *CosmWasm Documentation*. Accessed: Jul. 2022. [Online]. Available: https://docs.cosmwasm.com/docs/1.0/

[22] WebAssembly. *Webassembly Documentation—Developers Guide*. Accessed: Aug. 2024. [Online]. Available: https://webassembly.org/getting-started/developers-guide/

[23] EOS.IO. *EOS.IO White Paper*. Accessed: Aug. 2024. [Online]. Available: https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md

[24] Bitfury Group Limited. *Exonum: Byzantine Fault Tolerant Protocol for Blockchains*. Accessed: Aug. 2024. [Online]. Available: https://bitfury.com/content/downloads/wp_consensus_181227.pdf

[25] *Exonum Services*. Accessed: Aug. 2024. [Online]. Available: https://exonum.com/doc/version/latest/architecture/services/

[26] Fausto Spoto. *Hotmoka Github Repository*. Accessed: Aug. 2024. [Online]. Available: https://github.com/Hotmoka/hotmoka/blob/master/README.md

[27] F. Spoto, "A Java framework for smart contracts," in *Proc. 3rd Wokshop Trusted Smart Contracts (WTSC)* (Lecture Notes in Computer Science), vol. 11599, Saint Kitts, Nevis. Cham, Switzerland: Springer, Feb. 2019, pp. 122–137, doi: 10.1007/978-3-030-43725-1_10.

[28] Hyperledger. *Hyperledger Fabric Documentation*. Accessed: Aug. 2024. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.4/

[29] *Hyperledger Fabric Documentation*. Accessed: Aug. 2024. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/blockchain.html#what-is-hyperledger-fabric

[30] S. Popov and Q. Lu, "IOTA: Feeless and free," *IEEE Block chain Tech. Briefs*, Jan. 2019. [Online]. Available: https://blockchain.ieee.org/technicalbriefs/january-2019/iota-feeless-and-free

[31] Internet Computer. *White Paper*. Accessed: Jun. 2024. [Online]. Available: https://internetcomputer.org/whitepaper.pdf

[32] Liks. *Liks GitHub Repository*. Accessed: Aug. 2024. [Online]. Available: https://github.com/LiskHQ/lisk-core#lisk-core

[33] Neo. *Neo White Paper*. Accessed: Jul. 2022. [Online]. Available: https://docs.neo.org/v2/docs/en-us/basic/whitepaper.html

[34] Neo Team. *NEO Documentation—Smart Contracts*. Accessed: Jul. 2022. [Online]. Available: https://neo.org/technology#smart-contracts

[35] Polkadot. *Polkadot White Paper*. Accessed: Jul. 2022. [Online]. Available: https://polkadot.network/PolkaDotPaper.pdf

[36] Parity Technologies. *Ink! Documentation*. Accessed: Aug. 2024. [Online]. Available: https://github.com/use-ink/ink-docs

[37] QAN Platform. *QAN Platform—White Paper*. Accessed: Jun. 2024. [Online]. Available: https://learn.qanplatform.com/papers/white-paper

[38] *[QVM] Multi-Language Smart Contracts*. Accessed: Jun. 2024. [Online]. Available: https://learn.qanplatform.com/developers/qvm-multi-language-smart-contracts

[39] Anatoly Yakovenko. *Solana: A New Architecture for a High Performance Blockchain V0.8.13*. Accessed: Aug. 2024. [Online]. Available: https://solana.com/solana-whitepaper.pdf

[40] Solana. *Solana Getting Started With Solana Development*. Accessed: Aug. 2024. [Online]. Available: https://solana.com/news/getting-started-with-solana-development

[41] Stellar Development Foundation. *Stellar Documentation—Intro to Stellar*. Accessed: Aug. 2024. [Online]. Available: https://www.stellar.org/learn/intro-to-stellar

[42] *Stellar Official Blog—Smart Contracts on Stellar Why Now?* Accessed: Aug. 2024. [Online]. Available: https://stellar.org/blog/smart-contracts-on-stellar-why-now?locale=en

[43] *Stellar Official Blog—Project Jump Cannon Choosing Wasm?* Accessed: Aug. 2024. [Online]. Available: https://stellar.org/blog/project-jump-cannon-choosing-wasm?locale=en

[44] STRATIS Platform. *White Paper*. Accessed: Jun. 2024. [Online]. Available: https://www.stratisplatform.com/files/Stratis_Whitepaper.pdf

[45] V. Allombert, M. Bourgoin, and J. Tesson, "Introduction to the tezos blockchain," in *Proc. Int. Conf. High Perform. Comput. Simulation (HPCS)*, Jul. 2019, pp. 1–10, doi: 10.1109/HPCS48598.2019.9188227.

[46] SmartPy. *SmartPy Documentation*. Accessed: Aug. 2024. [Online]. Available: https://smartpy.io/docs/

[47] LIGO. *LIGO Documentation*. Accessed: Aug. 2024. [Online]. Available: https://ligolang.org/docs/intro/introduction

[48] XPR Network. *Introducing XPR Network*. Accessed: Jun. 2024. [Online]. Available: https://xprnetwork.org/blog/introducing-xpr-network

[49] F. Spoto, "Enforcing determinism of Java smart contracts," in *Proc. 4th Wokshop Trusted Smart Contracts (WTSC)* (Lecture Notes in Computer Science), vol. 12063. Kota Kinabalu, Malaysia: Springer, Feb. 2020, pp. 568–583, doi: 10.1007/978-3-030-54455-3_40.

[50] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn, "Corda: An introduction—White paper," *R3 CEV*, vol. 1, no. 15, p. 14, Aug. 2016. https://docs.r3.com/en/pdf/corda-introductory-whitepaper.pdf

[51] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000, doi: 10.1145/352029.352035.

[52] M. Crosara, L. Olivieri, F. Spoto, and F. Tagliaferro, "Re-engineering ERC-20 smart contracts with efficient snapshots for the Java virtual machine," in *Proc. 3rd Int. Conf. Blockchain Comput. Appl. (BCCA)*, Nov. 2021, pp. 187–194, doi: 10.1109/BCCA53669.2021.9657047.

[53] M. Crosara, L. Olivieri, F. Spoto, and F. Tagliaferro, "Fungible and non-fungible tokens with snapshots in Java," *Cluster Comput.*, vol. 26, no. 5, pp. 2701–2718, Oct. 2023, doi: 10.1007/s10586-022-03756-3.

[54] M. Bartoletti, L. Benetollo, M. Bugliesi, S. Crafa, G. D. Sasso, R. Pettinau, A. Pinna, M. Piras, S. Rossi, S. Salis, A. Spanò, V. Tkachenko, R. Tonelli, and R. Zunino, "Smart contract languages: A comparative analysis," *Future Gener. Comput. Syst.*, vol. 164, Mar. 2025, Art. no. 107563, doi: 10.1016/j.future.2024.107563.

[55] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *Principles of Security and Trust*, M. Maffei and M. Ryan, Eds. Berlin, Germany: Springer, 2017, pp. 164–186, doi: 10.1007/978-3-662-54455-6_8.

[56] N. Popper, "A hacking of more than $50 million dashes hopes in the world of virtual currency," *The New York Times*, vol. 17, Jun. 2016.

[57] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, "Blockchain-oriented software engineering: Challenges and new directions," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2017, pp. 169–171, doi: 10.1109/ICSE-C.2017.142.

[58] Y. Bakos and H. Halaburda, ''Tradeoffs in permissioned vs permissionless blockchains: Trust and performance,'' *SSRN*, Feb. 2021. [Online]. Available: https://papers.ssrn.com/abstract=3789425

[59] L. Olivieri, F. Tagliaferro, V. Arceri, M. Ruaro, L. Negrini, A. Cortesi, P. Ferrara, F. Spoto, and E. Talin, ''Ensuring determinism in blockchain software with GoLiSA: An industrial experience report,'' in *11t ACM SIGPLAN Int. Workshop State Art Program Anal. (SOAP)*, San Diego, CA, USA, L. Gonnord and L. Titolo, Eds. New York, NY, USA: ACM Press, 2022, pp. 23–29, doi: 10.1145/3520313.3534658.

[60] L. Olivieri, L. Negrini, V. Arceri, F. Tagliaferro, P. Ferrara, A. Cortesi, and F. Spoto, ''Information flow analysis for detecting non-determinism in blockchain,'' in *Proc. 37th Eur. Conf. Object-Oriented Program. (ECOOP)* (Leibniz International Proceedings in Informatics), vol. 263, K. Ali and G. Salvaneschi, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz Center for Informatics, 2023, p. 23, doi: 10.4230/LIPIcs.ECOOP.2023.23.

[61] L. Olivieri and F. Spoto, ''Software verification challenges in the blockchain ecosystem,'' *Int. J. Softw. Tools Technol. Transf.*, vol. 26, pp. 431–444, Jul. 2024, doi: 10.1007/s10009-024-00758-x.

[62] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons, ''Smart contracts vulnerabilities: A call for blockchain software engineering?'' in *Proc. Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Mar. 2018, pp. 19–25, doi: 10.1109/IWBOSE.2018.8327567.

[63] M. A. Mahdi H. Miraz, ''Blockchain enabled smart contract based applications: Deficiencies with the software development life cycle models,'' *Baltica J.*, vol. 33, pp. 101–116, Apr. 2020.

[64] T. Inc. (2022). *What is Tendermint: A Note on Determinism*. Accessed: 2024. [Online]. Available: https://github.com/tendermint/tendermint/blob/7983f9cc36c31e140e46ae5cb00ed39f637ef283/docs/introduction/what-is-tendermint.md#a-note-on-determinism

[65] Crypto.com. *Cosmos SDK CodeQL*. Accessed: Aug. 2024. [Online]. Available: https://github.com/crypto-com/cosmos-sdk-codeql

[66] J. Surmont, W. Wang, and T. V. Cutsem, ''Static application security testing of consensus-critical code in the cosmos network,'' in *Proc. 5th Conf. Blockchain Res. Appl. Innov. Netw. Services (BRAINS)*, Oct. 2023, pp. 1–8.

[67] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun, ''Potential risks of hyperledger fabric smart contracts,'' in *Proc. IEEE Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Feb. 2019, pp. 1–10.

[68] P. Lv, Y. Wang, Y. Wang, and Q. Zhou, ''Potential risk detection system of hyperledger fabric smart contract based on static analysis,'' in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Athens, Greece, Sep. 2021, pp. 1–7.

[69] L. Olivieri, L. Negrini, V. Arceri, B. Chachar, P. Ferrara, and A. Cortesi, ''Detection of phantom reads in hyperledger fabric,'' *IEEE Access*, vol. 12, pp. 80687–80697, 2024, doi: 10.1109/ACCESS.2024.3410019.

[70] K. Yamashita and J. Ry. (2020). *Chaincode Analyzer*. Accessed: Feb. 2024. [Online]. Available: https://github.com/hyperledger-labs/chaincode-analyzer

[71] C. Siva. (2021). *Revivecc*. Accessed: Feb. 2024. [Online]. Available: https://github.com/sivachokkapu/revive-cc

[72] P. Li, S. Li, M. Ding, J. Yu, H. Zhang, X. Zhou, and J. Li, ''A vulnerability detection framework for hyperledger fabric smart contracts based on dynamic and static analysis,'' in *Proc. 26th Int. Conf. Eval. Assessment Softw. Eng.*, New York, NY, USA, 2022, pp. 366–374, doi: 10.1145/3530019.3531342.

[73] R. Wilhelm, ''Principles of abstract interpretation: By patrick cousot MIT Press, 2021, ISBN 9780262044905, pp. 1–819. Reviewed by Reinhard Wilhelm,'' *Formal Aspects Comput.*, vol. 34, no. 2, pp. 1–3, Sep. 2022, doi: 10.1145/3546953.

[74] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, ''A survey of symbolic execution techniques,'' *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–39, May 2018, doi: 10.1145/3182657.

[75] D. Lehmann and M. Pradel, ''Finding the dwarf: Recovering precise types from WebAssembly binaries,'' in *Proc. 43rd ACM SIGPLAN Int. Conf. Program. Lang. Design Implement.*. New York, NY, USA, Jun. 2022, pp. 410–425, doi: 10.1145/3519939.3523449.

[76] SmartPy. *SmartPy Reference - Constants vs Expressions*. Accessed: Aug. 2024. [Online]. Available: https://smartpy.io/reference.html

[77] WebAssembly. *Webassembly Introduction*. Accessed: Aug. 2024. [Online]. Available: https://webassembly.github.io/spec/core/intro/introduction.html

[78] *WebAssembly GitHub—Non determinism*. Accessed: Aug. 2024. [Online]. Available: https://github.com/WebAssembly/design/blob/main/Nondeterminism.md

[79] T. Min and W. Cai, ''A security case study for blockchain games,'' in *Proc. IEEE Games, Entertainment, Media Conf. (GEM)*, Jun. 2019, pp. 1–8, doi: 10.1109/GEM.2019.8811555.

[80] Python Software Foundation. *Python3 Documentation—Objects*. Accessed: Aug. 2024. [Online]. Available: https://docs.python.org/3/reference/datamodel.html#objects-values-and-types

[81] WebAssembly. *WebAssembly GitHub—Number Types*. Accessed: Aug. 2024. [Online]. Available: https://webassembly.github.io/spec/core/syntax/types.html#number-types

[82] Ethereum Foundation. *EVM Documentation*. Accessed: Aug. 2024. [Online]. Available: https://ethereum.org/en/developers/docs/evm/opcodes

[83] L. Olivieri, L. Negrini, V. Arceri, T. Jensen, and F. Spoto, ''Design and implementation of static analyses for tezos smart contracts,'' *Distrib. Ledger Technol.*, Jan. 2024, doi: 10.1145/3643567.

[84] P. Ince, X. Luo, J. Yu, J. K. Liu, and X. Du, ''MATH—Finding and fixing exploits in algorand,'' in *Proc. IEEE 22nd Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Nov. 2023, pp. 1572–1579.

[85] F. Josselin (2020). *Tealer*. Accessed: Aug. 2024. [Online]. Available: https://github.com/crytic/tealer

[86] Z. Sun, X. Luo, and Y. Zhang, ''Panda: Security analysis of algorand smart contracts,'' in *Proc. 32nd USENIX Secur. Symp.*, New York, NY, USA, 2023, pp. 1–19.

[87] Y. Huang, B. Jiang, and W. K. Chan, ''EOSFuzzer: Fuzzing EOSIO smart contracts for vulnerability detection,'' in *Proc. 12th Asia–Pacific Symposiuma Internetware*, New York, NY, USA, 2021, pp. 99–109, doi: 10.1145/3457913.3457920.

[88] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, ''EOSAFE: Security analysis of EOSIO smart contracts,'' in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.)*, Aug. 2021, pp. 1271–1288. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/he-ningyu

[89] B. Jiang, Y. Chen, D. Wang, I. Ashraf, and W. K. Chan, ''WANA: Symbolic execution of wasm bytecode for extensible smart contract vulnerability detection,'' in *Proc. IEEE 21st Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Dec. 2021, pp. 926–937.

[90] S. Smolka, J.-R. Giesen, P. Winkler, O. Draissi, L. Davi, G. Karame, and K. Pohl, ''Fuzz on the beach: Fuzzing solana smart contracts,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, 2023, pp. 1197–1211, doi: 10.1145/3576915.3623178.

[91] S. Cui, G. Zhao, Y. Gao, T. Tavu, and J. Huang, ''VRust: Automated vulnerability detection for solana smart contracts,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, 2022, pp. 639–652, doi: 10.1145/3548606.3560552.

[92] J. Navas and A. Gurfinkel, ''Verification of solana programs,'' in *Proc. Solana Certora Prover 'Challenges Softw. Verification Symp.' (CSV)*, Venice, Italy, May 2023, pp. 1–28. Accessed: Aug. 2024. [Online]. Available: https://jorgenavas.github.io/slides/Solana-slides-CSV-05-26-23.pdf

[93] L. Olivieri, T. Jensen, L. Negrini, and F. Spoto, ''MichelsonLiSA: A static analyzer for tezos,'' in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops Affiliated Events (PerCom Workshops)*, Mar. 2023, pp. 80–85, doi: 10.1109/PerComWorkshops56833.2023.10150247.

[94] G. Bau, A. Miné, V. Botbol, and M. Bouaziz, ''Abstract interpretation of Michelson smart-contracts,'' in *Proc. 11th ACM SIGPLAN Int. Workshop State Art Program Anal.*, New York, NY, USA, 2022, pp. 36–43, doi: 10.1145/3520313.3534660.

[95] B. Bernardo, R. Cauderlier, G. Claret, A. Jakobsson, B. Pesin, and J. Tesson, ''Making Tezos smart contracts more reliable with Coq,'' in *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, T. Margaria and B. Steffen, Eds. Cham, Switzerland: Springer, 2020, pp. 60–72.

[96] Y. Nishida, H. Saito, R. Chen, A. Kawata, J. Furuse, K. Suenaga, and A. Igarashi, ''Helmholtz: A verifier for tezos smart contracts based on refinement types,'' *New Gener. Comput.*, vol. 40, no. 2, pp. 507–540, Jul. 2022.

[97] S. Kruglik, K. Nazirkhanova, and Y. Yanovich, ‘‘Challenges beyond blockchain: Scaling, oracles and privacy preserving,’’ in *Proc. XVI Int. Symp. ‘Problems Redundancy Inf. Control Syst.’ (REDUNDANCY)*, Oct. 2019, pp. 155–158.

[98] L. Foschini, A. Gavagna, G. Martuscelli, and R. Montanari, ‘‘Hyperledger fabric blockchain: Chaincode performance analysis,’’ in *Proc. IEEE Int. Conf. Commun. (ICC)*, Dublin, Ireland, Jun. 2020, pp. 1–6, doi: 10.1109/ICC40277.2020.9149080.

[99] P. P. Ray, ‘‘An overview of WebAssembly for IoT: Background, tools, State-of-the-art, challenges, and future directions,’’ *Future Internet*, vol. 15, no. 8, p. 275, Aug. 2023.

[100] Á. J. Varela-Vaca and A. M. R. Quintero, ‘‘Smart contract languages: A multivocal mapping study,’’ *ACM Comput. Surv.*, vol. 54, no. 1, pp. 1–38, 2021, doi: 10.1145/3423166.

[101] P. Fraga-Lamas, T. M. Fernández-Caramés, A. M. Rosado da Cruz, and S. I. Lopes, ‘‘An overview of blockchain for industry 5.0: Towards human-centric, sustainable and resilient applications,’’ *IEEE Access*, vol. 12, pp. 116162–116201, 2024.

[102] M. T. Alam, S. Chowdhury, R. Halder, and A. Maiti, ‘‘Blockchain domain-specific languages: Survey, classification, and comparison,’’ in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, December Y. Xiang, Z. Wang, H. Wang, and V. Niemi, Eds. Melbourne, VIC, Australia: IEEE Press, Dec. 2021, pp. 499–504, doi: 10.1109/Blockchain53845.2021.00076.

[103] R. M. Parizi, Amritraj, and A. Dehghantanha, ‘‘Smart contract programming languages on blockchains: An empirical evaluation of usability and security,’’ in *Blockchain–(ICBC)*, S. Chen, H. Wang, and L.-J. Zhang, Eds. Cham, Switzerland: Springer, 2018, pp. 75–91, doi: 10.1007/978-3-319-94478-4_6.

[104] P. L. Seijas, S. Thompson, and D. McAdams, ‘‘Scripting smart contracts for distributed ledger technology,’’ in *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, vol. 10323, M. Brenner et al., Eds. Springer, 2017, pp. 361–362, doi: 10.1007/978-3-319-70278-0.

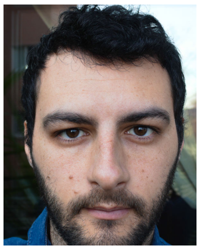
[105] W. Zou, D. Lo, P. S. Kochhar, X. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, ‘‘Smart contract development: Challenges and opportunities,’’ *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2084–2106, Oct. 2021, doi: 10.1109/TSE.2019.2942301. https://doi.org/10.1109/TSE.2019.2942301

[106] H. Guo and X. Yu, ‘‘A survey on blockchain technology and its security,’’ *Blockchain: Res. Appl.*, vol. 3, no. 2, Jun. 2022, Art. no. 100067, doi: 10.1016/j.bcra.2022.100067.

[107] H. T. M. Gamage, H. D. Weerasinghe, and N. G. J. Dias, ‘‘A survey on blockchain technology concepts, applications, and issues,’’ *Social Netw. Comput. Sci.*, vol. 1, no. 2, p. 114, Mar. 2020, doi: 10.1007/s42979-020-00123-0.

[108] D. Berdik, S. Otoum, N. Schmidt, D. Porter, and Y. Jararweh, ‘‘A survey on blockchain for information systems management and security,’’ *Inf. Process. Manage.*, vol. 58, no. 1, Jan. 2021, Art. no. 102397, doi: 10.1016/j.ipm.2020.102397.

[109] D. L. Fekete and A. Kiss, ‘‘A survey of ledger technology-based databases,’’ *Future Internet*, vol. 13, no. 8, p. 197, Jul. 2021, doi: 10.3390/fi13080197.

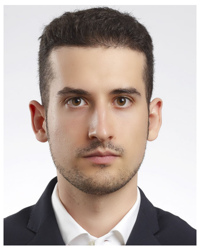


LUCA OLIVIERI received the Ph.D. degree in computer science from the University of Verona, focusing on verifying smart contracts and blockchain software. He was a Software Engineer and a Research Scientist for five years in the industrial field on static analysis based on abstract interpretation, mainly for Java and C# programs. After that, he joined the Software and System Verification (SSV) Group, Ca’ Foscari University of Venice, where he is currently an Assistant Professor (non-tenure track) of computer science.

VINCENZO ARCERI received the Ph.D. degree in computer science from the University of Verona, in May 2020. From September 2019 to September 2021, he was a Postdoctoral Researcher with the Software and System Verification Research Group, Ca’ Foscari University of Venice. He was a UROP Student with Imperial College London, in Summer 2016, under the supervision of Prof. Sergio Maffeis. He is currently an Assistant Professor (non-tenure track) with the Department of Mathematical, Physical, and Computer Sciences, University of Parma. His research interests include static program analysis, abstract interpretation, string analysis and verification, blockchain software verification, and, more generally, and formal methods for program security.

BADARUDDIN CHACHAR received the bachelor’s degree in computer science from COMSATS University, Lahore, Pakistan, in 2013, on a Fully Funded National Scholarship for Science and Engineering by the Ministry of IT, Pakistan, and the master’s degree in software engineering from Sukkur IBA University, Pakistan, in 2018. He is currently pursuing the Ph.D. degree in computer science with the Ca’ Foscari University of Venice, Venice, Italy. His Ph.D. project focuses on smart contract vulnerability detection using static analysis under the supervision of Agostino Cortesi and Pietro Ferrara.

LUCA NEGRINI received the bachelor’s and master’s degrees from the University of Verona, and the Ph.D. degree in computer science from the Ca’ Foscari University of Venice, in January 2023, focusing on multi-language static analysis. He has by five years of industrial experience in the development and applications of static analysis and abstract interpretation with the Julia static analyzer. He then joined the Ca’ Foscari University of Venice, as a Research Fellow, where he is currently an Assistant Professor (non-tenure track).



FABIO TAGLIAFERRO received the degree in computer science and engineering from the University of Verona, in 2020. Then, he conducted research activities at the University of Verona, for a few years after his graduation. He is currently a Software Engineer with Equixly Srl, Italy.

FAUSTO SPOTO is currently an Associate Professor with the University of Verona, Italy, where he studies programming languages and software engineering. He has developed techniques for the static analysis of Java and Java bytecode. More recently, he studied the use of Java for writing smart contracts and developed the Hotmoka blockchain. His current research interest includes static analysis and construction of verified smart contracts.

PIETRO FERRARA is an Expert of abstract interpretation-based static analysis, focusing on identifying security vulnerabilities and privacy breaches in object-oriented programs. He is currently an Associate Professor with the Ca' Foscari University of Venice. Prior to academia, he gained industry experience bridging scientific research with software development and delivery. This included roles, such as the Head of the Research and Development, JuliaSoft Srl, and a Research Staff Member with IBM Research.

AGOSTINO CORTESI received the Ph.D. degree from the University of Padova, Padua, Italy. He is currently a Full Professor of computer science with the Ca' Foscari University of Venice. His research interests include the software engineering and software verification areas, combining theoretical, and applicative approaches. He contributed to more than 200 articles in journals (including *ACM Transactions on Programming Languages and Systems* and IEEE TRANSACTIONS ON SOFTWARE ENGINEERING) and proceedings of international conferences (including ACM POPL, ACM PLDI, and IEEE LICS). According to Scopus, his current H-index is 23, with more than 1800 citations. He serves as the Co-Editor-in-Chief for the book series "*Services and Business Process Reengineering*" (Springer Nature).

Open Access funding provided by 'Università degli Studi di Parma' within the CRUI CARE Agreement