Chapter 2

Literature Review

This chapter serves as an introduction to the Ethereum ecosystem and offers an overview of the relevant literature.

Section I introduces the Ethereum Virtual Machine (EVM) and explores its gas consumption mechanism. Following that, Section II offers an examination of the existing languages compatible with the EVM. Section III provides a concise overview of compiler construction, and Section IV presents the overview of the Elixir programming language. To wrap up, Section V concludes this chapter with key decisions and findings from the literature.

I Ethereum Virtual Machine

According to [1] Ethereum is a decentralized system, and at its core, the EVM is an essential component of the Ethereum blockchain. The EVM serves as the computational heart of the Ethereum network, enabling the execution of smart contracts and providing the foundation for a wide array of decentralized applications.

EVM has a significant security mechanism - gas limit. Each operation and computation on the EVM consumes a certain amount of gas, which is a measure of computational work. The gas limit is a cap set by the user or the entity initiating a contract execution, and it represents the maximum amount of gas they are willing to spend for that operation.

The gas limit, the stack-based architecture of the EVM, and the wide set of available operations allow it to efficiently and securely for miners execute these smart contracts. Also, Buterin [1] defines EVM as a Turing-complete machine. This level of computational flexibility empowers developers to create sophisticated programs.

II Existing EVM languages

Official Ethereum documentation [2] enumerates only four languages that compile to EVM: Solidity, Vyper, Yul, and Fe. Also, other attempts to create a language for EVM are known[1], but only officially documented languages will be considered in this paper.

Among the available languages for EVM, Solidity stands out as the most widely adopted and utilized one [3], [4]. This dominance is further emphasized by data presented in [2], which reveals that there are approximately 800 times more Solidity files on Github compared to Vyper files. Consequently, the majority of existing smart contracts on the Ethereum blockchain are written in Solidity. On one hand, the concentration of development and resources around a single language like Solidity can be viewed positively. It fosters a strong

[1]A curated collection of resources on smart contract programming languages

[2]Analyzing Solidity and Vyper for Smart Contracts Programming

sense of community cohesion and ensures that the majority of developers are focused on refining and improving a single language, potentially enhancing its robustness and feature set. However, this dominance also brings significant challenges. The high prevalence of Solidity in the EVM ecosystem means that if a vulnerability or security flaw arises in the Solidity language, it affects a substantial portion of the smart contracts on the network. This centralized risk can be a double-edged sword, as a single point of failure in Solidity could have far-reaching consequences for the entire Ethereum blockchain. Furthermore, the Ethereum community is not without its criticisms of Solidity. Community members have voiced their concerns and challenges related to the language [4]. These concerns range from the complexity of the language to its lack of certain features, which may hinder the development of robust and secure smart contracts. So, decentralization and availability of choice are important, especially for such a decentralized ecosystem as blockchain.

A. Solidity

Solidity is a high-level, object-oriented programming, statically typed language designed specifically for the EVM. Its syntax is similar to C++, Python, and Javascript [5]. It supports multiple inheritance, complex user-defined types, and libraries as well as the number of other features [6].

B. Vyper

Vyper contrasts significantly with Solidity, it is based on Python and extends it to suit smart contract development. Vyper aims for security and simplicity in terms of compiler implementation itself and code readability to be auditable more easily. For instance, it lacks a list of features that are considered harmful for code readability, and that makes the language more error-prone [7].

III Compiler construction

The process of compiler construction is a crucial bridge between high-level programming languages and the machine code that EVM understands. This process can be divided into three or four main stages: tokenization, parsing, intermediate representation, and code generation. Each playing a pivotal role in transforming human-readable code into executable instructions that can be processed by the EVM. Here is a general overview of these stages:

A. Tokenization

According to Waite and Goose [8, pp. 135–148] tokenization is the initial step in the compilation process. During this stage, the source code is broken down into smaller units called tokens. These tokens are fundamental building blocks, including keywords, identifiers, operators, and constants. Tokenization simplifies the process of analyzing and parsing the code by providing a structured representation of the code.

Tokenization facilitates syntactical and lexical analysis, ensuring that the code adheres to the grammar of the language and can be properly understood.

B. Parsing

Parsing follows tokenization and focuses on the syntax of the source code. During parsing, the compiler checks the arrangement and structure of tokens to create a structured representation, often in the form of a syntax tree or abstract syntax tree (AST). This representation helps ensure that the code conforms to the grammar of the language rules and can be further processed [8, pp. 149–182].

C. Intermediate Representation (Optional)

While not always a mandatory stage, the use of an intermediate representation can significantly enhance the efficiency and capabilities of a compiler. In the context of EVM, intermediate representations like Yul and Yul+ are used for optimizing code. These representations help in enhancing code quality, optimizing performance, and preparing the code for final code generation.

This stage is particularly important in scenarios where verification, security, and performance optimizations are key concerns.

D. Code Generation

The final stage of the compilation process involves code generation [8, pp. 253–281]. During this stage, the compiler generates the actual machine code that the EVM can execute. The generated code is specific to the hardware architecture of the EVM and represents a translation of the original high-level code into instructions that the EVM can understand and execute.

The code generation stage is crucial in ensuring that the compiled code is efficient and correctly reflects the intended behavior of the high-level source code.

In conclusion, compiler construction is fundamental to the implementation of the new language, enabling the translation of high-level languages into machine code. The stages of tokenization, parsing, optional intermediate representation, and code generation are essential components of this process.

IV The Elixir language [9]

Elixir is a modern programming language that operates on top of the Erlang Virtual Machine. With its support for immutable variables, Elixir greatly enhances the clarity and simplicity of smart contract code. This immutability ensures that once data are defined, they cannot be altered, promoting predictability and security in the context of smart contract development.

A standout feature of Elixir is its metaprogramming capabilities. This functionality empowers developers to create domain-specific languages that are tailor-made for the intricacies of smart contract logic. This capability is invaluable as it allows developers to express contract logic in both highly readable and closely aligned with the specific problem domain way. By employing metaprogramming, smart contract development becomes more straightforward, and it becomes more accessible to a broader spectrum of developers.

As attested by O'Grady [10], Elixir stands strong among the top 50 programming languages. Remarkably, it surpasses even the popularity of Solidity, making it an attractive choice for Ethereum developers and thereby contributing to the expansion of the Ethereum community.

In summation, our decision to adopt Elixir as the language for our compiler and as the foundational framework for the Elixireum language is bolstered by its immutability, metaprogramming capabilities, and its prominence in the software development landscape. This choice ensures not only the simplicity and security of smart contract development but also the growth and diversification of the Ethereum ecosystem.

V Conclusion

In conclusion, this chapter has emphasized two significant matters. Firstly, the overwhelming dominance of Solidity has brought substantial challenges to the Ethereum ecosystem. Secondly, the absence of functional languages for smart contract development has limited the utilization of functional paradigm advantages. Consequently, we have decided to develop Elixirium, a functional programming language built on Elixir, tailored specifically for smart contract development.

Bibliography cited

[1] V. Buterin. “Ethereum whitepaper.” (2014), [Online]. Available: https://ethereum.org/en/whitepaper.

[2] “Smart contract languages.” Accessed Oct. 22, 2023. (2022), [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts/languages/.

[3] D. Harz and W. J. Knottenbelt, “Towards safer smart contracts: A survey of languages and verification methods,” ArXiv, vol. abs/1809.09805, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52844161.

[4] W. Zou, D. Lo, P. S. Kochhar, et al., “Smart contract development: Challenges and opportunities,” IEEE Transactions on Software Engineering, vol. 47, no. 10, pp. 2084–2106, 2021. DOI: 10.1109/TSE.2019.2942301.

[5] “Language influences.” Accessed Oct. 29, 2023. (2021), [Online]. Available: https://docs.soliditylang.org/en/v0.8.22/language-influences.html.

[6] “Solidity.” Accessed Oct. 29, 2023. (2023), [Online]. Available: https://docs.soliditylang.org/en/v0.8.22/.

[7] "Vyper." Accessed Oct. 29, 2023. (2020), [Online]. Available: https://docs.vyperlang.org/en/stable/.

[8] G. G. William M. Waite, Compiler Construction. 1984.

[9] "Elixir." (2023), [Online]. Available: https://elixir-lang.org/ (visited on 11/14/2023).

[10] S. O'Grady. "The redmonk programming language rankings: January 2023." Accessed Oct. 29, 2023. (2023), [Online]. Available: https://redmonk.com/sogrady/2023/05/16/language-rankings-1-23/.