List of Figures

Chapter 3

Literature Review

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

Section 3.1 introduces the Ethereum Virtual Machine (EVM) and explores its gas consumption mechanism. Following that, Section 3.2 examines of the existing languages compatible with the EVM. Section 3.3 provides a concise overview of compiler construction, and Section 3.4 presents an overview of the Elixir programming language. To conclude, Section 3.5 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 – 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 broad set of available operations allow it to execute these smart contracts efficiently and securely for miners. 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 than Vyper files. Consequently, most existing smart contracts on the Ethereum blockchain are written in Solidity. On the one hand, the concentration of development and resources around a single language like Solidity can be viewed positively. It fosters a strong sense

[1] A curated collection of resources on smart contract programming languages
[2] Analyzing Solidity and Vyper for Smart Contracts Programming

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 essential, especially for a decentralized ecosystem like blockchain.

A. Solidity

Solidity is a high-level, object-oriented programming, statically typed language explicitly designed for the EVM. Its syntax is similar to C++, Python, and Javascript [5]. It supports multiple inheritance, complex user-defined types, libraries, and some 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 considered harmful for code readability, making 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 plays a pivotal role in transforming human-readable code into executable instructions that the EVM can process. 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 enhance code quality, optimize performance, and prepare 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 implementing the new language, enabling the translation of high-level languages into machine code. The stages of tokenization, parsing, optional intermediate representa-

tion, and code generation are essential components of this process.

IV The Elixir language [9]

Elixir is a modern programming language that operates on the Erlang Virtual Machine. With its support for immutable variables, Elixir dramatically 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 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 prominence in the software development landscape. This choice ensures the simplicity and security of smart contract development and 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, explicitly tailored for smart contract development.

Chapter 4

Methodology

Chapter overview

I Compilers and evm theory

A. Introduction to compilers

Compiler construction is a sophisticated field in computer science, involving the creation of software that translates source code written in a high-level programming language into a lower-level language, often machine code. The process of compiler construction is typically divided into several distinct phases, each with its specialized function:

1. Lexical Analysis

    - Token Generation: The compiler's first phase involves breaking down the source code into tokens, which are the basic building blocks of the language (like keywords, operators, identifiers, literals).

- Regular Expressions: Lexical analyzers often use regular expressions to recognize these tokens, simplifying the source code into a stream of tokens for further processing.

2. Syntax Analysis

   - Parsing: This phase involves analyzing the token sequence from lexical analysis to ensure that the expressions and statements conform to the rules of the programming language's grammar.
   - Syntax Trees: The result is often a syntax tree, which represents the syntactic structure of the source code in a hierarchical tree format.

3. Semantic Analysis

   - Type Checking: The compiler checks whether the parsed code makes logical sense, adhering to the language's semantic rules. This includes type checking, where the compiler verifies that each operation is compatible with its operands.
   - Symbol Table: The compiler uses a symbol table to keep track of variable names, their types, and other relevant information to ensure correct interpretation and context.

4. Intermediate Code Generation

   - Abstract Representation: The compiler translates the source code into an intermediate representation, which is a lower-level representation of the program that retains its logical structure.

- Platform-Independent Code: This intermediate code is typically platform-independent, serving as a bridge between the high-level source code and the machine code.

5. Optimization

   - Performance Improvement: The optimization phase refines the intermediate code to improve efficiency and performance. This can include eliminating redundant calculations, optimizing loops, and more.
   - Target-Dependent Optimizations: These optimizations often vary depending on the target machine's architecture and capabilities.

6. Code Generation

   - Machine Code: The final phase involves translating the optimized intermediate code into machine code specific to the target machine CPU architecture or virtual machine instruction set.
   - Assembly Language: Sometimes, the output is in the form of assembly language, which is then further translated into machine code by an assembler.

7. Linking and Loading

   - Linking: If the program consists of multiple source code files or external libraries, the linking process combines these into a single executable.
   - Loading: The loader then places the executable into memory for execution by the machine.

In the context of Elixireum, these compiler construction principles must be adapted to suit the specific needs of the Ethereum Virtual Machine (EVM). This includes considerations for smart contract compilation, optimization for gas efficiency, and adhering to the deterministic execution model of the EVM. Understanding these compiler construction phases is essential for Elixireum development, as each stage plays a critical role in transforming high-level Elixireum code into efficient, secure, and EVM-compatible bytecode.

B. Ethereum Virtual Machine - A Deeper Insight

EVM is a core component of the Ethereum blockchain, pivotal in maintaining its decentralized and secure nature. It serves as an execution environment for smart contracts, enabling the remarkable functionality that Ethereum offers. Understanding the EVM is crucial for developing Elixireum, as it provides the foundational framework within which this new language operates.

1. Execution Environment

    - Isolated System: The EVM operates as an isolated environment, completely sandboxed from the Ethereum network. This isolation ensures that code execution within the EVM does not directly affect the network or the filesystem of the host computer.
    - Deterministic Execution: The EVM is designed to be deterministic. This means that executing the same smart contract with the same input will always produce the same output across all nodes in the Ethereum network. This characteristic is essential for maintaining consistency and agreement (consensus) across the decentralized network.

2. EVM's Role in Smart Contracts

    - Smart Contract Lifecycle: Developers write smart contracts in high-level languages like Solidity or Vyper. These contracts are then compiled into EVM bytecode. When a contract is deployed on the Ethereum blockchain, it's executed by the EVM to ensure it behaves as expected.
    - Gas Mechanism: The EVM uses a gas mechanism to measure and limit the resources (computation and storage) a contract can use. Each operation in the EVM has a gas cost associated with it, preventing inefficient or malicious code from overloading the network.

3. State Machine

    - State Transitions: The EVM can be viewed as a state machine, where transactions transition the Ethereum blockchain from one state to another. Smart contracts, through their executions, modify the state stored in the Ethereum blockchain.
    - EVM Bytecode: The EVM interprets a series of bytes (bytecode) which instruct the machine on the operations to perform. This bytecode is the compiled version of high-level contract code.

4. Storage and Memory

    - Storage: The EVM has a long-term storage model, where data is stored persistently on the blockchain. This storage is expensive but necessary for maintaining the state across transactions.

- Memory: The EVM also includes a volatile memory space, cleared after each transaction, used for temporary storage during contract execution.

5. Challenges and Considerations for Elixireum

- Gas Optimization: Given the cost implications of EVM operations, Elixireum must focus on gas optimization during the compilation process.
- Security and Determinism: Security vulnerabilities in smart contracts can be costly. Hence, Elixireum needs to ensure robust security measures. Additionally, the deterministic nature of the EVM requires that Elixireum-generated bytecode behaves consistently across all executions.

II Problems

Given the unique syntax borrowed from Elixir for the language we developed, our journey in crafting Elixireum often led us to delve into the implementation of Elixir itself. Due to Elixir's niche popularity, resources on writing a compiler in Elixir are scarce, and at times, completely nonexistent on the internet. Consequently, we found ourselves independently sourcing this information, relying heavily on the original codebase, experimentation, and our accumulated expertise. Similarly, our encounter with the intermediate Yul language presented its own set of challenges. The entirety of its documentation seems to be condensed into a single chapter within the Solidity documentation, leading to a notable dearth of practical usage examples and detailed

explanations of certain concepts. Faced with these constraints, we embarked on a quest for knowledge through hands-on experimentation and meticulous analysis of the opcodes found within the bytecodes of various examples. This approach allowed us to uncover and understand the intricacies of Yul, despite the limited resources at our disposal.

III Design and implementations

This section outlines the principal elements of the Elixireum compiler's design, focusing specifically on the methodology employed to compile Elixireum code into EVM bytecode. It overviews the technology stack employed in the development process and underscores the pivotal decisions that shaped the architecture of the compiler.

A. Choice of Elixir

- The rationale for selecting Elixir is straightforward: our team possesses substantial expertise in this programming language.
- Elixir is open source so we can easily reuse its machinery for tokenization and parsing Elixireum.

B. Choice of Yul

- Initially, our strategy entailed compiling Elixireum directly into Ethereum Virtual Machine (EVM) bytecode. Subsequently, we pivoted towards an intermediate representation (IR), selecting Yul as IR aligned with our requirements. Advantages from this decision encompass:

  - Compatibility with all EVM versions, obviating the need for version-specific concerns.
  - Avoidance of reimplementing primitives such as function calls, variable and stack management, thereby freeing up resources to enhance the feature set of the language.
  - Reduced gas consumption facilitated by the ability to optimize Yul code of Solidity compiler.

- Despite these considerations, the choice comes with limitations, notably:
  - Dependency on the Solidity compiler for Elixireum.

  No significant downsides compared to direct compilation to EVM bytecode, as bytecode can be used in Yul directly.

- We considered the mapping of BEAM (Erlang VM) bytecode to EVM bytecode as an option as well. This option was deemed challenging due to the fundamental architectural and philosophical differences between the two virtual machines. BEAM is specifically tailored for environments that demand high concurrency, distribution, and fault tolerance, boasting a set of opcodes optimized for these conditions. In contrast, the EVM is built to run smart contracts within blockchain ecosystems, prioritizing deterministic execution and gas metering to safeguard against spam and enhance the security of the network. Due to these distinct focuses, BEAM incorporates specialized instructions for concurrency and distributed processing, which do not directly translate to EVM's capabilities.

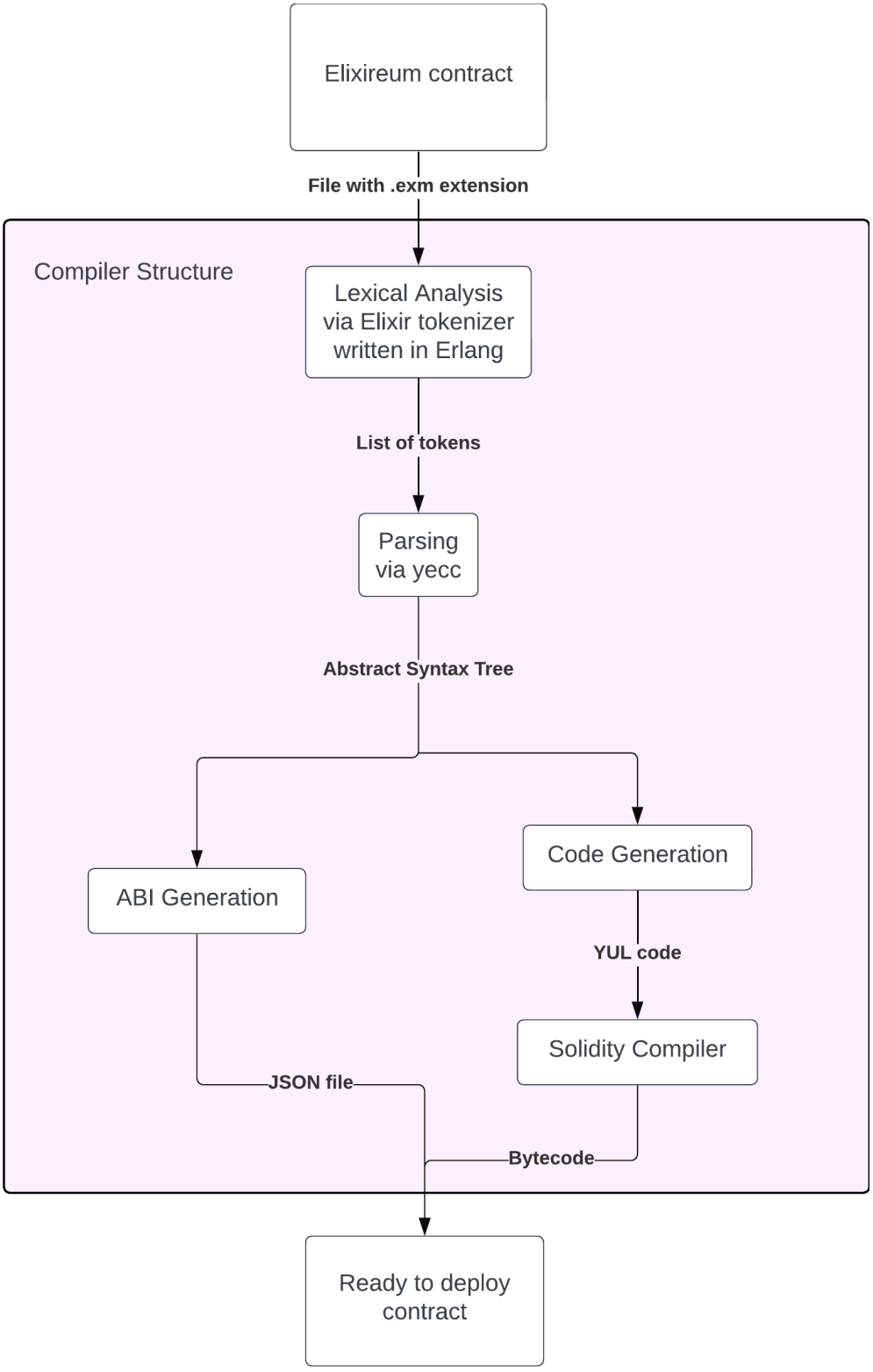

Fig. 1. Elixireum compiler architecture

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/.