Cryptic Contracts: The State of Smart Contract Transparency

Amine Smaoui, Richard von Seck*, Filip Rezabek*, and Kilian Glas*
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: amine.smaoui@tum.de, seck@net.in.tum.de, rezabeck@net.in.tum.de, glask@net.in.tum.de

***Abstract*—Blockchain technology has seen the emergence of new cryptocurrencies that use smart contracts. A smart contract is a self-executing code that, when deployed, is compiled into lower-level bytecode, which is usually more complex to grasp from a human perspective than high-level language. This complexity can hide several errors or vulnerabilities, potentially leading to stolen assets and undermining the network's stability. Consequently, there is a need for tools and decompilers to reverse engineer the process, gain an understanding of their logic, and enhance their security. With this paper, we seek to understand and compare various techniques for enhancing smart contract transparency on the major Blockchains of Ethereum, Algorand, and Dfinity's Internet Computer. We collect recent literature mainly focused on smart contract deployment, decompilation, and analysis tools. These tools and their approaches are examined and evaluated based on the transparency level they provide. Subsequently, we conclude that Ethereum has the best decompilation support, positioning it as a trustworthy and transparent Blockchain to build decentralized applications.**

***Index Terms*—Smart Contracts, Ethereum, Algorand, Dfinity, Decompilation, Reverse Engineering**

1. Introduction

Blockchain has gained significant popularity over the last few years, and even financial institutions are adopting it for their transactions [1]. Several Peer-to-Peer systems, such as Ethereum (introduced in Subsection 3.1), support smart contracts [2] that act as trustworthy, self-executing middlemen. Ethereum compiles the smart contract's high-level code into less readable bytecode to be processed by the distributed network. This procedure complicates the possibilities of error and vulnerability detection.

One of the significant factors that intensified the scientific community's interest in smart contracts decompilation was the "DAO Attack" [3]. This happened in 2016 due to a vulnerability in smart contract code; an attacker succeeded in controlling around 60 million US dollars worth of Blockchain tokens, called Ether, on Ethereum. Researchers have since started focusing more on methods that analyze, debug, and potentially decompile the bytecode to understand the vulnerabilities and prevent the recurrence of such attacks.

Our goal in this paper is to examine the efforts deployed in reverse engineering the process of smart contracts on Ethereum, Algorand, and Dfinity's Internet Computer, aiming to provide an overview of the effectiveness and usability of these tools.

1.1. Blockchain

Blockchain [4] was unveiled to the world in 2009 with the introduction of Bitcoin [5]. It was presented as a tool enabling decentralized, immutable, and transparent digital transactions. However, these properties may vary depending on the type of Blockchain. We differentiate three types of Blockchains [6]: Public, private, and hybrid Blockchain. In this paper, our work mainly focuses on public Blockchains. This type of Blockchain is an open, distributed ledger system that is available to everyone.

Decentralized systems, such as Blockchain, are designed to eliminate the need for a central authority to monitor the network traffic and approve transactions. These tasks are performed by network users, also known as nodes. The immutability property of this architecture means that all data recorded on the public ledger can no longer be modified after the approval of the network nodes [7]. This property is ensured thanks to "a consensus mechanism, cryptography, and back-referencing blocks." The transparency of this technology stems from the fact that all transactions on the Blockchain are recorded on a public ledger. These actions are visible and verifiable by all participants [4].

1.2. Smart contracts

Nick Szabo [8] introduced the term *smart contract* and presented it as "a set of promises, specified in digital form, including protocols within which the parties perform on these promises." [9].

Nowadays, smart contracts use Blockchains as their underlying platform [10]. After agreeing on the contract details, these are translated into computer code that can be executed automatically. To replicate the decision-making process, smart contracts usually rely on programming structures, such as "if-else statements," and every action taken is recorded on the ledger and cannot be changed. A penalty can also be predetermined if a contract's party does not honor the agreement. This penalty is automatically subtracted from the violator's deposit.

A smart contract life cycle [10] starts from written code in a programming language supported by the Blockchain. This code is compiled and stored in the network; henceforth, it cannot be altered. Additionally, the funds of participating members are frozen in their digital wallets. The execution of smart contracts is similar

 doi: 10.2313/NET-2024-09-1_06

to buying from a vending machine [11]. In contrast to buying from a supermarket, where you need to interact with a cashier, the process of purchasing from a machine is fully automated; The buyer throws in enough coins, presses the button, and gets the product. The final step involves unlocking parties' funds and updating the states of all contract parties.

Since our main focus is reverse engineering smart contracts, presenting the procedure and the challenges encountered is important to our transparency study process. A decompiler's job is to convert bytecode back into its source code format. This protocol [12] generally unfolds as follows: Firstly, the tool tries to decode the binary files of the contract and converts them into a stream of instructions and other important data. This phase is prone to errors due to various language version compatibility issues with binary file formats. The instruction streams generated in the previous step are transformed into assembly code. This is particularly challenging as it is hard to differentiate between program code and data. Assembly programs are then developed into various Intermediate Representation (IR) forms, from abstract syntax trees to control-flow graphs. Without control structures, this process is complex. Finally, the process is concluded by generating high-level code from the previous IRs. The reconstruction is a challenging procedure in Ethereum, for example, due to the absence of identifiers (variable names and types).

In this survey, we study the decompilation and analysis processes of various tools and evaluate their effectiveness.

2. Related work

Numerous studies have covered this topic but usually target only one Blockchain. For instance, Liu et al. [12] conducted an empirical study of Ethereum's smart contract decompilers, gathering the insights and challenges of the major tools present on the market in one paper. On Algorand, the research mainly focused on formal verification and analysis tools. Notable contributions include the work by Bartoletti et al. [13] and analysis tool Panda [14]. On the other hand, efforts on Dfinity's Internet Computer for smart contract decompilation were relatively limited, with the primary focus on WebAssembly (discussed in Subsection 3.3) decompilation efforts made by Dfinity [15] and Google [16] to debug and analyze the low-level language code. In this paper, we aim to bridge these gaps by providing a comprehensive overview of smart contract transparency across these various ecosystems.

3. Representative Blockchains

The following sections present a detailed overview of each ecosystem focusing on how they manage and implement smart contracts.

3.1. Ethereum

Buterin [17] introduced Ethereum in 2013 as an "alternative protocol for building decentralized applications," it has since fulfilled its promise, developing into one of the major platforms in the Blockchain space. Ethereum's Blockchain technology goes beyond financial transactions; it has several real-life applications such as insurance, saving wallets, or even cloud computing.

Ethereum's differs from the Bitcoin Blockchain by integrating a Turing-complete high-level programming language, Solidity. A Turing-complete [18] language is a programming language capable of creating and computing any wanted program. Additionally, a run-time environment that supports smart contracts [19], called Ethereum Virtual Machine (EVM) was implemented. It executes smart contract's bytecode, also known as EVM Bytecode. These tools allow users to directly interact with the Blockchain by creating their own smart contracts and decentralized applications (dApps) for different purposes beyond just currency exchange. As a result of Solidity's properties, the network members are able to perform any computable function within the Ethereum ecosystem. However, the execution is not free of charge; it is influenced by the "gas" cost, which is merely the price of computational efforts on the Blockchain [17].

3.2. Algorand

First presented in 2017 as a low latency and highly scalable cryptocurrency, Algorand [20] uses a Pure Proof of Stake [21] consensus mechanism. To ensure security, this mechanism applies various techniques based on the Byzantine Agreement protocol (BA*). It creates groups of nodes called "committees" that approve the transactions. BA* ensures that 2/3 of the weighted users in the committee are honest. The same protocol applies cryptographic sortition to privately choose committee members, hence protecting them from targeted attacks. The random sortition process is guaranteed by the so-called "Verifiable Random Function" (VRF), a random number generator that expresses, among others, the probability of the user being part of the committee. Moreover, Algorand's Byzantine Agreement allows committee members to contribute once, and then they are generally replaced. This prevents members from being deliberately targeted by attackers and jeopardizing the consensus. This protocol is designed to reach consensus on transactions securely. One of Algorand's main features is its low-level bytecode-based stack language, the Transaction Execution Approval Language (TEAL) [22]. Introduced as a non-Turing-complete programming language, this property helps reduce the risk of attacks [13]. However, Python provides a high-level language alternative. "Pyteal" is specially tailored to write smart contracts on Algorand. TEAL is executed as a script and returns a boolean that either approves or rejects the transaction. An important additional feature of Algorand is its Virtual Machine (AVM) [14], capable of executing the bytecode resulting from compiled TEAL code.

Smart contracts in Algorand are categorized into single State and Algogeneous contracts [22]. Single State contracts can be used for various purposes such as transactions and creating applications. Such smart contracts can be Stateless or Stateful and they have distinct functions. Stateless smart contracts are primarily used for transaction validation. They approve and deny transactions and can also serve as "signature delegators". On the other hand, stateful smart contracts are mostly used to store and manage data on the Blockchain. Both smart contract types could be combined to produce complex applications.

Algogenous contracts represent a more advanced type of smart contracts. They comprise the functionalities of both Stateless and Stateful contracts. This design allows it to do multiple tasks, combining validation and verification.

3.3. Dfinity's Internet Computer

Dfinity's Internet Computer [23] represents a relatively new member of the Blockchain family. The particularity of this Blockchain is its use of a hybrid model, named DAO-controlled network, which is a consensus mechanism based on subnets that use a permissioned consensus mechanism. These subnets are chosen by the network nervous system (NNS) to manage the network functions. This step is similar to PoS, as the members of the network stake tokens to vote for the entities that create "replicas" and perform other tasks. These replicas are stored on distributed servers to ensure their security. On the Internet Computer, smart contracts are written in a high-level language, such as Rust [24] or Motoko, a Dfinity-tailored language that aligns with IC's semantics. The written high-level code is then compiled down to WebAssembly (Wasm), a binary instruction format that provides a way to run code on the web. After the compilation, the program is then deployed on the Blockchain in a Canister [23]. A Canister is similar to the concept of a "process" in traditional computing, they are coded in Wasm and consist of a program and its state. Canisters run autonomously on the Internet Computer and interact with each other through an interface called Candid.

4. Smart Contract Transparency

The methodologies applied for the transparency analysis differ from one Blockchain platform to another. The following subsections handle the specifics of smart contracts transparency approaches used by major platforms.

4.1. Smart Contracts decompilation on Ethereum

Writing smart contracts directly in EVM bytecode, an assembly-like language [25], is an intricate operation and rather prone to errors. Therefore, Ethereum supports various high-level programming languages besides Solidity to implement smart contracts, such as Vyper [26]. The compilation process from high-level to EVM bytecode occurs before deploying to the network. A smart contract's bytecode comprises three parts [12]: a deployment code is responsible for deploying smart contracts on Ethereum. It is put to execution as soon as it is created. It also checks if the function can receive Ether payments. Runtime Code defines the contract's functionality on the Blockchain. Auxiliary data contains a hash value linked to the metadata of the deployed contract and can be used for verification.

Having established the fundamental challenges of smart contract decompilation in Subsection 1.2, we will now focus on the approaches and solutions adopted by decompilers to tackle these issues.

The EVM uses 256-bit pseudo-registers containing 160-bit addresses called "accounts" to identify them. EVM's pseudo-registers fundamentally operate as a stack, which facilitates passing parameters to perform various operations [27]. This observation is exploited by Porosity [27] to extract the addresses from the bytecode using bitmasks to isolate the 160-bit address from the 256-bit EVM pseudo-registers. Gigahorse [28] addresses the issues of disassembly and intermediate representation by applying "Context-sensitivity" that considers varying states and conditions available at each step of the program. Basically, heuristics are used to determine the program control-flows. Elipmoc [29], a decompiler based on Gigahorse, uses the same principle but creates IRs using only stack locations that might contain jump targets. Using this technique, Elipmoc claims a 99,5% success rate in fully decompiling Smart Contracts. A better ratio than Gigarhorse's 62,8% decompilation rate.

4.2. Smart Contracts Transparency on Algorand

Although significant efforts have been deployed toward smart contract decompilation on Algorand, fully decompiled smart contracts are still not the standard. Other approaches and methodologies are frequently employed.

Stateless Smart Contracts on Algorand or ASC1 [13], are programmed using non-Turing-complete language to reduce vulnerability risks. However, there are still potential threats without a formalized mathematical model that ensures the contract's accuracy and security. Bartoletti et al. [13] decided to create a formal model that defines the behavior of Algorand accounts, transactions, and smart contracts using a state machine that acts to fundamentally understand their functioning and experiment on them. An attacker model was also developed to simulate attacks on their formal smart contract model. This model can potentially be a valuable tool for debugging and identifying security flaws and susceptible points of attack.

Panda [14], a security analysis tool of Algorand smart contract, has an architecture composed of several components. The main components we are interested in are: **(1)** The user interface for user input and settings, **(2)** a Blockchain Explorer that fetches the TEAL bytecode from the contract and disassembles it, **(3)** a control-flow graph (CFG) Builder generates graphs from TEAL programs, and **(4)** a symbolic executor analyzes these CFGs and processes each command symbolically. Moreover, detection rules are defined to address any already known vulnerabilities. Sun et al. [14] have reported several security concerns on Algorand. They were later categorized into five key groups. Three of these categories affect application operations, while the remaining two are vulnerabilities in smart signatures (also known as stateless smart contracts). We examine the vulnerabilities related to stateless smart contracts.

The first refers to an "Unchecked Transaction Fee." This vulnerability is generally caused by transaction fees that are not properly restricted. This allows an attacker to set high transaction fees and deplete the account's funds.

The second weakness was called 'Unchecked Transaction Parameters.' This describes vulnerabilities arising from inadequate verification of transaction parameters. One of the function arguments in the transaction code sets the authorized address for future exchanges. An attacker can gain access to the signature account by modifying this field. Another important parameter directs where the remaining balance of an account should go when the

transaction is closed. The attacker can drain all the Algos (Algorand's native currency) from the signature account by setting this parameter to their address.

4.3. Smart Contracts transparency on Dfinity's Internet Computer

To have an overview of the transparency efforts of smart contracts on the Internet Computer, one needs to look at the IC's Wasm Libraries and the key tools provided and developed by Dfinity, especially for Canisters. The ic_wasm library (v0.7.1) [15] includes an experimental feature that instruments Canisters, which can help debug and analyze their code. For instance, the instrumented Canister can provide additional endpoints to access the execution trace log and the current cycle counter. Furthermore, certain flags can also be added to trace logging of a specific function during its execution. Google on the other hand has developed a toolkit to debug and analyze WebAssembly code, namely The WebAssembly Binary Toolkit (WABT) [16]. This repository contains several helpful tools, for instance, **wat2wasm** and **wasm2wat** convert between WebAssembly text and binary formats. Furthermore, **wasm-objdump** generates an objdump providing a general overview of the code structure. Additionally, this toolkit contains a Wasm stack-based interpreter that executes binary files. Another significant tool is a Wasm decompiler that converts a Wasm binary file into an intelligible C-like syntax, providing readable partial decompilation. Although WABT is not directly associated with Dfinity's Internet Computer, it still has potential general application to a wide range of wasm-written code. However, this toolkit's capabilities might eventually be less effective in some IC-specific cases.

5. Discussion

In Section 4, we explore the methodologies and approaches used to fully decompile smart contracts or analyze them through other methods, such as debugging. Given that each discussed platform differs by its fundamental architecture, consensus mechanism, smart contract deployment, publication date, and development phase, a direct comparison might not provide equitable insights. Therefore, we perform a case-by-case analysis to better understand the state of smart contract's transparency on each ecosystem independently.

Ethereum has the most advanced development phase compared to the other platforms, with functional smart contracts decompilers and relatively high success rates. Although each decompiler claims to have one of the best correct decompilation ratios, these percentages generally depend on the used dataset and smart contract types. Consequently, we will refer in our analysis to the work of Liu et al. [12] as the main source of data. This recent empirical study tests with the same dataset on major decompilers and rates their execution based on mathematical formulas.

When applying decompilers on normal datasets with the compiler optimizations turned on, Gigahorse [28], Vandal [30], and Ethervm [31] had success rates all above 99.70%. Whereas Panoramix [32] succeded 98,14% of the time, leaving Erays [33] with the least success rate of 63,92%. Another test on a dataset of buggy contracts yielded a 100% decompilation rate for Gigahorse and EtherVM. Vandal had one failed decompilation, Panoramix encountered 21 failures, and Erays could not decompile 359 buggy contracts.

These results indicate that smart contract decompilers on Ethereum have achieved significant milestones. However, there is still room for improvement, especially with the accuracy and completeness of the decompilation.

Algorand's Panda and the formal model offered important insights to address the transparency problem in smart contracts, but their approaches are limited. Panda, for instance, can be helpful when detecting vulnerabilities, but the symbolic executor cannot process a certain type of opcode. Additionally, there are some challenges when identifying the Validator when the smart signature uses implicit invoking. The formal model, on the other hand, provides a theoretical approach to understanding smart contracts. This might not be suited for practical evaluation since it does not capture every behavior of the Algorand implementation in real life and mainly focuses on stateless smart contracts.

On IC, using the instrumentation feature of 'ic_wasm' could be a tool for understanding smart contracts. Although valuable for bug identification, it lacks the required transparency to fully comprehend the logic of the contract. WABT, however, presents significant means for Wasm analysis and debugging. The tools offered can even partially decompile code. However, this toolkit is more of a general debug and analysis tool and not IC-specific, therefore, its ability to address certain tasks from the Internet Computer might be limited.

Table 1 summarizes the available tools for smart contract decompilation and transparency across the different ecosystems, based on the papers mentioned in this survey:

TABLE 1: Availability of Smart Contract Transparency tools across Blockchains

Platform	Debug	Analysis	Par. Dec[1]	Full Dec[2]
Ethereum	✓	✓	✓	✓
Dfinity	✓	✓	∼[3]	×
Algorand	✓	✓	×	×

1. Partial Decompilation
2. Full Decompilation
3. Not enough data available to make a statement

Table 1, shows a complete set of tools available for Ethereum, ranging from debugging to full decompilation. While Algorand only has debugging and analysis tools, Dfinity offers additionally a potential partial decompiler as well as debugging and analysis transparency instruments. The key factors responsible for these disparities differ from one ecosystem to another. Ethereum for instance, is a more mature and established Blockchain. It has experienced rapid growth in popularity, and its applications, Decentralized Finance (DeFi), for example, have reached 200 billion US dollars in 2021 [34]. Indicating significant transaction traffic and enough data for scientists to work on. Algorand [22] and Dfinity's Internet Computer [23], on the other hand, are still relatively new. But Dfinity has an advantage since its high-level code is compiled into WebAssembly. A widely utilized programming language

with various applications [35], in contrast to TEAL which is exclusively used on Algorand.

6. Conclusion and future work

In this paper, we analyze and compare multiple approaches to achieve smart contract transparency. These techniques range from decompiling to analysis and debugging. Their applicability depends on the Blockchain itself; for instance, in Algorand, only debugging and analysis tools were available. Dfinity's Internet Computer introduced, additionally to analysis and debug tools, a potential partial decompiler. On the other hand, smart contracts were successfully fully decompiled on Ethereum. The high transparency level of smart contract decompilation on Ethereum makes the platform more attractive for users and developers to create secure and optimized applications in a trustworthy and secure environment.

Future work could look into integrating machine learning for vulnerability detection and decompilation of smart contracts. Sendner et al. [36], and Gioka et al. [37] have already initiated research efforts towards this topic. Improving the decompilation and analysis tools will help anticipate potential risks and attacks, rendering the Blockchain a more secure platform for developing decentralized applications.

References

[1] M. Javaid, A. Haleem, R. P. Singh, R. Suman, and S. Khan, "A review of blockchain technology applications for financial services," *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, vol. 2, no. 3, p. 100073, 2022.

[2] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *International symposium on automated technology for verification and analysis*. Springer, 2018, pp. 513–520.

[3] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*. Springer, 2017, pp. 164–186.

[4] A. S. Rajasekaran, M. Azees, and F. Al-Turjman, "A comprehensive survey on blockchain technology," *Sustainable Energy Technologies and Assessments*, vol. 52, p. 102039, 2022.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009, accessed: Apr. 7, 2024. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[6] P. Paul, P. Aithal, R. Saavedra, and S. Ghosh, "Blockchain technology and its types—a short review," *International Journal of Applied Science and Engineering (IJASE)*, vol. 9, no. 2, pp. 189–200, 2021.

[7] F. Hofmann, S. Wurster, E. Ron, and M. Böhmecke-Schwafert, "The immutability concept of blockchains and benefits of early standardization," in *2017 ITU Kaleidoscope: Challenges for a Data-Driven Society (ITU K)*, 2017, pp. 1–8.

[8] D. Magazzeni, P. McBurney, and W. Nash, "Validation and verification of smart contracts: A research agenda," *Computer*, vol. 50, no. 9, pp. 50–57, 2017.

[9] N. Szabo, "Smart contracts: building blocks for digital markets," *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, no. 2, p. 28, 1996.

[10] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.

[11] R. Wilkens, R. Falk, R. Wilkens, and R. Falk, "Rechtliche aspekte," pp. 29–42, 2019.

[12] X. Liu, B. Hua, Y. Wang, and Z. Pan, "An empirical study of smart contract decompilers," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, pp. 1–12.

[13] M. Bartoletti, A. Bracciali, C. Lepore, A. Scalas, and R. Zunino, "A formal model of algorand smart contracts," in *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I 25*. Springer, 2021, pp. 93–114.

[14] Z. Sun, X. Luo, and Y. Zhang, "Panda: Security analysis of algorand smart contracts," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1811–1828.

[15] DFINITY Foundation, "ic-wasm: A library for transforming wasm canisters running on the internet computer," 2024, accessed: Apr. 7, 2024. [Online]. Available: https://github.com/dfinity/ic-wasm

[16] G. O. Source, "Wabt: The webassembly binary toolkit, binary," 2023, accessed: Apr. 7, 2024. [Online]. Available: https://chromium.googlesource.com/external/github.com/WebAssembly/wabt/+/refs/tags/binary_0xb/README.md

[17] V. Buterin, "Ethereum white paper: A next generation smart contract & decentralized application platform," 2013, accessed: Apr. 7, 2024. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[18] S. Kepser, "A simple proof for the turing-completeness of xslt and xquery." in *Extreme Markup Languages®*. Citeseer, 2004.

[19] Y. Fu, M. Ren, F. Ma, H. Shi, X. Yang, Y. Jiang, H. Li, and X. Shi, "Evmfuzzer: detect evm vulnerabilities via fuzz testing," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 1110–1114.

[20] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.

[21] C. Lepore, M. Ceria, A. Visconti, U. P. Rao, K. A. Shah, and L. Zanolini, "A survey on blockchain consensus with a performance comparison of pow, pos and pure pos," *Mathematics*, vol. 8, no. 10, 2020. [Online]. Available: https://www.mdpi.com/2227-7390/8/10/1782

[22] A. Chaudhury and B. Haney, "Smart contracts on algorand," 2021.

[23] T. D. Team, "The internet computer for geeks," Cryptology ePrint Archive, Paper 2022/087, 2022, accessed: Apr. 7, 2024. [Online]. Available: https://eprint.iacr.org/2022/087

[24] S. Klabnik and C. Nichols, *The Rust programming language*. No Starch Press, 2023.

[25] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 204–217.

[26] M. Kaleem, A. Mavridou, and A. Laszka, "Vyper: A security comparison with solidity based on common vulnerabilities," in *2020 2nd conference on blockchain research & applications for innovative networks and services (BRAINS)*. IEEE, 2020, pp. 107–111.

[27] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," *DEF con*, vol. 25, no. 11, 2017.

[28] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: thorough, declarative decompilation of smart contracts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1176–1186.

[29] N. Grech, S. Lagouvardos, I. Tsatiris, and Y. Smaragdakis, "Elipmoc: advanced decompilation of ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: https://doi.org/10.1145/3527321

[30] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[31] EtherVM, "Online solidity decompiler," 2024, accessed: Apr. 7, 2024. [Online]. Available: https://ethervm.io/

[32] palkeo, "Panoramix," 2024, accessed: Apr. 7, 2024. [Online]. Available: https://github.com/palkeo/panoramix

[33] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: reverse engineering ethereum's opaque smart contracts," in *27th USENIX security symposium (USENIX Security 18)*, 2018, pp. 1371–1385.

[34] P. Zheng, B. Su, Z. Jiang, C. Yang, J. Chen, and J. Wu, "Exploring heterogeneous decentralized markets in defi and nft on ethereum blockchain," in *2023 IEEE 10th International Conference on Cyber Security and Cloud Computing (CSCloud)/2023 IEEE 9th International Conference on Edge Computing and Scalable Cloud (EdgeCom)*. IEEE, 2023, pp. 259–267.

[35] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," in *Proceedings of the web conference 2021*, 2021, pp. 2696–2708.

[36] C. Sendner, H. Chen, H. Fereidooni, L. Petzi, J. König, J. Stang, A. Dmitrienko, A.-R. Sadeghi, and F. Koushanfar, "Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning." in *NDSS*, 2023.

[37] M. A. Gioka, I. Lagouvardos, and I. Tsatiris, "Machine learning aided tuning of static analysis for evm bytecode decompilation," 2020, accessed: Apr. 25, 2024. [Online]. Available: https://pergamos.lib.uoa.gr/uoa/dl/object/2923693/file.pdf