İSTANBUL TİCARET ÜNİVERSİTESİ
FEN BİLİMLERİ DERGİSİ

İstanbul Commerce University Journal of Science

http://dergipark.org.tr/ticaretfbd



Research Article / Araştırma Makalesi

A COMPARATIVE STUDY OF SMART CONTRACT PROGRAMMING LANGUAGES: SECURITY, PERFORMANCE, AND OPTIMIZATION TRADE-OFFs

AKILLI SÖZLEŞME PROGRAMLAMA DİLLERİNİN KARŞILAŞTIRMALI İNCELEMESİ: GÜVENLİK, PERFORMANS VE OPTİMİZASYON DENGELERİ

Hüseyin Ahmet Cemil ÖZASLAN[1] **Şafak DURUKAN ODABAŞI**[2]

https://doi.org/10.55071/ticaretfbd.1790366

Corresponding Author (Sorumlu Yazar)	*Received (Geliş Tarihi)*	*Revised (Revizyon Tarihi)*	*Accepted (Kabul Tarihi)*
safak.odabasi@iuc.edu.tr	24.09.2025	13.10.2025	18.10.2025

Abstract

Smart contracts have become a fundamental component of blockchain ecosystems, and their reliability is strongly shaped by the programming languages in which they are written. While prior studies have classified vulnerabilities, fewer have quantitatively examined how language design and secure coding practices affect performance and resilience. To address this gap, this study empirically compares Solidity and Vyper under controlled conditions and complements these experiments with a literature-based evaluation of Rust and Move. Test scenarios included deployment, deposits, withdrawals, arithmetic overflow, reentrancy, and transaction origin misuse. For both vulnerable and secure variants in Solidity and Vyper, metrics such as gas consumption, deployment size, and runtime execution time were collected. The results indicate that deployment costs differ substantially between the two languages (Solidity ≈ 177 k gas vs Vyper ≈ 135 k gas, ~24% lower), whereas runtime performance is mixed: deposit calls are nearly identical (Δ ≈ 0.02 ms), whereas withdraw shows a noticeable gap (Δ ≈ 4.97 ms) favoring Vyper; nevertheless, these call-level differences remain small relative to the larger deployment-time gap. Importantly, secure coding practices such as explicit arithmetic checks and the Checks–Effects–Interactions pattern eliminate critical vulnerabilities while adding less than 1% to the overall execution cost. Although Rust and Move are considered through a literature-based review, they illustrate alternative approaches that embed security guarantees directly into the language. Based on these observations, this study proposes a measurable framework to understand how different smart contract programming languages vary in terms of security and efficiency, emphasizing the role of language design and secure coding practices in shaping contract development.

Keywords: Blockchain security, comparative benchmarking, secure coding, smart contracts.

Öz

Akıllı sözleşmeler, blok zinciri ekosistemlerinin temel bir bileşeni haline gelmiş olup, güvenilirlikleri büyük ölçüde yazıldıkları programlama dilleri tarafından şekillendirilmektedir. Önceki çalışmalar zafiyetleri sınıflandırmış olsa da, dil tasarımı ve güvenli kodlama uygulamalarının performans ve dayanıklılık üzerindeki etkilerini nicel olarak inceleyen araştırmalar daha sınırlıdır. Bu boşluğu gidermek amacıyla, bu çalışmada kontrollü koşullar altında Solidity ve Vyper dillerini deneysel olarak karşılaştırılmakta ve bu deneyler Rust ve Move dillerine ilişkin literatür temelli bir değerlendirme ile tamamlanmaktadır. Test senaryoları; dağıtım, para yatırma, para çekme, aritmetik taşma, yeniden giriş (reentrancy) ve işlem kaynağı kötüye kullanımı gibi durumları içermektedir. Solidity ve Vyper'ın hem zafiyetli hem de güvenli sürümleri için gaz tüketimi, dağıtım boyutu ve çalışma zamanı yürütme süresi gibi metrikler toplanmıştır. Sonuçlar, iki dil arasında dağıtım maliyetlerinin önemli ölçüde farklılaştığını (Solidity ≈ 177k gas, Vyper ≈ 135k gas, yaklaşık %24 daha düşük) gösterirken, çalışma zamanı performansının karma olduğunu ortaya koymaktadır: deposit çağrıları neredeyse aynıdır (Δ ≈ 0.02 ms), buna karşın withdraw çağrısında belirgin bir fark vardır (Δ ≈ 4.97 ms) ve Vyper lehinedir; yine de bu çağrı düzeyi farkları, dağıtım zamanındaki daha büyük farka kıyasla küçüktür. Ayrıca, açık aritmetik kontrolleri ve Checks–Effects–Interactions modelinin uygulanması gibi güvenli kodlama uygulamaları, kritik zafiyetleri ortadan kaldırırken toplam yürütme maliyetine %1'den daha az ek yük getirmektedir. Rust ve Move her ne kadar bu çalışmada literatür temelli olarak ele alınmış olsa da, güvenlik garantilerini doğrudan dile entegre eden alternatif yaklaşımları örneklemektedir. Bu gözlemlerden hareketle, bu çalışma; farklı akıllı sözleşme programlama dillerinin güvenlik ve verimlilik açısından nasıl farklılaştığını anlamaya yönelik ölçülebilir bir çerçeve önermekte ve dil tasarımı ile güvenli kodlama uygulamalarının sözleşme geliştirme sürecini nasıl şekillendirdiğini vurgulamaktadır.

Anahtar Kelimeler: Akıllı sözleşmeler, blok zincir güvenliği, güvenli kodlama, karşılaştırmalı performans ölçümü.

[1] Istanbul University-Cerrahpaşa, Faculty of Engineering, Department of Computer Engineering, Istanbul, Türkiye. hozaslan@ogr.iuc.edu.tr.
[2] Istanbul University-Cerrahpaşa, Faculty of Engineering, Department of Computer Engineering, Istanbul, Türkiye. safak.odabasi@iuc.edu.tr.

1. INTRODUCTION

The concept of smart contracts was first introduced by Nick Szabo in 1996, who defined them as "a set of digitally defined commitments and protocols by which the parties fulfill these commitments" (Szabo, 1996). Smart contracts were initially envisioned to automate agreements by ensuring execution when predefined conditions were met (Buterin, 2014). Despite this early vision, the lack of secure and stable computational environments hindered practical implementation, and development remained stagnant for years. The emergence of blockchain technology, which was first realized in the Bitcoin protocol through distributed ledger technology (Nakamoto, 2008), provided the infrastructure necessary to revive this idea. As blockchain systems are fundamentally designed to establish trust between parties, they simultaneously facilitate the advancement of smart contracts. Ethereum marked a significant milestone by introducing a programmable environment that bound contracts directly to blockchain execution, making smart contracts and blockchain technology inseparable (Wood, 2014).

Smart contracts differ from other computational paradigms due to unique features such as automated execution, multi-party verification, and decentralized propagation. Their applications extend from simple bilateral agreements to complex transactional logics and compliance checks, thereby enabling trust-based interactions across a wide range of industries (Zheng et al., 2020). This evolution has transformed Szabo's original architecture into a more sophisticated framework, further driven by the proliferation of blockchain platforms and domain-specific languages tailored to secure distributed execution (Atzei, Bartoletti, & Cimoli, 2017).

Choice of programming language has become central to smart contract development, as it directly affects readability, security, and performance. Ethereum has positioned Solidity as the most widely adopted contract-oriented language, supported by alternatives such as Vyper and LLL, while general-purpose languages like Java, Python, Go, and C++ have also been applied in certain contexts (Dannen, 2017). However, the increasing complexity of smart contracts has introduced new risks, including susceptibility to cyberattacks, highlighting the need for safer language designs and verification methods (Besbas et al., 2024). Recent research emphasizes that vulnerabilities in languages such as Solidity remain a key factor behind high-profile incidents, including reentrancy attacks and unchecked arithmetic errors (Durieux et al., 2020). Emerging alternatives, such as Vyper, prioritize simplicity and auditability by restricting unsafe features, while languages like Rust and Move embed stronger semantic guarantees into their execution models (Smolka et al., 2023).

Despite these advances, performance bottlenecks persist. The efficiency of smart contracts is strongly tied to the underlying blockchain and the programming language semantics, particularly in terms of gas consumption and runtime execution. Recent studies increasingly focus on comparative benchmarking to assess these trade-offs, revealing that secure coding idioms can often be adopted with negligible overhead while substantially reducing the attack surface (Zou et al., 2021; Bartoletti, 2024). These findings underscore the need for systematic evaluations of language-level choices to guide developers and researchers.

In this study, we build on this line of inquiry by empirically comparing Solidity and Vyper under controlled conditions and complementing this analysis with a literature-based evaluation of Rust and Move. By focusing on the intersection of security, performance, and optimization, this work aims to clarify the practical implications of programming language choice in smart contract development and to contribute evidence-based insights for both academia and practice.

The remainder of this paper is organized as follows: Section 2 reviews related work on smart contract programming languages, with emphasis on their security, performance, and optimization aspects. Section 3 presents the methodology, detailing the experimental setup for Solidity and Vyper as well as the literature-based analysis of Rust and Move. Section 4 discusses the results, highlighting comparative findings and their implications. Finally, Section 5 concludes the study and outlines directions for future research.

2. RELATED WORK

Research on smart contract programming languages has converged on the view that language design decisively shapes both security and performance characteristics of decentralized applications. Foundational systematizations exposed how Solidity's semantics enabled recurring vulnerability classes like reentrancy, unchecked arithmetic, thereby motivating safer language designs and verification-centric workflows (Atzei, Bartoletti, & Cimoli, 2017). Subsequent analyses questioned whether Solidity's permissive features are "solid enough," arguing that auditability degrades when unrestricted constructs are combined with complex application logic (Crafa, Di Pirro, & Zucca, 2019). These early findings seeded a trajectory toward domain-specific languages and stricter semantics intended to reduce the error surface at the source (Luu et al., 2016; Parizi, Amritraj, & Dehghantanha, 2018).

Broader surveys mapped the language and tooling landscape from multiple vantage points. Varela-Vaca and Quintero (2021) provided a multivocal mapping of smart contract languages, highlighting tensions between expressiveness and analyzability; Vacca et al. (2021) reviewed techniques and tools across the lifecycle, emphasizing persistent gaps in formal verification support; and Hewa et al. (2021) synthesized technical aspects and open problems in blockchain-based contracts. Hu et al. (2021) categorized paradigms, tools, and systems for contract construction and execution, while Zheng et al. (2020) emphasized challenges and platform-level trade-offs, further reinforcing the role of programming choices in shaping both security and efficiency. Together, these works underscore that language choice is not merely a matter of developer ergonomics but a first-order determinant of resilience and maintainability.

Security-focused studies have examined analysis pipelines and mitigation strategies in greater detail. A systematic review of Ethereum analysis tools categorized static, dynamic, and hybrid approaches and showed that effectiveness is tightly coupled to Solidity's design (Kushwaha et al., 2022). Ivanov et al. (2023) proposed a taxonomy of 133 mitigation techniques across five orthogonal dimensions, arguing that safeguards at the language level, like restricted control flow, stricter typing, are among the most sustainable defenses in practice. Complementing these surveys, formal methods lines of work examined how verification compensates for semantic risks. Garfatta et al.

(2021) surveyed formal verification for Solidity. Wesley et al. (2022) demonstrated that communication abstraction and abstract interpretation (SmartACE) increase coverage for Solidity contracts. Meanwhile, ecosystems with stricter base languages reported encouraging verification results. The Helmholtz verifier brings refinement types to Michelson (Nishida et al., 2022), and recent static analyses for Michelson (Olivieri et al., 2024) strengthen the case that restricted semantics ease sound analysis (Nishida et al., 2022; Olivieri et al., 2024).

Performance and optimization threads show that metering, execution models, and concurrency interact with language semantics. Chen et al. (2017) proposed adaptive gas-cost mechanisms to blunt under-priced denial-of-service vectors at the Ethereum Virtual Machine (EVM) level. Perez and Livshits (2019) revealed inconsistencies between gas pricing and actual resource consumption, exposing attacks on resource metering in the EVM and illustrating how policy choices at the VM level leak into cost models defined at the language level. Bartoletti, Galletta, and Murgia (2021) formalized transaction parallelism, linking concurrency models to throughput and highlighting how language/VM interaction constrains scalability. Randomness, a recurring requirement in on-chain logic, has also been identified as a significant challenge. Qian et al. (2023) established a taxonomy and detection strategy for insecure random number practices in Ethereum, demonstrating how ostensibly simple API-level choices can cascade into security/performance trade-offs (Qian et al., 2023).

Comparative evaluations sharpen the security-performance-optimization picture across languages. One of the earliest empirical comparisons argued that domain-specific designs can improve usability and reduce vulnerability density relative to general-purpose patterns typical of Solidity (Parizi et al., 2018). Zou et al. (2021) synthesized development challenges and opportunities, noting persistent tensions between developer convenience and enforceable safety properties. Developer-centric platform comparisons further observed that Rust and Move embed stronger semantic guarantees that shift failure modes earlier in the lifecycle (Voloder & Di Angelo, 2023).

Work in Rust-based and Move-based ecosystems supplies concrete evidence about toolchains and effects on vulnerability profiles. On Solana, VRust performs automated vulnerability detection for Rust smart contracts (Cui et al., 2022), and subsequent fuzzing campaigns uncovered additional runtime weaknesses (Smolka et al., 2023). On Aptos (Move), industrial experiences describe integrating the Move Prover into framework development, indicating that proof-oriented workflows can be embedded into CI/CD without prohibitive friction (Park et al., 2024). These studies collectively indicate that language-level semantics and mature analysis tooling jointly reduce exploitability at scale (Cui et al., 2022; Smolka et al., 2023; Park et al., 2024).

Bartoletti et al. (2025) presented a large-scale comparative analysis covering Solidity, Vyper, Rust, Move, and others, assessing usability, safety, and programming style with a common benchmark suite and reporting systematic trade-offs between expressiveness and verifiability (Bartoletti et al., 2025). In parallel, an open benchmark used to compare SolCMC and Certora along completeness/soundness axes, precisely the sort of infrastructure needed to quantify the cost of secure idioms and the coverage of verification pipelines, was articulated in (Bartoletti et al., 2024).

Literature review findings indicate that programming language design has a decisive influence on the security, performance, and verifiability of smart contracts. Solidity's expressiveness enables rapid development but also correlates with recurring vulnerabilities and higher verification costs, while Vyper's restrictive syntax reduces potential attack surfaces at the expense of flexibility. Rust and Move represent a different trajectory, embedding security guarantees directly into language semantics and thereby shifting risk mitigation to earlier phases of the development cycle. Furthermore, performance-oriented studies highlight that gas metering, randomness handling, and concurrency models are tightly coupled with language and virtual machine semantics, whereas formal verification research demonstrates how rigorous analysis can transform these challenges into enforceable guarantees. Building on these insights, this study contributes by empirically comparing Solidity and Vyper under controlled conditions, complemented with a literature-based evaluation of Rust and Move, to systematically assess how language-level choices shape security, efficiency, and optimization in smart contract development.

3. METHODOLOGY

This study adopts a comparative approach to examine four smart contract programming languages: Solidity, Vyper, Rust, and Move**.** The emphasis is placed on the interplay between language design, security guarantees, and execution performance**.** Solidity and Vyper were benchmarked empirically in a controlled environment, whereas Rust and Move were evaluated through a literature-based analysis due to their fundamentally different execution models and platform dependencies. This dual approach ensures reproducibility for the Ethereum-based languages while also providing breadth across non-EVM ecosystems.

3.1. Language Overview

3.1.1. Solidity / Ethereum

First introduced in 2014, Solidity remains the primary high-level programming language for blockchains that implement the Ethereum Virtual Machine (EVM), including Ethereum, Avalanche C-Chain, and Hedera. Before deployment, Solidity code is compiled into EVM bytecode for execution by network nodes (Solidity Team, 2024).

Solidity is a statically typed language, requiring explicit definition of variable and method types. It supports subtype and ad-hoc polymorphism, and while some conversions are type-safe, unsafe casts must be explicitly declared, often resulting in compilation errors if misapplied. Solidity incorporates several potentially unsafe features, such as external function calls and inline assembly, that demand careful handling. The language also supports multiple inheritance, modularization via libraries and interfaces, and the definition of multiple contracts within a single file. Its syntax, which resembles JavaScript and other procedural languages, contributes to its popularity among developers, particularly beginners. However, design peculiarities, combined with the rising complexity of decentralized applications, introduce significant security risks, including reentrancy and unchecked arithmetic vulnerabilities.

3.1.2. Vyper / Ethereum

Vyper is a Python-inspired language developed to improve contract simplicity, security, and auditability. Unlike Solidity, Vyper intentionally restricts its feature set, omitting constructs such as modifiers, inheritance, inline assembly, operator overloading, recursion, unbounded loops, and floating-point arithmetic (Ethereum Foundation, 2017). These constraints enhance readability and reduce the risk of introducing subtle vulnerabilities, but they also increase the development effort needed for certain functionalities.

Vyper's design philosophy is grounded in reducing the attack surface and promoting human-auditable code. Although it is less mature and still under active development, Vyper has gained attention for fostering secure-by-design contracts where transparency is prioritized over expressiveness.

3.1.3. Rust / Solana

Rust, a systems programming language created by Mozilla Research, was adopted as the primary language for smart contract development on Solana (Matsakis & Klock, 2014). Similar to Solidity on Ethereum, Rust code must be compiled into bytecode for execution by Solana's runtime environment.

Rust is statically typed and enforces strong guarantees on memory safety through its ownership and borrowing model. This design detects errors such as null pointer dereferencing at compile time, preventing runtime crashes common in languages like C and C++. Unlike garbage-collected languages, Rust manages memory efficiently without additional runtime overhead. These properties make Rust highly suitable for blockchain environments where security and efficiency are paramount (Cui et al., 2022; Smolka et al., 2023).

3.1.4. Move / Aptos

Move is a smart contract programming language originally developed for the Libra/Diem project and later adopted by platforms such as Aptos, Sui, and Starcoin (Catalini et al., 2019). Influenced by Rust, Move introduces linear types, also referred to as must-move semantics, to enforce strict ownership and ensure that digital resources such as tokens cannot be duplicated or accidentally destroyed. This type system provides strong protection against asset mismanagement, addressing vulnerabilities that commonly arise in EVM-based languages.

While Move offers stronger safety guarantees, its practical implementation differs across platforms. Each blockchain adopting Move tailors the language by building a dedicated framework, providing a standard library, and adapting the core semantics to its own architecture. Aptos, for example, extends Move with integrated tooling such as the Move Prover, which enables formal verification workflows during development and deployment (Park et al., 2024; Voloder & Di Angelo, 2023).

3.2. Language Features and Security

Programming language design is a crucial factor in smart contract development, as it directly shapes readability, safety, and execution correctness. While platform choice defines the underlying consensus and execution layer, the language determines how effectively developers can model logic without exposing unnecessary vulnerabilities.

Bitcoin's early stack-based scripting language, while effective for basic transaction validation, is intentionally non–Turing complete and lacks advanced control structures, recursion, or loops, limiting its expressiveness (Nakamoto, 2008). This simplicity reduces complexity but restricts contract functionality and introduces inefficiencies in transaction verification.

Ethereum adopted a more versatile approach, introducing the Ethereum Virtual Machine (EVM) and high-level languages such as Solidity and Vyper. Solidity, the dominant choice, offers extensive flexibility but includes features, like external calls, inline assembly, multiple inheritance, that expand the attack surface (Wood, 2014). In contrast, Vyper was designed to minimize unsafe constructs, prioritizing simplicity and auditability over expressiveness (Ethereum Foundation, 2017; Kaleem, Mavridou, & Laszka, 2020). Beyond the Ethereum ecosystem, Rust and Move represent languages embedding stricter semantic guarantees at the type system level. Rust enforces memory safety through ownership and lifetime checks (Rust Foundation, 2023), while Move introduces linear types to prevent unauthorized duplication or destruction of resources (Blackshear et al., 2019).

Collectively, these languages illustrate different trade-offs: Solidity maximizes developer convenience but requires stricter coding discipline; Vyper restricts expressiveness to improve verifiability; Rust and Move integrate safety into the language semantics, aiming to prevent vulnerabilities by construction.

3.3. Execution Models and Performance

Execution models define how contract code is compiled and run, and they affect both scalability and runtime efficiency.

3.3.1. Imperative execution in Solidity

Solidity follows an imperative paradigm that facilitates arbitrary state changes at runtime. While this flexibility improves expressiveness, it also increases complexity and vulnerability incidence, as illustrated by recurring reentrancy and arithmetic flaws. Static analysis tools have repeatedly shown that managing data dependencies and state transitions requires careful handling to ensure correctness.

3.3.2. Functional determinism in Vyper and formal languages

Languages that restrict state transitions—such as Vyper or functional/formally verified languages—treat contract execution as a deterministic mapping between states. This model simplifies reasoning, supports formal verification, and mitigates classes of

runtime errors. The trade-off is higher upfront verification costs, but the benefit is improved predictability and reduced attack surfaces.

3.3.3. VM-level performance considerations

Execution efficiency is commonly measured in terms of gas consumption. Gas usage depends not only on compiler output but also on the language semantics. Vyper's restricted syntax can reduce gas overhead in some cases, while Solidity's feature-richness may increase costs for complex constructs. Rust and Move, executed on Solana and Aptos, respectively, inherit optimizations from their host VMs but enforce stricter correctness at compile time.

3.4. Benchmark Design and Workloads

To enable a systematic comparison between Solidity and Vyper, a benchmark suite was designed that isolates representative vulnerability patterns and evaluates their performance and security implications under controlled conditions. The selection of workloads was guided by their recurrence in high-profile incidents, their prominence in the literature on smart contract security, and their coverage of language-level semantics. Each workload was implemented in two variants: a vulnerable baseline and a secure counterpart that incorporates widely recommended mitigation techniques. This design allows for a direct measurement of both the raw performance characteristics of each language and the overhead, if any, introduced by secure coding idioms.

The workloads were defined as follows:

- **Deployment Cost**: Baseline measurement of compiler output size and gas required to deploy contracts. This captures differences in compiler layout, metadata handling, and language abstractions.
- **Deposit and Withdrawal Operations:** Standard state-updating functions that reflect the majority of financial logic in decentralized applications. These measure routine execution costs under both languages.
- **Reentrancy Pattern**: Modeled using a simple victim contract with a withdrawAll function. The secure variant applies the Checks–Effects–Interactions (CEI) pattern to remove the reentrancy window.
- **Transaction Origin Misuse:** Implementation of an authorization check using tx.origin in the vulnerable variant and a secure equivalent using msg.sender. This pattern evaluates semantic safety in identity management.
- **Arithmetic Overflow**: Contracts that perform unchecked arithmetic operations compared with secure counterparts using explicit overflow checks.

Metrics collected include gas consumption (as recorded by EIP-1559 receipts), deployment size (in bytes), and execution time (wall-clock duration for deposit and withdrawal calls). Each workload was executed 20 times under deterministic local conditions, and the mean values were reported to account for run-to-run variation.

The experimental design ensures that findings reflect language-level trade-offs rather than network-level noise. By isolating recurring vulnerability classes and pairing them

with their mitigated forms, the benchmarks make visible the cost–benefit profile of security-first design in Solidity and Vyper, providing a reliable baseline for comparison with the literature-based evaluation of Rust and Move.

The benchmarking environment was configured to ensure reproducibility and determinism. Table 1 summarizes the setup.

Table 1. Experimental Environment and Configuration Parameters

Parameter	**Specification**
Hardware	Dockerized; host-agnostic. Host: Intel i5-13400F (10 cores), 32 GB RAM, 1 TB SSD
Operating System	Ubuntu Linux 22.04 LTS (Docker base); Host optional: Windows 10
Containerization	Docker image from ubuntu:22.04 with Foundry and Vyper installed
Blockchain Testnet	Local Ethereum testnet via Anvil (chainId 31337)
EVM Toolchain	Foundry (forge/cast/anvil), installed via foundryup
Solidity Compiler	solc 0.8.24, optimizer enabled (200 runs); EVM target: Cancun
Vyper Compiler	vyper 0.4.3
Python Tooling	web3.py 6.20.3
Benchmark Workloads	Deploy; deposit/withdraw; reentrancy; tx.origin misuse; arithmetic overflow
Metrics Collected	Gas (EIP-1559 receipts), deployment size, runtime execution time
Benchmark Duration	≈1–2 minutes end-to-end (build + scripts) on a laptop
Repetition	Each workload repeated 20×; mean, min, max reported
Validation	Deterministic chain, cross-checked safe vs vulnerable and Solidity vs Vyper

3.5. Validity and Limitations

The experiments are subject to several limitations. Microbenchmarks isolate vulnerability patterns but do not capture the full complexity of production contracts. Results were obtained in a controlled local environment, which ensures reproducibility but does not fully account for network-level effects such as gas fluctuations and consensus latency. Findings also depend on specific compiler versions (Solc 0.8.24, Vyper 0.4.3) that may evolve. Rust and Move were not empirically benchmarked due to incompatibility with the EVM; instead, their evaluation was drawn from literature reporting ownership/resource semantics and security models. Despite these constraints, the **r**elative trends observed are consistent and generalizable, providing actionable insights into the trade-offs of language design.

4. COMPARATIVE ANALYSIS AND RESULTS

This section compares Solidity, Vyper, Rust, and Move with a focus on security, performance, and developer experience. Solidity and Vyper are benchmarked through controlled experiments on the Ethereum Virtual Machine (EVM), while Rust and Move are assessed through literature-based analysis due to their distinct execution models. Overall, the comparison shows that choices in language design, ranging from Solidity's flexibility and Vyper's emphasis on auditability to Rust's ownership guarantees and Move's resource semantics, directly influence the reliability and efficiency of smart contracts.

4.1. Solidity vs. Vyper

As the two most prominent languages targeting the Ethereum Virtual Machine (EVM), Solidity and Vyper represent contrasting design philosophies. Solidity emphasizes flexibility and feature richness, while Vyper deliberately restricts features to enhance auditability and reduce vulnerability incidence. The empirical benchmarks conducted in this study show that these design choices result in meaningful differences in deployment cost, execution time, and contract size, though runtime behavior remains largely comparable.

4.1.1. Gas consumption

Table 2 summarizes baseline gas costs for core operations across the two languages. Deployment is more expensive in Solidity (176,959 gas) compared to Vyper (134,618 gas), a difference of +42,341 gas. However, runtime operations such as deposit and withdraw exhibit near identical costs, differing by fewer than 150 gas on average.

Table 2. Average Gas Consumption Over 20 Runs

Operation	Solidity	Vyper	Solidity–Vyper (difference)
deploy	176,959	134,618	+42,341
deposit	43,499	43,357	+142
withdraw	28,509	28,399	+110

Figure 1 shows the comparison results regarding gas consumption.

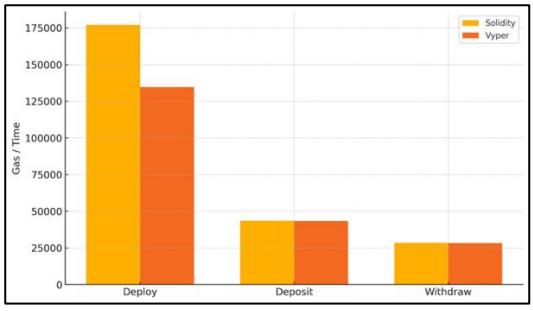

Figure 1. Gas Consumption Comparison Between Solidity and Vyper

4.1.2. Execution time

Table 3 shows execution times measured in conds. Deployment again exhibits the largest gap, with Solidity averaging 2.85 ms versus Vyper's 2.30 ms ($\Delta \approx 0.55$ ms). For runtime calls, the picture is mixed: deposit is nearly identical (2.38 ms vs 2.36 ms, $\Delta \approx 0.02$ ms), whereas withdraw shows a noticeable difference (7.40 ms vs 2.43 ms, $\Delta \approx 4.97$ ms). We attribute this disparity to call-path and state-access patterns specific to the withdraw logic; importantly, the call-level gap is still modest compared to the deployment-time difference.

Table 3. Average Execution Time Over 20 Runs

Operation	Solidity	Vyper	Solidity–Vyper (difference)
deploy	2.85	2.30	+0.55
deposit	2.38	2.36	+0.02
withdraw	7.40	2.43	+4.97

Figure 2 shows the comparison results regarding execution time in milliseconds (ms).

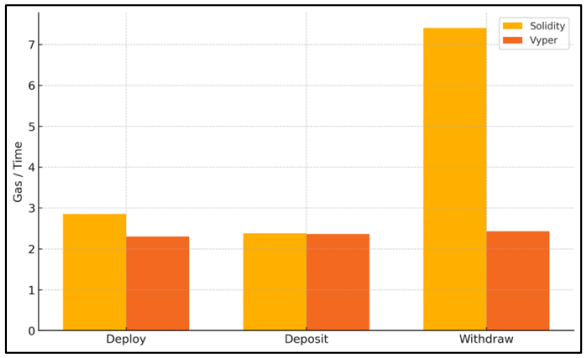

Figure 2. Execution Time Comparison Between Solidity and Vyper (ms)

4.1.3. Deployment size

Differences are most pronounced in deployment size (Table 4). Contracts compiled with Solidity averaged 572 bytes, compared to 374 bytes for Vyper—a reduction of 198 bytes. This reflects Solidity's more complex compiler metadata and richer feature set.

Table 4. Deployment Size Comparison Between Solidity and Vyper

Operation	Solidity	Vyper	Solidity–Vyper (difference)
deploy_size	572	374	+198

4.1.4. Mitigation overheads

Tables 5–7 examine the gas overhead of applying secure coding practices. For arithmetic overflow in Table 5, safe variants incurred higher deployment cost (11,246 gas) but negligible differences at runtime.

Table 5. Gas Consumption for Overflow Mitigation
(Lower Values Indicate Better Performance)

Operation	Vulnerable	Safe	Vulnerable − Safe (difference)
deploy	91,503	102,749	−11,246
deposit	43,506	43,602	−96
withdraw	0	0	0

Figure 3 illustrates the impact of arithmetic overflow mitigation on gas consumption, showing that secure implementations incur only marginal overhead during deployment and none during execution.

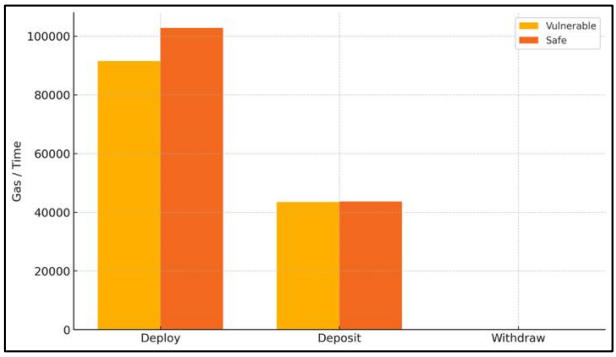

Figure 3. Gas Consumption Under Arithmetic Overflow Mitigation

Reentrancy mitigation in Table 6 introduced virtually no overhead, with only a −5 gas difference in withdrawal calls.

Table 6. Gas Consumption for Reentrancy Mitigation
(Lower Values Indicate Better Performance)

Operation	Vulnerable	Safe	Vulnerable − Safe (difference)
deploy	176,959	176,959	0
deposit	43,499	43,499	0
withdraw	28,509	28,514	−5

Figure 4 presents the gas usage comparison for reentrancy mitigation, demonstrating that applying the Checks–Effects–Interactions pattern introduces virtually no additional cost.

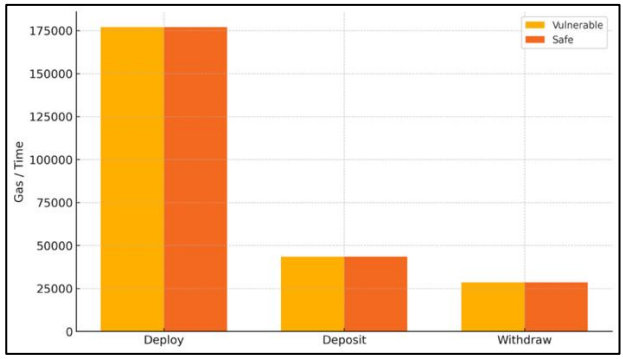

Figure 4. Gas Consumption Under Reentrancy Mitigation

Transaction origin checks in Table 7 showed a moderate decrease in deployment cost (-6,476 gas) with no effect on runtime performance.

Table 7. Gas Consumption for Tx.Origin Mitigation
(Lower Values Indicate Better Performance)

Operation	**Vulnerable**	**Safe**	**Vulnerable − Safe (difference)**
deploy	145,933	139,457	-6,476
deposit	0	0	0
withdraw	23,513	23,513	0

Figure 5 shows the effect of replacing tx.origin with msg.sender for authentication, where the secure variant leads to a moderate decrease in deployment cost.

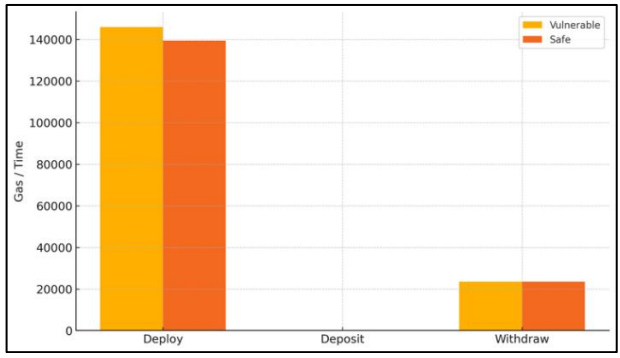

Figure 5. Gas Consumption Under Tx.Origin Misuse Mitigation

Across these benchmarks, runtime costs are operation-dependent—deposit is nearly identical (Δ≈0.02 ms), whereas withdraw shows a noticeable gap (Δ≈4.97 ms)—with larger and more consistent discrepancies observed during deployment. Crucially, secure coding conventions such as CEI, msg.sender verification, and explicit arithmetic checks impose negligible overhead while eliminating entire vulnerability classes. This

finding underscores that security-first practices can be adopted at minimal performance cost, independent of the chosen EVM language.

4.2. Rust vs. Move

4.2.1. Type systems and built-in safety guarantees

Rust and Move both enforce strong static guarantees, but they do so with different aims and mechanisms. Rust's ownership–borrowing discipline and lifetime tracking prevent use-after-free, data races, and many classes of memory-unsafety at compile time without a garbage collector (Rust Foundation, 2023). This model is especially impactful for Solana programs written in Rust, where parallel runtime scheduling (Sealevel) benefits from aliasing constraints made explicit by the type system. Move, by contrast, was designed for digital assets: its linear resource types (must-move semantics) make creation, movement, and destruction of on-chain resources explicit and verifiable in the type system, eliminating accidental duplication or loss of tokens by construction (Blackshear et al., 2019; Aptos Labs, 2025). In short, Rust primarily eliminates memory errors; Move primarily eliminates asset misuse.

4.2.2. Resource/asset model and state discipline

Rust exposes account state through program-defined layouts, like Solana accounts, and relies on careful API and account-access discipline to avoid logical asset bugs. Libraries and conventions help, but asset integrity is not a first-class type of property. Move elevates assets to first-class resources with linearity, so every mint/burn/transfer is statically tracked; illegal states are unrepresentable, and transfer invariants can be encoded as types and specifications (Blackshear et al., 2019; Park et al., 2024). This materially narrows the vulnerability surface in token and DeFi logic compared with general-purpose languages.

4.2.3. Execution model, concurrency, and scheduling

On Solana, Rust programs target a runtime that supports transaction-level parallelism when account read/write sets are disjoint, allowing high throughput under Sealevel's scheduler. Correctly declaring account access is essential; otherwise, conservative locks reduce parallelism. Move runtimes (e.g., Aptos, Sui) also exploit parallel execution by analyzing read/write sets inferred from Move's resource semantics, enabling deterministic parallelism with fewer ad-hoc locks (Aptos Labs, 2025). Thus, both ecosystems unlock parallelism, but Move's static resource model provides more structured inputs to the scheduler.

4.2.4. Call model and reentrancy risk

Rust does not intrinsically forbid reentrancy; risk depends on the program's call patterns and external interfaces (e.g., CPI, cross-program invocation, on Solana) and must be mitigated by design rules and analysis tools. In Move, reentrancy risks are reduced by the module system, explicit entry functions, and linear resource discipline, which constrain state transitions and make effectful calls explicit; this structure simplifies reasoning about Checks–Effects–Interactions-style patterns and can be

mechanically verified (Blackshear et al., 2019; Park et al., 2024). While neither model magically removes all logic bugs, Move shifts more correctness obligations into the type and module system.

4.2.5. Verification toolchains and analysis coverage

Rust benefits from a mature general-purpose ecosystem (property testing, fuzzing, symbolic execution), and domain tools have emerged for Solana contracts: VRust automates vulnerability detection for Rust-based programs, and large-scale fuzzing campaigns have uncovered runtime weaknesses tied to account handling and CPI misuse (Cui et al., 2022; Smolka et al., 2023). Move brings the Move Prover, which leverages the resource semantics to discharge functional correctness properties and to enforce module-level invariants in CI/CD (Park et al., 2024). Empirically, Rust's breadth of tools aids performance-oriented code with memory safety guarantees, whereas Move's prover focuses on the semantic safety of assets and protocol invariants.

4.2.6. Performance and optimization implications

Rust's zero-cost abstractions and ahead-of-time compilation (LLVM) yield predictable, low-overhead code, which aligns with Solana's parallel runtime for high throughput. The ownership model also helps the optimizer by making aliasing explicit. Move's design trades a modest amount of checking/verification overhead for stronger correctness guarantees: resource linearity adds static discipline that simplifies bytecode analysis and can enable aggressive but safe parallel execution and conflict detection at the transaction layer (Aptos Labs, 2025). In practice, Rust maximizes raw performance; Move maximizes asset-safety-aware performance.

4.2.7. Developer ergonomics and learning curve

Rust's learning curve (borrowing/lifetimes) is significant for developers coming from garbage-collected languages, but the payoff is strong: once mastered, patterns are reusable beyond blockchain. Move's conceptual model is narrower but closer to on-chain needs; developers reason in terms of resources rather than plain structs, which matches token and DeFi logic. However, Move ecosystems (Aptos/Sui) remain younger than the Rust+Solana stack; library maturity and documentation continue to improve, particularly around formal specs and prover-friendly patterns (Aptos Labs, 2025; Park et al., 2024).

4.2.8. Portability and ecosystem fragmentation

Rust is broadly portable as a language, but Solana-specific program models (accounts/CPI) limit direct portability across chains. Move aspires to multi-chain usage, yet each chain ships a distinct framework and standard library; minor semantic and framework differences mean that "write once, run anywhere" is not fully realized, and careful adaptation is required (Aptos Labs, 2025). Both ecosystems, therefore face platform-binding realities: Rust to Solana's runtime; Move to each chain's framework.

4.2.9. Synthesis relative to Solidity/Vyper

Compared with EVM languages, Rust and Move push more safety into the language/bytecode layer: Rust by eliminating memory-safety classes and clarifying aliasing for parallelism; Move by making assets linear resources and enabling first-class specification/verification. Findings reported for Solana (Cui et al., 2022; Smolka et al., 2023) and Aptos (Park et al., 2024) indicate that these semantics do reduce exploitability in the classes they target, with performance characteristics shaped primarily by their runtimes rather than by ad-hoc language checks. For studies like ours that measured the runtime cost of secure idioms on the EVM, the literature on Rust/Move suggests that many such idioms can be made structural (type-system-enforced), potentially lowering the developer cost of security while preserving throughput under parallel scheduling.

4.3. Cross-Language Security and Performance Metrics

When evaluated across common vulnerability patterns, reentrancy, transaction origin misuse, and arithmetic overflow, the empirical benchmarks demonstrate that Solidity and Vyper perform similarly in runtime terms, with only minor gas overheads introduced by secure coding practices. For instance, CEI mitigations and explicit arithmetic checks eliminate entire classes of vulnerabilities with negligible additional cost, suggesting that secure design does not inherently trade off against efficiency.
Rust and Move, based on literature-driven findings, address these same patterns through semantics rather than idioms. Rust's ownership model prevents unauthorized reentrancy by enforcing strict control over mutable references, while Move's resource linearity ensures that transaction origin misuse and asset duplication cannot occur by construction. Arithmetic safety is similarly guaranteed through the use of strict type systems and compiler-enforced constraints. In short, while EVM-based languages require developers to adopt best practices to mitigate vulnerabilities, Rust and Move integrate these protections directly into the language design.

4.4. Synthesis and Insights

The cross-language synthesis highlights the broader implications of programming language design for blockchain ecosystems. Solidity's flexibility continues to drive its adoption, but correlates with higher vulnerability incidence and reliance on external analysis tools. Vyper demonstrates that restricting unsafe features can meaningfully reduce attack surfaces with minimal performance penalties. Rust and Move extend these insights by showing how semantics such as ownership and resource linearity can enforce safety by construction, albeit within ecosystems that are still maturing. The key contrasts are summarized in Table 8.

Table 8. Summary of Main Insights

Aspect	Solidity	Vyper / Functional Languages
Security	Higher vulnerability incidence	Lower incidence due to reduced complexity

Execution Model	Imperative, flexible but complex	Deterministic, state transitions as functions
Tooling Availability	Mature ecosystem (static, dynamic tools)	Emerging tools; intensive formal verification
Optimization Trade-Offs	Emphasis on gas efficiency at an increased risk	Slight overhead for improved security and verifiability
Future Development Focus	Enhanced runtime protection and analysis tools	Integration of formal methods with conventional workflows

Collectively, these findings demonstrate that security-first language designs, whether enforced through coding idioms (Vyper, Solidity) or embedded semantics (Rust, Move), achieve resilience at surprisingly low cost. For blockchain platforms seeking to balance usability with safety, the evidence suggests that language-level choices are as consequential as consensus mechanisms or scalability models in shaping long-term system trustworthiness.

5. CONCLUSION

This study presented a comparative analysis of four smart contract programming languages, Solidity, Vyper, Rust, and Move, from the perspective of security, performance, and optimization. Solidity and Vyper were benchmarked empirically under controlled conditions, while Rust and Move were analyzed based on the literature, reflecting their distinct execution models and design philosophies. Across languages, results show that programming semantics are not neutral; they actively shape vulnerability incidence, runtime performance, and the cost of mitigation strategies. Solidity's flexibility facilitates developer expressiveness but introduces higher risks, whereas Vyper reduces vulnerability density through restrictive semantics and improved auditability, though at the expense of some deployment overhead. Rust and Move exemplify a different design trajectory, embedding ownership and resource semantics that prevent critical classes of vulnerabilities at the language level and enabling integration with formal verification tooling.

The empirical evaluation of Solidity and Vyper is subject to several constraints. Benchmarks were performed using microcontracts in a deterministic local testnet (Anvil), which isolates representative vulnerability patterns but does not capture the full complexity of production deployments. Compiler versions (solc 0.8.24 and vyper 0.4.3) and toolchain configurations may also influence absolute performance values, though the relative trends between languages and mitigation strategies are expected to hold across versions. These limitations delineate the scope of the findings but do not undermine their validity; rather, they suggest avenues for future research, including extending benchmarks to more complex real-world contracts, incorporating larger workloads, and evaluating the impact of compiler and toolchain evolution.

In conclusion, the findings show that security-first idioms, whether enforced by language restrictions in Vyper or by design patterns in Solidity, substantially reduce vulnerabilities with negligible performance overhead, while acknowledging operation-dependent nuances in runtime results. Rust and Move further highlight how embedding security guarantees directly into the type and resource models offers a promising

alternative to post-hoc verification. Future work should therefore expand cross-language benchmarks and integrate richer application scenarios, enabling a more holistic understanding of the trade-offs between expressiveness, assurance, and efficiency in smart contract ecosystems.

Contribution of The Authors
The authors declared that they contributed equally to this paper.

Conflict of Interest
The authors declare that there is no conflict of interest.

Statement of Research and Publication Ethics
Research rules and publication ethics were followed in the study

REFERENCES

Aptos Labs. (2025). Move - A Web3 Language and Runtime. Retrieved September 10, 2025, from https://aptos.dev/network/blockchain/move.

Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A survey of attacks on Ethereum smart contracts (SoK). *Principles of Security and Trust (POST), LNCS, 10204*, 164–186. Springer.

Bartoletti, M., Galletta, L., & Murgia, M. (2021). A theory of transaction parallelism in blockchains. *Logical Methods in Computer Science, 17*(4), 1–41.

Bartoletti, M., Fioravanti, F., Matricardi, G., Pettinau, R., & Sainas, F. (2024). Towards benchmarking of Solidity verification tools. *Workshop on Formal Methods for Blockchains (FMBC 2024), OASIcs*.

Bartoletti, M., Benetollo, L., Bugliesi, M., Crafa, S., Dal Sasso, G., Pettinau, R., Pinna, A., Piras, M., Rossi, S., Salis, S., Spanò, A., Tkachenko, V., Tonelli, R., & Zunino, R. (2025). Smart contract languages: A comparative analysis. *Future Generation Computer Systems, 164*, 107563.

Blackshear, S., Cheng, E., Dill, D. L., Gao, V., Maurer, B., Nowacki, T., ... & Zhou, R. (2019). Move: A language with programmable resources. Retrieved September 7, 2025, from https://developers.diem.com/papers/diem-move-a-language-with-programmable-resources/2019-06-18.pdf.

Buterin, V. (2014). *Ethereum white paper: A Next Generation Smart Contract & Decentralized Application Platform*. Retrieved September 10, 2025 from https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf

Besbas, A., Ailane, A., Kahloul, L., Slatnia, S., & Bourekkache, S. (2024, November). On the formal verification of smart contracts and blockchain: Challenges and future directions. In *2024 4th International Conference on Embedded & Distributed Systems (EDiS)*, 213-217.

Catalini, C., Gratry, O., Hou, J. M., Parasuraman, S., & Wernerfelt, N. (2019). The libra reserve. *Libra White Paper*.

Chen, T., Li, X., Wang, Y., Chen, J., Li, Z., Luo, X., Au, M. H., & Zhang, X. (2017). An adaptive gas cost mechanism for Ethereum to defend against under-priced DoS attacks. In *Information Security Practice and Experience (ISPEC 2017)*, 3–24. Springer.

Crafa, S., Di Pirro, M., & Zucca, E. (2019). Is Solidity solid enough? In *Financial Cryptography Workshops* (LNCS 11599, pp. 138–153). Springer.

Cui, S., Zhao, G., Gao, Y., Tavu, T., & Huang, J. (2022). VRust: Automated vulnerability detection for Solana smart contracts. In *Proceedings of the 2022 ACM CCS* (pp. 639–652). ACM.

Dannen, C. (2017). *Introducing Ethereum and Solidity: Foundations of cryptocurrency and blockchain programming for beginners.* Apress.

Durieux, T., Ferreira, J. F., Abreu, R., & Cruz, P. (2020, June). Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 530-541).

Ethereum Foundation. (2017). *Vyper: Pythonic smart contract language*. Retrieved September 5, 2025, from https://vyper.readthedocs.io

Garfatta, I., Klai, K., Gaaloul, W., & Graiet, M. (2021). A survey on formal verification for Solidity smart contracts. In *Proceedings of the Australasian Computer Science Week (ACSW '21)* (pp. 1–10). ACM.

Hu, B., Zhang, Z., Liu, J., Liu, Y., Yin, J., Lu, R., & Lin, X. (2021). A comprehensive survey on smart contract construction and execution: Paradigms, tools, and systems. *Patterns, 2*(2), 100179.

Hewa, T. M., Hu, Y., Liyanage, M., Kanhere, S. S., & Ylianttila, M. (2021). Survey on blockchain-based smart contracts: Technical aspects and future research. *IEEE Access, 9*, 87643–87662.

Ivanov, N., Li, C., Yan, Q., Sun, Z., Cao, Z., & Luo, X. (2023). Security threat mitigation for smart contracts: A comprehensive survey. *ACM Computing Surveys, 55*(14s), 326:1–326:37.

Kaleem, M., Mavridou, A., & Laszka, A. (2020, September). Vyper: A security comparison with Solidity based on common vulnerabilities. In *2020 2nd conference on blockchain research & applications for innovative networks and services (BRAINS)* (pp. 107-111). IEEE.

Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M., & Lee, H.-N. (2022). Ethereum smart contract analysis tools: A systematic review. *IEEE Access, 10*, 57037–57062.

Luu, L., Chu, D., Olickel, H., Saxena, P., & Hobor, A. (2016). Making smart contracts smarter. In *Proceedings of the 2016 ACM CCS* (pp. 254–269). ACM.

Matsakis, N. D., & Klock, F. S. (2014). The Rust language. *ACM SIGAda Ada Letters, 34*(3), 103–104.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. Retrieved September 7, 2025, from https://bitcoin.org/bitcoin.pdf.

Nishida, Y., Saito, H., Chen, R., Kawata, A., Furuse, J., Suenaga, K., & Igarashi, A. (2022). Helmholtz: A verifier for Tezos smart contracts based on refinement types. *New Generation Computing, 40*, 507–554.

Olivieri, L., Negrini, L., Arceri, V., Jensen, T., & Spoto, F. (2024). Design and implementation of static analyses for Tezos smart contracts. *Distributed Ledger Technologies: Research and Practice, 2*(2), 1–17.

Parizi, R. M., Amritraj, & Dehghantanha, A. (2018). Smart contract programming languages on blockchains: An empirical evaluation of usability and security. In *ICBC 2018*, 75–91. Springer.

Park, J., Zhang, T., Grieskamp, W., Xu, M., Di Giacomo, G., Chen, K., ... & Chen, R. (2024). Securing Aptos framework with formal verification. In *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*, 9-1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Perez, D., & Livshits, B. (2019). Broken metre: Attacking resource metering in EVM. *arXiv preprint arXiv:1909.07220*.

Qian, P., He, J., Lu, L., Wu, S., Lu, Z., Wu, L., Zhou, Y., & He, Q. (2023). Demystifying random number in Ethereum smart contracts: Taxonomy, vulnerability identification, and attack detection. *IEEE Transactions on Software Engineering, 49*(7), 3793–3810.

Rust Foundation. (2023). *The Rust programming language (2nd ed.)*. Retrieved August 16, 2025, from https://doc.rust-lang.org/book/

Smolka, S., Giesen, J., Winkler, P., Draissi, O., Davi, L., Karame, G., & Pohl, K. (2023). *Fuzz on the beach: Fuzzing Solana smart contracts*. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23),1197–1211. ACM.

Solidity Team. (2024). *Solidity documentation*. Retrieved September 2, 2025, from https://docs.soliditylang.org

Szabo, N. (1996). Smart contracts: Building blocks for digital markets. *Extropy*, 16(1), 18–23.

Vacca, A., Di Sorbo, A., Visaggio, C. A., & Canfora, G. (2021). A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. *Journal of Systems and Software, 174*, 110891.

Varela-Vaca, Á. J., & Quintero, A. M. R. (2021). Smart contract languages: A multivocal mapping study. *ACM Computing Surveys, 54*(1), 1–38.

Voloder, A., & Di Angelo, M. (2023). Comparison of smart contract platforms from the perspective of developers. In *International Conference on Blockchain*. Springer. 104–118.

Wesley, S., Christakis, M., Navas, J. A., Trefler, R. J., Wüstholz, V., & Gurfinkel, A. (2022). Verifying Solidity smart contracts via communication abstraction in SmartACE. In *VMCAI 2022* (LNCS) 13182, Springer, 425–449.

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, *151*(2014), 1-32.

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

Zou, W., Lo, D., Kochhar, P. S., Le, X. D., Xia, X., Feng, Y., Chen, Z., & Xu, B. (2021). Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering, 47*(10), 2084–2106.