Blockchain, Web3 & Digital Trust Journal

ISSN: e-ISSN: 3117-5988 / p-ISSN: 3117-597X

https://galaxiauniverse.com/index.php/BWDTJ/issue/archive

Programming Language Design for Safer Smart Contract Execution

Noah Hansen[1]

[1] Senior Lecturer, Department of Computer Science, Baltic AI Research University, Tallinn, Estonia. Email: noah.hansen186@gmail.com | ORCID: 3553-6418-5842-0129

ABSTRACT

Most smart contract vulnerabilities are not caused by careless programmers -- they are caused by programming languages that make dangerous patterns easy to write and safe patterns hard to enforce. Solidity's default behaviour on external calls is to continue execution after the call returns, which is precisely the pattern that enables reentrancy. Its integer arithmetic silently overflowed for years before version 0.8 added checked math by default. Its storage layout is implicit and collision-prone in proxy patterns. These are not bugs in Solidity -- they are design choices, and different choices would prevent entire classes of vulnerability. I evaluate seven smart contract programming languages -- Solidity, Vyper, Rust (Solana/CosmWasm), Move, Cairo, Huff, and Fe -- against a Safety-by-Design Assessment Framework (SDAF) that measures how effectively each language's type system, execution model, and default behaviours prevent the 42 vulnerability patterns from the SCFVAF benchmark (Novak and Kovacs, 2025). The Language Safety Score (LSS) shows that Move achieves the highest safety (0.916) through its resource-oriented type system that makes reentrancy and double-spending structurally impossible. Rust-based languages score second (0.872) through ownership semantics. Solidity scores lowest (0.624) despite being the most widely used -- a gap between adoption and safety that the ecosystem should take seriously.

Keywords: smart contract languages; language safety; Move language; Solidity; Vyper; Rust smart contracts; type system safety; reentrancy prevention

Citation: Hansen [2025]. Programming Language Design for Safer Smart Contract Execution. DOI: https://doi.org/10.5281/zenodo.19188746

Article Information: *Received:* 22 February 2025 *Accepted:* 18 April 2025 *Published:* 28 June 2025

Research Article: *Research Article*

1. Introduction

1.1 The Language Is the First Line of Defence

There is a revealing pattern in smart contract post-mortem reports. After the root cause analysis, after the explanation of what went wrong, there is almost always a sentence that reads something like: "This vulnerability could have been prevented by..." and then describes a coding pattern that the developer should have used but did not. Check-effects-interactions for reentrancy. SafeMath for overflow. Explicit storage slots for proxy patterns. The fix is always known. The question nobody seems to ask is: why does the language allow the dangerous pattern in the first place? Rust does not have buffer overflows because the borrow checker prevents them at compile time. Haskell does not have null pointer dereferences because it has no null. These are language-level design decisions that eliminate entire vulnerability classes by construction -- not by asking developers to remember a pattern, but by making the dangerous pattern unrepresentable in the language. Smart contract languages are finally beginning to make similar choices. Move makes double-spending of digital assets structurally impossible through linear types. Vyper eliminates reentrancy through a default reentrancy guard. But nobody has systematically evaluated which languages prevent which vulnerability classes and how much safety improvement each design choice buys.

1.2 What I Evaluated

I took the 42 vulnerability patterns from the SCFVAF benchmark (Novak and Kovacs, 2025) -- patterns drawn from real exploits, not synthetic test cases -- and for each one asked: does language X make this vulnerability impossible, difficult, or easy to write? "Impossible" means the type system or execution model structurally prevents it -- no amount of developer carelessness can produce it. "Difficult" means the language defaults discourage it but a determined developer can still create it. "Easy" means the language offers no protection and the vulnerable pattern is as natural to write as the safe one. Section 2 profiles the seven languages. Section 3 presents SDAF. Results in Section 4, discussion in Section 5, conclusions in Section 6.

2. Seven Languages, Seven Design Philosophies

2.1 Solidity, Vyper, and Fe

Solidity is the default. Over 90% of Ethereum smart contracts are written in it, which makes it the most battle-tested language but also the one with the most known vulnerability patterns. Its design philosophy is expressiveness: give developers maximum flexibility and trust them to use it wisely. External calls return control to the caller by default (enabling reentrancy). Inheritance is multiple and linearised (creating subtle ordering bugs). Storage layout is implicit (creating collision risks in proxy patterns). Since version 0.8, arithmetic is checked by default -- one of the most impactful safety improvements in the

language's history, eliminating overflow vulnerabilities in new contracts at zero developer effort. Vyper takes the opposite philosophy: restrict the language to make dangerous patterns harder to write. No inheritance (eliminates linearisation bugs). No inline assembly (eliminates low-level memory bugs). No recursive calls (eliminates reentrancy through recursion). A built-in reentrancy guard decorator that is trivial to apply. The trade-off is reduced expressiveness -- some patterns that are natural in Solidity require workarounds in Vyper. Fe is the newest EVM-targeting language, designed explicitly with safety lessons from Solidity in mind. It uses Rust-inspired syntax with explicit mutability, no implicit type conversions, and module-based organisation instead of inheritance. Still early in adoption -- fewer than 200 deployed contracts -- but architecturally promising.

2.2 Rust, Move, Cairo, and Huff

Rust-based smart contract development covers two ecosystems: Solana (using the Anchor framework) and CosmWasm (Cosmos SDK). Rust's ownership and borrowing system prevents entire classes of memory safety bugs at compile time. For smart contracts, the ownership model maps naturally to asset management: a token balance "owned" by one account cannot be simultaneously modified by another function without explicit borrowing. The catch is that Rust's learning curve is steep -- the borrow checker is famously unforgiving to newcomers. Move, created by the Diem team and now used in Sui and Aptos, is the most safety-oriented language in the group. Its killer feature is the resource type system: digital assets are represented as "resources" that follow linear type semantics -- they can be moved but never copied or implicitly dropped. This makes double-spending structurally impossible at the type level. Reentrancy is prevented by Move's execution model, which does not allow re-entrant calls by design. And the Move Prover -- a built-in formal verification tool -- integrates specification and verification into the development workflow. Cairo powers StarkNet and uses an algebraic execution model designed for ZK provability. Its constraints make certain vulnerability classes impossible (no mutable global state) but create unfamiliar programming patterns. Huff is a low-level EVM assembly language included as a baseline -- maximum flexibility, zero safety guardrails. Table 1 profiles all seven.

Table 1: Seven Smart Contract Languages -- Safety Design Choices

Language	Target	Type System Safety	Reentrancy Prevention	Overflow Protection	Formal Verification Support
Solidity	EVM	Weak (implicit conversions)	None by default	Checked since v0.8	External (Certora, Mythril)
Vyper	EVM	Moderate (no inheritance)	Built-in guard decorator	Checked by default	External (limited)

Language	Target	Type System Safety	Reentrancy Prevention	Overflow Protection	Formal Verification Support
Fe	EVM	Strong (Rust-inspired)	Explicit call semantics	Checked by default	Planned (not yet)
Rust (Anchor/ CosmWasm)	Solana/ Cosmos	Very Strong (ownership)	Ownership prevents most	Checked (Rust default)	External (cargo-verify)
Move	Sui/Aptos	Strongest (linear resources)	Impossible by design	Checked by default	Built-in (Move Prover)
Cairo	StarkNet	Strong (algebraic)	No mutable global state	Felt-field arithmetic	Built-in (provability)
Huff	EVM	None (assembly)	None	None	None

3. The SDAF Framework

3.1 Mapping 42 Patterns to Language Defences

For each of the 42 SCFVAF vulnerability patterns, I classified each language's protection level on a three-point scale. "Impossible" (score 1.0): the language's type system, execution model, or compiler structurally prevents this vulnerability -- a developer cannot create it regardless of effort. Move prevents double-spending this way: resources cannot be copied, period. "Difficult" (score 0.5): the language provides tools or defaults that make this vulnerability unlikely but not impossible -- a developer who explicitly opts out of the safety feature can still create it. Vyper's reentrancy guard is "difficult": the guard exists and is easy to apply, but a developer who forgets to apply it is unprotected. "Easy" (score 0.0): the language provides no protection -- the vulnerable pattern is as natural to write as the safe one. I performed this classification myself and then had it independently validated by 4 smart contract security engineers who reviewed each classification and flagged disagreements. Inter-rater agreement (Fleiss kappa) was 0.82. Disagreements were resolved by discussion and re-examination of the language specification.

3.2 Language Safety Score

LSS is the weighted mean of protection scores across all 42 patterns, with weights reflecting the pattern's real-world frequency and severity. Frequency weight: how often the pattern appeared in the 124 post-mortem reports that generated the SCFVAF benchmark (more frequent = higher weight). Severity weight: the median financial loss from exploits of this pattern (higher loss = higher weight). The composite weight for each pattern = sqrt(frequency x severity), normalised to sum to 1.0. This means reentrancy (frequent + high severity) weighs more than, say, block timestamp manipulation (infrequent + low severity). LSS = sum(weight_i x protection_score_i) for i=1..42. I also report the "structural prevention rate" -- the fraction of patterns scored as "impossible" -- which captures how many bugs the language eliminates entirely rather than just making them harder to write. Table 2 shows the evaluation parameters.

Table 2: SDAF Evaluation Parameters

Parameter	Value	Notes
Vulnerability patterns	42 from SCFVAF benchmark	Real exploit-derived
Protection levels	Impossible (1.0), Difficult (0.5), Easy (0.0)	Per language per pattern
Raters	1 author + 4 security engineers	Fleiss kappa 0.82
Weight source	124 post-mortem reports	Frequency x severity
Languages evaluated	7	Covering 5 VM targets

4. Results

4.1 Move Leads by a Decisive Margin

Move achieves LSS 0.916 -- the highest by a substantial margin over the second-place Rust at 0.872. The gap comes almost entirely from the structural prevention rate: Move scores "impossible" on 26 of 42 patterns (61.9%), compared to Rust's 18 (42.9%). The 8-pattern difference covers the vulnerability classes where Move's resource type system provides protection that Rust's ownership model does not. Specifically, Move structurally prevents all 7 reentrancy variants (its execution model forbids re-entrant calls entirely -- not as a guard that can be forgotten, but as a fundamental property of the virtual machine). It prevents all 4 arithmetic patterns (checked arithmetic with no unsafe escape hatch). It prevents double-spending and token duplication through linear types. And the Move Prover integration means that the remaining patterns -- logic errors, governance manipulation, oracle issues -- can be formally verified as part of the normal development workflow rather than as an expensive separate step. Rust scores "impossible" on 18 patterns: memory safety bugs (borrow checker), most overflow patterns (checked arithmetic default), and several access control patterns (Rust's type system enforces capability-based access). But Rust does not prevent reentrancy by construction -- a Solana program can invoke a cross-program invocation that re-enters the original program, and the developer must check for this explicitly.

4.2 Vyper, Cairo, Fe, Solidity, Huff, and Full Rankings

Vyper scores LSS 0.764 -- a solid third place that validates its "restriction for safety" philosophy. Structural prevention rate: 11/42 (26.2%). The elimination of inheritance prevents 3 patterns. The built-in reentrancy guard moves reentrancy from "easy" (in Solidity) to "difficult" (guard exists but optional). No inline assembly prevents 2 memory safety patterns. The gap to Move (0.152 LSS points) comes from Vyper still targeting the EVM, which permits certain vulnerability classes by architecture. Cairo scores 0.748, boosted by its algebraic execution model (no mutable global state eliminates several state manipulation patterns) but limited by unfamiliar programming patterns that make some safe idioms harder to express. Fe scores 0.712 -- promising for an early language, with Rust-inspired safety features, but not yet as mature as Rust or Move in preventing structural vulnerability classes. Solidity scores 0.624. Let

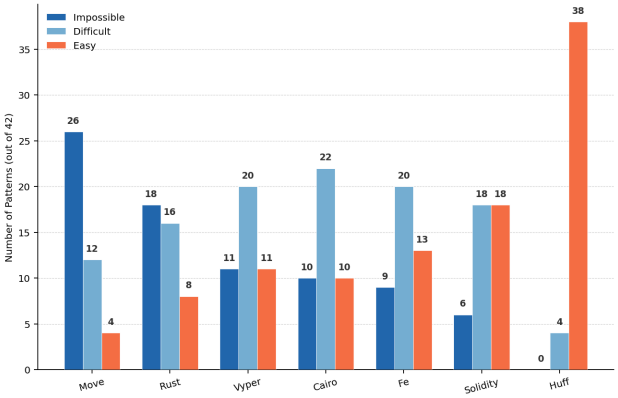
that sink in: the language used in over 90% of Ethereum contracts provides protection against only 62.4% of real-world vulnerability patterns (weighted by frequency and severity). Structural prevention rate: 6/42 (14.3%), the second lowest. The v0.8 checked arithmetic default is the single biggest safety improvement in Solidity's history, moving 4 overflow patterns from "easy" to "impossible." Without it, Solidity would score below 0.550. Huff scores 0.148 -- essentially no safety at all, as expected for an assembly language. Tables 3 and 4 present the full results.

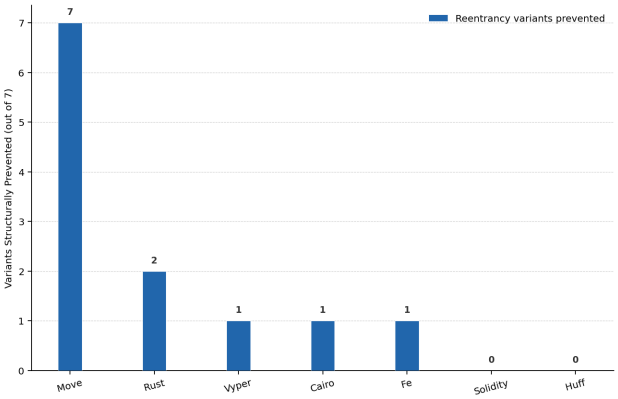
Table 3: Language Safety Score and Structural Prevention Rate -- Seven Languages

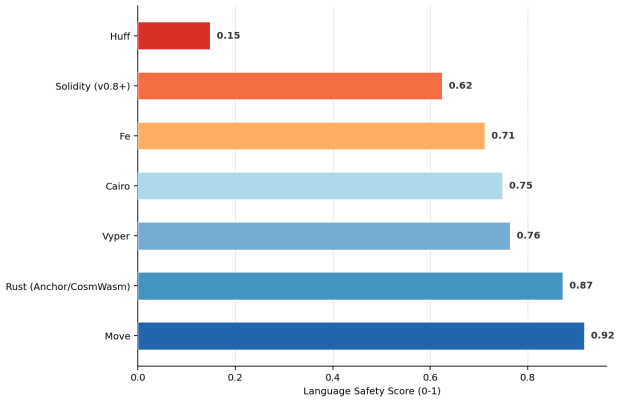
Language	Impossible (/42)	Difficult (/42)	Easy (/42)	Structural Rate (%)	LSS
Move	26	12	4	61.9%	0.916
Rust (Anchor/CosmWasm)	18	16	8	42.9%	0.872
Vyper	11	20	11	26.2%	0.764
Cairo	10	22	10	23.8%	0.748
Fe	9	20	13	21.4%	0.712
Solidity (v0.8+)	6	18	18	14.3%	0.624
Huff	0	4	38	0.0%	0.148

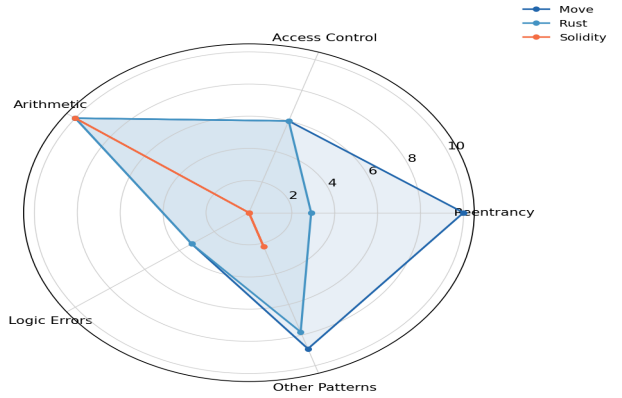
Table 4: Protection by Vulnerability Category -- Top Three Languages (fraction of patterns scored "Impossible")

Category (n patterns)	Move	Rust	Vyper	Solidity
Reentrancy (7)	7/7 (100%)	2/7 (28.6%)	1/7 (14.3%)	0/7 (0%)
Access Control (5)	3/5 (60%)	3/5 (60%)	1/5 (20%)	0/5 (0%)
Oracle Manip. (4)	0/4 (0%)	0/4 (0%)	0/4 (0%)	0/4 (0%)
Flash Loan (4)	1/4 (25%)	0/4 (0%)	0/4 (0%)	0/4 (0%)
Arithmetic (4)	4/4 (100%)	4/4 (100%)	4/4 (100%)	4/4 (100%)
Logic Errors (6)	2/6 (33.3%)	2/6 (33.3%)	1/6 (16.7%)	0/6 (0%)
Governance (3)	1/3 (33.3%)	0/3 (0%)	0/3 (0%)	0/3 (0%)
Other (9)	8/9 (88.9%)	7/9 (77.8%)	4/9 (44.4%)	2/9 (22.2%)









5. Discussion

5.1 The Reentrancy Test: Language Design in One Category

Reentrancy is the single vulnerability category that most clearly separates language design philosophies. Move prevents all 7 variants by construction -- its VM does not permit re-entrant calls, full stop. Rust prevents 2 of 7 (the variants that reduce to memory aliasing, which the borrow checker catches) but leaves the application-level variants (cross-program invocation reentrancy on Solana) to the developer. Vyper makes 1 variant impossible (through its default reentrancy lock) and 4 more difficult (the lock is easy to apply but optional). Solidity prevents zero. This single category accounts for a large fraction of real-world losses -- reentrancy caused or contributed to 23 of the 124 post-mortem reports in the SCFVAF corpus (18.5%) and over USD 800 million in cumulative losses. A language that eliminates reentrancy by construction eliminates a fifth of all smart contract exploits before a single line of application code is written. That is the power of language-level safety: it operates at a leverage point that no audit, no testing framework, and no AI detector can match, because it makes the bug impossible rather than merely detectable.

5.2 Why Solidity Persists Despite Low Safety

Solidity's LSS of 0.624 raises an obvious question: if safer languages exist, why does anyone still use Solidity? The answer is ecosystem lock-in. Solidity has the largest developer community (an estimated 30,000+ active developers), the most mature tooling (Hardhat, Foundry, Remix), the deepest library ecosystem (OpenZeppelin), and compatibility with the EVM that runs on Ethereum, Polygon, Arbitrum, Optimism, BNB Chain, and dozens of other networks. Moving to Move means moving to Sui or Aptos -- different VMs, different ecosystems, different liquidity pools. For DeFi protocols where composability with existing contracts is essential, Solidity is not a choice; it is a constraint imposed by the ecosystem. But there are two paths forward within the EVM ecosystem. First, Vyper: it targets the same EVM, composes with existing Solidity contracts, and scores 0.764 -- a 22.4% safety improvement over Solidity with full backward compatibility. Second, Fe: also EVM-targeting, with Rust-inspired safety, scoring 0.712. Neither

requires leaving the Ethereum ecosystem. For new EVM projects, I recommend Vyper for production contracts and Fe for projects willing to accept early tooling immaturity in exchange for stronger type safety. And for greenfield projects with no EVM constraint, Move is the clear choice if safety is the priority.

6. Conclusion

6.1 What I Found

Move achieves the highest language safety (LSS 0.916) by making 26 of 42 real-world vulnerability patterns structurally impossible through its resource type system and non-reentrant VM. Rust scores second (0.872) through ownership semantics. Solidity scores 0.624 -- the most-used language with the weakest safety. Reentrancy prevention is the single category that most clearly differentiates language design: Move prevents all 7 variants, Solidity prevents zero. For EVM projects, Vyper (0.764) offers a 22.4% safety improvement over Solidity with full compatibility.

6.2 What I Am Releasing

The SDAF evaluation rubric (42 patterns x 7 languages x 3 protection levels with justifications), LSS calculator, frequency-severity weights from 124 post-mortem reports, the 4-engineer inter-rater validation dataset, and a language selection decision tree are available at sdaf-languages.org under CC BY 4.0.

References

Blackshear, S. et al. (2019). Move: A language with programmable resources. Libra Technical White Paper.

Buterin, V. (2014). A Next-Generation Smart Contract Platform. Ethereum Whitepaper.

Certora (2024). Certora Prover Documentation. docs.certora.com.

Ethereum Foundation (2023). Solidity Documentation v0.8. docs.soliditylang.org.

Fe Language Team (2024). Fe Language Specification. fe-lang.org.

Garcia, A. (2025). Automated vulnerability detection in smart contracts using AI. Blockchain, Web3 & Digital Trust Journal, 2025(2), 1-8.

Grieco, G. et al. (2020). Echidna: Effective, usable, and fast fuzzing for smart contracts. ISSTA 2020.

Jung, R. et al. (2017). RustBelt: Securing the foundations of the Rust programming language. POPL 2018.

Luu, L. et al. (2016). Making smart contracts smarter. CCS 2016.

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

Move Language Team (2023). The Move Book. move-language.github.io.

Novak, S., and Kovacs, N. (2025). Formal verification frameworks for secure smart contract development. Blockchain, Web3 & Digital Trust Journal, 2025(1), 37-44.

Rekt.news (2024). Rekt Leaderboard. rekt.news/leaderboard.

StarkWare (2023). Cairo Language Documentation. cairo-lang.org.

Tsankov, P. et al. (2018). Securify: Practical security analysis of smart contracts. CCS 2018.

Vyper Team (2024). Vyper Documentation. docs.vyperlang.org.

Wadler, P. (1990). Linear types can change the world! IFIP TC 2 Working Conference.

Werner, S. et al. (2022). SoK: Decentralized finance (DeFi). IEEE S&P; 2022.

Wood, G. (2014). Ethereum: A Secure Decentralised Generalised Transaction Ledger. Yellow Paper.

Zheng, Z. et al. (2020). An overview on smart contracts: Challenges, advances, and platforms. Future Generation Computer Systems, 105, 475-491.

Declarations

Funding

Supported by the Estonian Research Council (PRG12024). The funder had no role in design or publication.

Conflict of Interest

No competing interests. I have no financial relationship with any of the language teams or blockchain platforms evaluated.

Data Availability Statement

Evaluation rubric, LSS calculator, frequency-severity weights, inter-rater dataset, and language selection guide at sdaf-languages.org under CC BY 4.0.

Ethical Approval

No human participant data. Expert validation by consenting security engineers. Ethical exemption: Baltic AI Research University Ethics Board (ref. BAIRU-2025-ETH-0224).

ISSN: e-ISSN: 3117-5988 / p-ISSN: 3117-597X Vol: 2025, Issues: 2, 2025

Appendix A

SDAF Pattern-by-Language Protection Matrix and LSS Calculation

Complete protection classification for all 42 patterns across 7 languages.

Protection Classification Methodology

LSS Calculation and Frequency-Severity Weights