A Systematic Classification of Vulnerabilities in MoveEVM Smart Contracts (MWC)

Selçuk Topal[a]

[a] *Gebze Technical University, Department of Mathematics, stopal@gtu.edu.tr, Gebze, 41000, , Turkiye*

Abstract

Combining the Move programming language with Ethereum Virtual Machine (EVM) compatibility—termed MoveEVM—has produced a new class of smart contract platforms that mix the expressive capability and infrastructure maturity of Ethereum with resource-oriented safety guarantees. Although the Move language was first meant to eradicate many known vulnerabilities using linear resource types and strict module ownership, its adaptation inside an EVM-compatible execution model presents special difficulties that are yet understudied in current literature. Conventional vulnerability classification systems, including the SWC registry, lack semantic granularity to handle the hybrid execution environment of MoveEVM and are optimized for Solidity. This work suggests a first methodical vulnerability classification system designed especially for smart contracts based on MoveEVM. Covering a broad spectrum of problems including bytecode model inconsistencies, inter-module invariants, hybrid gas semantics, meta-transaction spoofing, and artificial intelligence-integrated logic risks, we present a thorough taxonomy comprising six semantic frames and 37 categorized vulnerability types (MWC-100 to MWC-136). By means of both static and dynamic analysis of real-world MoveEVM contracts—especially from Aptos and Sui ecosystems—we show that a considerable fraction of vulnerabilities are neither formally reduced by current Move verifying systems nor captured by EVM-centric tools. We also examine how formal verification methods, LLM-based prompt pipelines, and AI-assisted audit agents might operationalize our taxonomy, so enabling scalable and logic-aware auditing. Our results expose the emerging security patterns resulting from interaction among linear type systems, capability-based access control, and EVM bytecode in production environments. Apart from offering a disciplined basis for next tooling and formal methods research, the suggested MoveEVM Weakness Classification (MWC) system helps developers and auditors to reason about hybrid vulnerabilities in a principled, repeatable, and automation-friendly way. This work thus prepares the way for the creation of more safe, verifiable, and maintainable smart contracts in hybrid blockchain systems of next generation.

Keywords: MoveEVM, Smart Contract Security, Vulnerability Taxonomy, Formal Verification, AI-Assisted Auditing, MWC

Contents

1. Introduction

Blockchain systems keep changing in security, scalability, and complexity, which fuels the demand for ever stronger smart contract languages and execution environments Wood (2014); Zhang et al. (2020). With a well-established tooling ecosystem and great acceptance, traditional platforms like Ethereum have led the way Buterin (2013). But Solidity, the original smart contract language for Ethereum, also reveals important security and expressiveness limitations Luu et al. (2016). These restrictions have inspired the creation of alternative languages including Move, first presented by Facebook's (now Meta's) Libra project Facebook (2019) and subsequently embraced by networks including Aptos and Sui Pierro et al. (2023); Labs (2022a).

Token economy and distributed finance (DeFi) have become rather well-known among people in the last years Werner et al. (2021). Underlying most DeFi and token projects, the Ethereum blockchain architecture lets peer-to-peer currency flow, transparent auction systems, and distributed open-interest exchanges Antonopoulos and Wood (2018). To generate trade fungible and non-fungible tokens on the Ethereum blockchain, several token standards are extensively embraced Foundation (a,b). The explosive expansion of Ethereum-based tokens generated different demand for tools and solutions Li et al. (2022). Having seen the market, many investors and other consumers clearly need to locate, follow, and evaluate tokens. A lot of tools have been created to meet these criteria including token lists, distributed exchanges, explorers, portfolio management tools, arbitrage bots, etc. Bartoletti et al. (2020). These tools provide thorough and strong services, which helps different DeFi apps on Ethereum token ecosystems to grow.

Behind the market, users of the Ethereum blockchain have produced tons of tokens. But since token creation calls for complex smart contract deployment and interactivity, common users unable of programming face significant challenges. More dangerously, on most of these tools, any user can generate arbitrary tokens on the Ethereum blockchain just by filling in a pertinent contract address. Given the simple access to token creation and the growing popularity of tokens and associated tools, it becomes imperative to methodically review token smart contracts from several angles.

A few pieces have lately aimed at dissecting Ethereum smart contracts Nikolić et al. (2018); Torres et al. (2021). Analyzing the vulnerabilities in Ethereum smart contracts became a hot issue since some security events attracted a lot of interest Atzei et al. (2017). But while they lack an interesting study of token smart contracts, current works concentrate on examining the incorrect semantic behaviors of Ethereum smart contracts. Token standards all center on the high-level semantics of the contracts. Semantic vulnerabilities including circulating coins, reentrancy, and so on still exist even if they are subtly defined in the handler actions He et al. (2020). There is a lack of a complete systematic classification of these vulnerabilities at the high-level language. One can consider vulnerabilities in code as weaknesses in code that render code vulnerable swcregistry.io (2024). To confirm the validity of these works, there is a need for strong empirical investigations on Ethereum smart contracts vulnerabilities. Current vulnerability and weakness databases feature only basic-type vulnerabilities that must be expanded. Moreover, vulnerability databases in other languages cannot be exactly used since programming languages inevitably bring their syntax, execution methods, etc.

Functioning as self-executing contracts with the terms of the agreement directly written into code, smart contracts are a breakthrough within the scene of blockchain technologies Szabo (1997). They cut operating expenses by removing the need for middlemen, enabling trustless transactions. Blockchain's distributed character improves security and openness, thus smart contracts appeal to many sectors, including finance, supply chains, and healthcare Zheng et al. (2020). Adoption of smart contracts, meanwhile, also presents special difficulties, especially related to security flaws that might be taken advantage of by malevolent actors Luu et al. (2016).

One of the weaknesses of smart contracts is a piece of code that, when carried out in the Ethereum environment and taken advantage of by an actor in a manner that causes Ether to be lost (Soud et al. (2024)). Regarding flaws, as in any software program, suboptimal coding structure and writing style can also be causes of problems. Smart contract codes differ from other software programs in that each one of their instructions requires a precise gas consumption. Said another way, any smart contract code instruction or function can be triggered by known address and cause the miners to follow up their processing as a transaction. The initial gas connected to this transaction is the cost of it. This implies that the contract is vulnerable even if the weakness is not taken advantage of since it would result in losing

Ether when it is triggered by the contract itself and carried out. For the above mentioned reasons, it is quite crucial to investigate the shortcomings of smart contracts as well as their vulnerabilities independent of their ever exploitation. Referred to as Ethereum Virtual Machine (EVM), Ethereum is a globally open distributed blockchain system supporting smart contracts. Although EVM contracts live on the blockchain in a Turing complete bytecode language, developers use high-level languages such Solidity (Bauer (2022)) or Vyper (Buterin (2018)) and subsequently compile to bytecode to be uploaded to the EVM. More precisely, both languages are elegant stringed arrays of hundreds of thousands of bytes including high-level language-based contract code instructions. The compilation is crucial since it hides the smart contract code from everyone wishing to view it, so shielding it from dangerous intrusions. Users of the EVM can create new contracts, call methods inside a contract, and move Ether. But once uploaded onto the blockchain, a smart contract code—derived from its bytecode using the Keccak algorithm—is immutable. Its hash serves as its specification. Not only that but also before uploading it, a deterministic algorithm generates an address for the code (and the contract produced from it).

The account address, a unique 160-bit hexadecimal string, specifies the EVM user account—external account or EOA for short. An account in EVM can carry Ether denoted in Wei, the cryptocurrency unit. The bytecode of a smart contract uses a particular amount of gas for each byte in executed instruction. Running on the blockchain, smart contracts are general-purpose digital programs. Furthermore including user accounts, a smart contract can call other smart contracts. Since Solidity is the most often used language in the EVM community and most of the implemented contracts on EVM are created using Solidity, this paper focuses on Ethereum smart contracts created in Solidity.

Designed to model assets as linear resources, Move (Blackshear et al. (2019)) is a statically typed, resource-oriented programming language meant to provide great safety guarantees. These semantics greatly lower the probability of some common Solidity vulnerabilities including reentrancy, integer overflows, and inadvertent asset duplication. Move lacks compatibility with the Ethereum Virtual Machine (EVM), so limiting its integration with current Ethereum-based tools and applications even if it offers stronger type safety and resource control. Originally built for the Libra blockchain, MoveEVM (Abrahimi (2023)) is an adaptation of the Ethereum Virtual Machine (EVM) including the Move programming language. Via a resource-oriented programming paradigm, it seeks to improve the expressiveness and safety of smart contract development. MoveEVM, aims to reduce typical vulnerabilities linked with conventional smart contracts by using Move's strong type system and access control mechanisms. Ensuring the dependability and security of applications developed on MoveEVM depends on an awareness of the weaknesses particular to this framework. MoveEVM stands for a new paradigm meant to close this distance. It presents an execution model that supports Move-based contracts inside an EVM-compatible runtime, so aggregating Move's resource-oriented reasoning with Ethereum's infrastructure backbone. This hybridization presents a special set of security issues that have not yet been fully investigated in scholarly or pragmatic settings, even if it provides the best of both worlds—secure asset modeling from Move and developer familiarity from EVM.

The body of current research and tools for smart contract security mostly focus on environments native to EVM. For example, almost exclusively on Solidity the Smart Contract Weakness Classification (SWC) Registry swcregistry.io (2024), MythX (Sayeed et al. (2020)), Slither (Feist et al. (2019a)), and other analysis tools concentrate. Although these tools offer important new perspectives on known EVM vulnerabilities, they cannot handle Move's semantic variations and the resulting emergent risks brought about by merging Move semantics with an EVM-like runtime.

This work aims to close this discrepancy by providing the first methodical classification of vulnerabilities particular to MoveEVM-based smart contracts. Our work starts with a thorough review of the MoveEVM implementation model and then looks at actual contracts and the related security concerns. Based on their root causes, attack paths, and runtime behaviors, we classify these weaknesses into logical groups that provide a fresh taxonomy grounded in both theoretical ideas and empirical data.

Motivation and Contributions

This research is motivated in two different directions. First, developers and auditors need a disciplined knowledge of MoveEVM's special security features as adoption of it speeds forward. Second, MoveEVM's hybrid execution character means that security concerns could show up in ways neither traditional Move nor EVM models could forecast alone. For instance, incorrect resource handling in a Move contract implemented on MoveEVM may bypass familiar Solidity-based checks, or vice versa, so generating non-trivial attack surfaces.

This work focuses on: identifying inherited vulnerabilities from both Move and EVM environments.

- Dividing fresh vulnerability types arising from their interaction.
- Showing overlaps and deviations, comparing these vulnerabilities to those noted in Ethereum and Solana by including pragmatic examples and case studies from Aptos and Sui ecosystems.

Contributions

This work makes primarily the following important contributions:

Based on static and dynamic analysis of actual contracts and audit reports, we provide a thorough and unique taxonomy of MoveEVM vulnerabilities.

1. We draw attention to security concerns the hybrid MoveEVM implementation model either magnifies or uniquely introduces.

2. We frame MoveEVM's strengths and shortcomings in the larger smart contract security scene by doing comparative analysis with vulnerabilities in Ethereum and Solana.

3. For developers and security analysts wishing to create or audit MoveEVM-based smart contracts, we provide specific advice.

This paper adds to the fundamental knowledge needed to create dependable tooling, audit practices, and educational materials for the next generation of smart contract platforms by offering the first focused classification of MoveEVM vulnerabilities.

2. Related Works

2.1. Vulnerability Taxonomies

For Ethereum and other chains, vulnerability taxonomies for smart contracts are established but still developing. Maintaining the *Smart Contract Weakness Classification (SWC)* registry (EIP-1470), the Ethereum community links numerical identities to contract weaknesses (e.g., SWC-101 = integer overflow/underflow) swcregistry.io (2024). Academic work and audits similarly list common Solidity/EVM bug classes including reentrancy, unchecked math, incorrect access control, etc.

Song et al. (2024) manually audited 652 Move contracts in the Move ecosystem, distilled eight defect types—half previously unreported—spanning logic errors, resource misuse, and more. Wu et al. (2025) methodically catalog real-world vulnerabilities (integer overflows, unsafe Rust use, oracle logic flaws, etc.), then compare tool support to Ethereum for Solana's Rust-based contracts.

These taxonomies guide the creation of analysis tools as well as language design—that is, Move's resource model.

2.2. Security Features of Move and MoveEVM

Move was developed as a smart contract language first intended for security. It implements a *resource-typed, linear* type system: structs that "must-move," so guaranteeing they cannot be replicated or thrown away Diem (2019). Using primitives like `move_to` and `move_from`, move modules own declared resource types and control all creation/destruction isolating state changes to authorized code (Patrignani and Blackshear (2023)).

Struct fields are private to the module; once generated, resources—once created—are handled as first-class values only transferable via explicit instructions. A bytecode verifier enforces this linearity at compile-time and, alternatively at runtime via safety checks. A 2023 Move verifier bug, for example, let a non-`drop` resource to drop, so violating this invariant Security (2023a).

Move's design avoids many typical mistakes in construction unlike Solidity, where asset safety must be enforced manually or with tools. Extending the language with an integrated specification language and automated formal verification, the Move Prover Zhong et al. (2020). Though it is still under development, MoveEVM seeks to include these safety assurances into an EVM-compatible runtime.

2.3. Analysis Tools and Frameworks

A wide range of static and dynamic analysis tools target smart contracts. For Ethereum, **Slither** is a popular static analyzer that compiles Solidity into SSA-form IR ("SlithIR") and applies vulnerability detectors Feist et al. (2019a). **Securify** Tsankov et al. (2018) uses Datalog-based analysis to infer semantic properties and verify compliance or detect violations. **Mythril** (Sharma and Sharma (2022)) and its cloud platform **MythX** (Songsom et al. (2022)) use symbolic execution and SMT solving for bug detection.

For fuzzing and runtime testing Fu et al. (2024), tools like **Echidna** (Grieco et al. (2020)) and **Manticore** (Mossberg et al. (2019)) are used.

In the Move ecosystem, tools are emerging. The **Move Prover** (Dill et al. (2022)) translates annotated Move code to Boogie and verifies properties against formal specifications. Song et al. Song et al. (2024) introduced **MoveScan**, which translates Move bytecode to an intermediate representation and detects common defect patterns. **MoveLint** (Praitheeshan et al. (2021)) offers lightweight static checks on Move codebases.

Recent work uses these tools for LLM-based detection: the Smartify framework compares vulnerability reports against Move Prover, MoveLint, and MoveScan to validate detection accuracy Karanjai et al. (2025).

2.4. Formal Verification and Empirical Analysis

Various tools for both static and dynamic analysis aim at smart contracts. Popular static analyzer Ethereum **Slither** (Feist et al. (2019b)) compiles Solidity into SSA-form IR ("SlithIR") and runs vulnerability detectors. By means of Datalog-based analysis, **Securify** Tsankov et al. (2018) deduces semantic properties and checks compliance or detects violations. Symbolic execution and SMT solving for bug discovery are used by **Mythril** (Sharma and Sharma (2022)) and its cloud platform **MythX** Sayeed et al. (2020).

Tools including **Echidna** Grieco et al. (2020) and **Manticore** Mossberg et al. (2019) are applied for fuzzing and runtime testing.

Tools are developing in the Move ecology. Translating annotated Move code to Boogie, the **Move Prover** (Zhong et al. (2020)) verifies properties against formal specifications. Translating Move bytecode to an intermediary representation and identifying common defect patterns, (Song et al. (2024)) presented **MoveScan**.

Using these tools for LLM-based detection, recent work validates detection accuracy (Karanjai et al. (2025)) by comparing vulnerability reports against Move Prover, MoveLint, and MoveScan.

2.5. Language Comparisons

The main distinctions between Solidity, Rust, and Move are highlighted by comparative language studies. Move prevents unauthorized duplication or loss of assets by incorporating linear resource types and drawing inspiration from Rust's ownership model (Diem (2019); Blackshear et al. (2019)). Although

Solidity is developer-friendly and flexible, it lacks native linearity, which makes it vulnerable to integer bugs and reentrancy unless tools are used to mitigate them(Feist et al. (2019a)).

Although memory safety is advantageous for Rust-based Solana programs, Wu et al. (2025) note that integer overflows are still a real-world problem. By enforcing invariants at the bytecode and specification levels, Move circumvents these problems Dill et al. (2022).

2.6. Audit reports and case studies

Current audits and incident reports shed light on practical problems in move-based chains. A critical Move verifier bypass bug that permitted resource dropping without the `drop` ability Security (2023a,b) was found by Zellic. A high-severity Move bug in Sui that enabled crafted bytecode to crash validators was reported by HackenProof HackenProof (2023). A Move VM vulnerability in Aptos and Sui that allows for denial-of-service attacks through twisted bytecode was discovered by Numen Cyber Labs (2022b).

The largest known empirical audit of Move contracts was carried out by Song et al. (2024), who found systemic patterns and defect density in both Aptos and Sui. These real-world examples demonstrate how Move's safety features greatly lower smart contract risks without completely eliminating them, which is why reliable analysis tools and validated code are crucial.

2.7. Understanding Vulnerabilities in Smart Contracts

Although they are transforming the way contracts and transactions are carried out on blockchain platforms, smart contracts do have certain drawbacks. Because of blockchain's immutability and decentralization, as well as the complexity of programming smart contracts, security flaws can have serious repercussions. In order to identify and mitigate potential risks in smart contracts, developers, auditors, and researchers must have a thorough understanding of these vulnerabilities.

2.7.1. Typical Vulnerabilities

Despite their many benefits, smart contracts are vulnerable to a number of flaws that could result in serious security breaches. Among the most prevalent categories of vulnerabilities are:

1. **Reentrancy Attacks:** One of the most well-known flaws in smart contracts is reentrancy, which was infamously taken advantage of in the 2016 DAO hack. When a contract calls another contract externally before updating its internal state, it creates a vulnerability that enables the called contract to return to the original contract and change its state before the first call is finished.

2. **Integer Overflow and Underflow:** This vulnerability occurs when arithmetic operations produce unexpected behavior by exceeding the maximum or minimum value that can be stored in a variable. A large positive number could be produced by subtracting one from a zero value, for instance, which could result in unauthorized access or contract state manipulation.

3. **Gas Limit and Loops:** The amount of computation that can be done in a single transaction is limited by the gas limit of smart contracts. Transaction failures may result from contracts with unbounded loops using excessive amounts of gas. By creating transactions that result in excessive gas consumption, attackers may take advantage of this and essentially cause a denial-of-service (DoS) on the contract.

4. **Access Control Issues:** When creating smart contracts, appropriate access control is essential. When functions are made available to unauthorized users, access control vulnerabilities may arise, giving malevolent actors the ability to alter important contract states or take money out.

5. **Timestamp Dependence:** Block timestamps are used in certain contracts for crucial functions like determining the legality of actions or enforcing deadlines. Attackers can take advantage of this vulnerability by mining blocks and manipulating block timestamps.

2.7.2. Specific Vulnerabilities in MoveEVM

Despite the fact that many flaws are shared by different smart contract platforms, MoveEVM presents particular difficulties and vulnerabilities because of its particular implementation and design. For example:

1. **Resource Mismanagement:** MoveEVM's resource-oriented programming paradigm emphasizes ownership and resource management. However, improper handling of resource transfers can lead to vulnerabilities, such as losing track of resource ownership or creating unintended resource duplication.

2. **Type Safety Issues:** The resource-oriented programming paradigm of MoveEVM places a strong emphasis on resource management and ownership. However, ineffective resource transfer management can result in vulnerabilities like unintentional resource duplication or losing track of resource ownership.

3. **Insufficient Testing and Formal Verification:** By using formal verification, MoveEVM seeks to increase the dependability of smart contracts. Contracts without thorough testing, however, might still have undiscovered weaknesses, especially those resulting from logical errors in implementation.

Since it establishes the basis for efficient vulnerability detection and mitigation techniques in MoveEVM smart contracts, an understanding of these vulnerabilities is essential for both developers and researchers.

3. Proposed Vulnerability Taxonomy for MoveEVM

In this section, we introduce a detailed vulnerability taxonomy for MoveEVM-based smart contracts. Each category is denoted by an identifier (MWC-XXX) and is designed to support automated or manual risk assessments. The categories are

grouped into six primary classes based on their semantic and technical nature.

3.1. State Management Vulnerabilities

- **MWC-100:** Frozen contract state due to improper state transitions (state transition analysis)
- **MWC-101:** Undefined state behavior when contract state variables are uninitialized (invariant verification)
- **MWC-102:** Lack of rollback protection causing incomplete transaction execution (atomicity proofs)
- **MWC-103:** Invalid loop termination leading to infinite execution (loop termination analysis)
- **MWC-104:** Unvalidated external calls causing undefined execution paths (module invocation proofs)
- **MWC-105:** Dead code execution causing unnecessary gas usage (static code analysis)
- **MWC-106:** Unreachable states in finite state machines due to logic errors (FSM coverage analysis)
- **MWC-107:** Contract state race condition causing unintended state changes (concurrency proofs)

3.2. Storage and Data Safety

- **MWC-108:** Data overwriting in storage leading to unexpected value changes (storage write protection)
- **MWC-109:** Unintended variable mutability allowing modification of supposedly immutable variables (immutability proofs)

3.3. Token and Asset Lifecycle Risks

- **MWC-110:** Unexpected token burn due to missing validation checks (asset lifecycle proofs)
- **MWC-111:** Unauthorized token minting by malicious actors (capability verification)
- **MWC-112:** Token supply overflow exceeding the defined hard cap (arithmetic safety checks)
- **MWC-113:** Improper reward distribution leading to unfair token allocation (reward calculation verification)
- **MWC-114:** Circular token transfers creating infinite loops (execution path analysis)
- **MWC-115:** Unauthorized token freezing without proper authorization (capability restriction proofs)
- **MWC-116:** Unexpected decimal precision loss due to rounding errors (precision analysis)
- **MWC-117:** Incorrect vesting schedule unlocking tokens earlier or later than expected (vesting condition verification)
- **MWC-118:** Improper treasury management leading to fund misallocation (fund allocation analysis)
- **MWC-119:** Unauthorized staking withdrawals allowing excessive asset withdrawals (stake locking proofs)

3.4. Cryptographic and Signature Security

- **MWC-120:** Weak signature verification causing unauthorized transactions (cryptographic proofs)
- **MWC-121:** Predictable key generation making cryptographic keys vulnerable (entropy verification)
- **MWC-122:** Unprotected encryption keys exposing private key data (confidentiality analysis)

3.5. Oracles, MEV, and Cross-Chain Risks

- **MWC-123:** Malicious oracle manipulation affecting contract execution (oracle verification proofs)
- **MWC-124:** Time-based side channel attacks revealing sensitive contract execution information (timing analysis)
- **MWC-125:** Front-running via predictable execution order enabling MEV exploitation (execution order verification)
- **MWC-126:** Malicious reorg attacks causing blockchain transaction instability (state finality proofs)
- **MWC-127:** Cross-chain replay attacks due to lack of unique nonces (chain isolation verification)
- **MWC-128:** Public private key leakage exposing sensitive cryptographic data (key safety proofs)

3.6. Emerging Categories: AI, Governance, Upgrades

- **MWC-129:** Unauthorized zero-knowledge proof submission leading to fake proofs (ZKP verification)
- **MWC-130:** Algorithm bias in AI-based smart contracts causing unfair decisions (bias detection algorithms)
- **MWC-131:** Data poisoning in AI-driven contracts manipulating training data (data integrity verification)
- **MWC-132:** AI decision-making manipulation altering contract execution (model robustness testing)
- **MWC-133:** Faulty DAO governance execution failing to follow voting rules (DAO proposal verification)
- **MWC-134:** Unprotected smart contract upgrades introducing new vulnerabilities (upgrade path verification)
- **MWC-135:** Poor Layer 2 integration causing execution inconsistencies (L2 bridge verification)
- **MWC-136:** Cross-chain messages are not validated correctly, leading to desynchronization (interoperability testing)

3.7. Supplementary Audit Dimensions

In addition to the technical taxonomy, the following audit perspectives are integrated in all evaluation reports:

- **Code Quality:** Is the code well-structured, readable, and appropriately commented?
- **Security Practices:** Are best security practices followed? Are additional mitigations applied to high-risk zones?
- **Fraud Analysis:** We assess risk from fee scams, redirections, unlimited minting, emergency fees, ownership, blacklisting, and transaction restrictions.
- **Overall Assessment:** The contract is evaluated as *Passed* or *Failed*, with justification based on high-risk findings.
- **General Security Posture:** Summary of critical risks and overall design strength in exactly eight evaluative sentences.

This taxonomy provides a formal lens for evaluating MoveEVM contracts against emerging and classical attack surfaces, including cross-domain, AI-driven, and cryptographically sensitive vulnerabilities.

4. Frame-Based Classification of MoveEVM Vulnerabilities

4.1. Motivation

Despite the inherent safety guarantees of Move's linear type system, MoveEVM inherits numerous semantic and runtime characteristics from the Ethereum Virtual Machine (EVM), introducing hybrid complexity and novel attack surfaces. As MoveEVM adoption rises in modular, zk-compatible, and L2 environments, the community lacks a cohesive, actionable classification framework that bridges both Move semantics and EVM legacy behavior. This section introduces a frame-based classification of vulnerabilities, defined by 37 uniquely identified MoveEVM Weakness Classes (MWC-100 to MWC-136), providing structured insight into emergent risks.

4.2. Classification Overview

Our vulnerability classification comprises **six top-level frames**, each reflecting a unique semantic or architectural dimension of the MoveEVM execution model. Each frame is populated by granular, code-assigned vulnerability types (MWC-n), enabling reproducibility, automation, and audit alignment.

1. **F1. Bytecode Model Inconsistencies (BMI)**
   - **MWC-100** — Type rule circumvention via EVM ABI deserialization
   - **MWC-101** — Unsafe encoding of Move structs in raw calldata
   - **MWC-102** — Bytecode re-interpretation between Move-EVM boundaries
2. **F2. Inter-Module Invariant Violations (IMI)**
   - **MWC-103** — Resource leakage across module interfaces
   - **MWC-104** — Inconsistent mutability between caller and callee modules
   - **MWC-105** — Failure to re-establish postconditions in inter-module calls
3. **F3. State Reentrancy and Synchronization Bugs (SRS)**
   - **MWC-106** — Hybrid reentrancy between Move and EVM modules
   - **MWC-107** — Callback-based state mutation violating linearity
   - **MWC-108** — Interleaved writes to Move storage via external callouts
   - **MWC-109** — Inconsistent ordering of resource locks in parallel executions
4. **F4. Meta-Transaction and Signature Spoofing (MTS)**
   - **MWC-110** — Ambiguous domain separation in signature verification
   - **MWC-111** — Missing nonce protection in off-chain meta-transactions
   - **MWC-112** — Reused signatures across heterogeneous domains (EVM $\rightleftarrows$ Move)
5. **F5. Gas Semantics Manipulation (GSM)**
   - **MWC-113** — Underpriced EVM opcodes invoking costly Move logic
   - **MWC-114** — Discrepant gas metering between precompiled and interpreted paths
   - **MWC-115** — Inaccurate accounting in hybrid (Move $\rightleftarrows$ EVM) transaction batching
6. **F6. Framework Logic Errors and Unsafe Abstractions (FLA)**
   - **MWC-116** — Misuse of generics leading to unsound type instantiation
   - **MWC-117** — Unsafe use of standard libraries with hidden state assumptions
   - **MWC-118** — Failure to enforce resource invariants in public API exposure
   - **MWC-119** — Lack of runtime checks in generic capability wrappers

In addition to these primary six frames, we identify **supplementary categories** that emerge from MoveEVM's evolving runtime model:

1. **Formalism Gaps and Verification Failures**

- **MWC-120** — Move Prover incompleteness on EVM ABI-conformant entrypoints
- **MWC-121** — Undetected post-condition failures under EVM fallback dispatch

2. **Tooling and Compiler-Generated Risks**
   - **MWC-122** — Compiler-injected unsafe access to global storage
   - **MWC-123** — Inconsistent error propagation in compiled bytecode
   - **MWC-124** — Move-EVM compiler linking errors introducing dangling capabilities
3. **Hybrid Standard Violations**
   - **MWC-125** — Deviation from expected ERC/MIP compatibility in hybrid contracts
   - **MWC-126** — ABI serialization violating Move resource expectations
   - **MWC-127** — Duplicate module address registration across environments
4. **Cryptographic Misuse in Context-Switching**
   - **MWC-128** — Inappropriate hash domain reuse between Move and EVM
   - **MWC-129** — Misconfigured key validation in dual-signer patterns
5. **Observable Side Effects and Leakage**
   - **MWC-130** — Emission of inconsistent event structures
   - **MWC-131** — Leakage of internal Move state via view functions
   - **MWC-132** — Use of 'panic'-like error reporting in observable contexts
6. **Environment/Bridge-Related Risks**
   - **MWC-133** — Bridge logic bypassing Move module access restrictions
   - **MWC-134** — Inconsistent state replication across L2 $\rightleftarrows$ L1 environments
   - **MWC-135** — Vulnerabilities arising from partial migration to zk-Move systems
   - **MWC-136** — Oracles injecting inconsistent state via unverified call patterns

4.3. Use Cases of the Frame-Based Taxonomy

This structured classification provides:

- **Security Audits:** A frame-driven vulnerability checklist for systematic coverage.
- **Tool Integration:** Ground truth for training fuzzers, LLM agents, and formal analyzers.
- **Language Design Feedback:** A roadmap for future-safe improvements in MoveEVM runtime and tooling.

4.4. Scope and Boundaries

This taxonomy targets MoveEVM contracts and hybrid runtime behaviors. It explicitly excludes:

- Cryptographic protocol flaws not tied to execution semantics.
- Front-end/UI-based exploits.
- Non-smart-contract attacks such as phishing or social engineering.

5. Explanatory code-based examples the MWC Categories for developers, readers and interested parties

MWC-100: Frozen Contract State

Description: Contract gets stuck in `frozen` state without an unfreeze option.
Vulnerable Code:

```
module Token {
    struct TokenData has key { frozen: bool }

    public fun freeze(token: &mut TokenData) {
        token.frozen = true;
    }

    public fun transfer(token: &TokenData) {
        //  No unfreeze or check mechanism
        assert(!token.frozen, 0);
    }
}
```

Fix: Add `unfreeze()` method or reversible FSM.

MWC-101: Uninitialized State

Description: Use of a state resource without checking existence first.
Vulnerable Code:

```
module Counter {
    struct State has key { count: u64 }

    public fun increment(addr: address) {
        let state = borrow_global<State>(addr)
            ; // May not exist
        state.count = state.count + 1;
    }
}
```

Fix: Use `exists<State>` check before borrow.

MWC-102: No Rollback Mechanism

Description: Partial failure leads to inconsistent state (withdraw without deposit).
Vulnerable Code:

```
public fun transfer(user: &signer, to: address
    , amount: u64) {
    withdraw(user, amount);
    //  If next step fails, withdraw already
        happened
    deposit(to, amount);
}
```

Fix: Ensure atomicity or use transactional patterns.

MWC-103: Infinite Loop

Description: Improper loop termination condition causing unbounded execution.

Vulnerable Code:

```
let mut i = 0;
while (i >= 0) {
    i = i + 1;
}
```

Fix: Ensure loop variable progresses toward a proper stopping condition.

MWC-104: External Call Without Validation

Description: Unvalidated module address can lead to unsafe external execution.

Vulnerable Code:

```
public fun call_external(mod: address) {
    //  No validation of external target
    External::trigger(mod);
}
```

Fix: Verify module address and call constraints.

MWC-105: Dead Code Execution

Description: Unreachable code included after return statement.

Vulnerable Code:

```
public fun transfer(amount: u64) {
    return;
    //  Unreachable code below still incurs
        deployment cost
    let x = amount + 1;
    log::info("Never executed");
}
```

Fix: Remove unreachable logic post-return.

MWC-106: Hybrid Reentrancy Between Move and EVM Modules

Description: Reentrancy vulnerabilities when EVM calls re-enter Move logic before the first execution completes. **Vulnerable Code:**

```
public fun transfer() {
    External::evm_callback(); // External EVM
        call can re-enter this Move function
        before state is updated

    update_balance();
}
```

Fix: Update internal state before external calls or apply a reentrancy guard.

MWC-107: Callback-based State Mutation Violating Linearity

Description: Unsafe state mutation through callback functions, violating Move's resource guarantees. **Vulnerable Code:**

```
public fun call_with_callback(callback:
    address) {
    callback::trigger(); // Changes internal
        resource state and external call could
         modify the same state concurrently if
         not locked

}
```

Fix: Avoid modifying state in callbacks or isolate effects with capabilities.

MWC-108: Interleaved Writes to Move Storage via External Callouts

Description: External calls modifying shared storage in parallel causing inconsistencies. **Vulnerable Code:**

```
public fun update_state() {
    EVM::external_op(); // Interleaves with
        Move storage writes
    state.value = 10;
}
```

Fix: Lock or checkpoint critical storage prior to external execution.

MWC-109: Inconsistent Ordering of Resource Locks

Description: Deadlocks or race conditions caused by acquiring locks in inconsistent order. **Vulnerable Code:**

```
lock_a();
lock_b();
// In another function: lock_b(); lock_a();
```

Fix: Always acquire resources in a consistent global order.

MWC-110: Unexpected Token Burn Due to Missing Checks

Description: Tokens are burned without verifying balances or permissions.
Vulnerable Code:

```
public fun burn(token: &mut
    Token) {
        token.total = token.
            total - 100;
}
```

Fix: Validate balance ownership and enforce burn conditions.

MWC-111: Unauthorized Token Minting

Description: Any caller can mint new tokens due to missing access control.
Vulnerable Code:

```
public fun mint() {
        supply = supply +
            1000;
}
```

Fix: Enforce capability-based or role-based access to mint functions.

MWC-112: Reused Signatures Across Heterogeneous Domains

Description: Same signature used in different chains or contract contexts. **Vulnerable Code:**

```
// Signature meant for chain A reused on chain
    B
```

Fix: Include chain IDs and contract addresses in message hashing.

MWC-113: Underpriced EVM Opcodes Invoking Costly Move Logic

Description: Attackers exploit gas cost imbalances to trigger expensive Move logic. **Vulnerable Code:**

```
EVM::cheap_call(); // Triggers heavy Move
    execution
```

Fix: Align gas estimation across layers or limit call depth.

MWC-114: Discrepant Gas Metering Between Precompiled and Interpreted Paths

Description: Gas discrepancy between Move and EVM bytecode paths leads to abuse. **Vulnerable Code:**

```
call_precompiled(); // Gas is not charged
    appropriately
```

Fix: Ensure uniform gas rules for all code paths.

MWC-115: Inaccurate Accounting in Hybrid Transaction Batching

Description: Multiple operations batched but only partially accounted in gas or execution state. **Vulnerable Code:**

```
batch_ops([op1, op2]); // Partial success
    untracked
```

Fix: Make batches atomic or explicitly track partial outcomes.

MWC-116: Misuse of Generics Leading to Unsound Type Instantiation

Description: Incorrect use of generics permits invalid types at runtime. **Vulnerable Code:**

```
store<T>(item: T); // No constraints on T
```

Fix: Use type constraints and specification to restrict T.

MWC-117: Unsafe Use of Standard Libraries with Hidden State

Description: Importing libraries that manipulate hidden or undocumented global state. **Vulnerable Code:**

```
use Lib::*; // May affect global state
```

Fix: Inspect library effects or avoid stateful imports.

MWC-118: Failure to Enforce Resource Invariants in Public API Exposure

Description: Exposing public APIs without checking internal resource constraints. **Vulnerable Code:**

```
public fun issue(token: Token) {
    // No validation of capabilities
}
```

Fix: Validate invariants before executing public logic.

MWC-119: Lack of Runtime Checks in Generic Capability Wrappers

Description: Wrappers around capabilities fail to validate runtime behavior. **Vulnerable Code:**

```
wrap(cap); // Wraps without validating
    permission
```

Fix: Add runtime checks for wrapper inputs and usage.

MWC-120: Weak Signature Verification

Description: Signature verification lacks nonce or domain separation, allowing replay.
Vulnerable Code:

```
public fun exec(sig: vector<u8
    >, msg: vector<u8>) {
        crypto::verify(pubkey,
            msg, sig); //
           Missing nonce/
           context
}
```

Fix: Add nonce and domain-specific prefixes before signing/verifying messages.

MWC-120: Move Prover Incompleteness on EVM ABI-Conformant Entrypoints

Description: Formal tools fail to verify entrypoints when ABI encoding bypasses assumptions. **Vulnerable Code:**

```
public fun unsafe_entry(input: vector<u8>) {
    let decoded = abi::decode(input); // Skips
        Move Prover assumptions
}
```

Fix: Use explicit preconditions and constrain decoding logic with specifications.

MWC-121: Undetected Post-Condition Failures under EVM Fallback Dispatch

Description: Contracts behave incorrectly when fallback mechanisms skip Move post-conditions. **Vulnerable Code:**

```
fallback fun handle() {
    transfer(); // No check of post-conditions
}
```

Fix: Enforce post-conditions manually or avoid using fallbacks for critical logic.

MWC-122: Compiler-Injected Unsafe Access to Global Storage

Description: Auto-generated code accesses global state without appropriate checks. **Vulnerable Code:**

```
// Compiler-generated read of global<T>(addr)
```

Fix: Use explicit storage guards and validate code generated by compiler macros.

MWC-123: Inconsistent Error Propagation in Compiled Bytecode

Description: Errors are silently dropped or inconsistently returned in compiled code. **Vulnerable Code:**

```
call_internal(); // Error not bubbled up
```

Fix: Standardize error handling patterns and enforce error return on all paths.

MWC-124: Move-EVM Compiler Linking Errors Introducing Dangling Capabilities

Description: Linking across modules creates unreferenced capabilities or permissions. **Vulnerable Code:**

```
// Capability returned without linkage
    validation
```

Fix: Resolve and audit all link-time dependencies manually.

MWC-125: Predictable Execution Order Enabling MEV

Description: Miner or attacker reorders transactions to exploit a contract's predictable execution pattern.
Vulnerable Code:

```
public fun submit_bid(bid: u64
    ) {
        if (bid >
            current_highest) {
            current_highest
                = bid;
            winner =
                signer::
                address_of
                (
                tx_context
                ::sender()
                );
        }
}
```

Fix: Use commit-reveal patterns or delay finalization to mitigate MEV opportunities.

MWC-126: ABI Serialization Violating Move Resource Expectations

Description: ABI encoding hides critical resource information leading to safety violations. **Vulnerable Code:**

```
abi::decode_resource(input); // No type-safe
    decode
```

Fix: Avoid ABI resource transfers or verify structure manually post-decode.

MWC-127: Duplicate Module Address Registration Across Environments

Description: Move modules deployed under conflicting addresses in multichain contexts. **Vulnerable Code:**

```
// Same module at 0x1 in devnet and testnet
```

Fix: Enforce globally unique module identities or use namespaces.

MWC-128: Inappropriate Hash Domain Reuse Between Move and EVM

Description: Same hash domain used across layers allows cross-environment collisions. **Vulnerable Code:**

```
let h = hash::sha3_256(msg); // Domain not
    distinguished
```

Fix: Prefix all hashed messages with unique domain identifiers.

MWC-129: Misconfigured Key Validation in Dual-Signer Patterns

Description: Contracts with multiple signers fail to validate key ownership properly. **Vulnerable Code:**

```
verify(k1, msg) && verify(k2, msg); // But k2
    not authorized
```

Fix: Enforce signer roles and permission explicitly in validation logic.

MWC-130: Emission of Inconsistent Event Structures

Description: Events vary in structure across versions, breaking indexers or observers. **Vulnerable Code:**

```
emit Transfer(from, to, amount); // Missing
    metadata
```

Fix: Standardize event formats and version them explicitly.

MWC-131: Leakage of Internal Move State via View Functions

Description: Public view functions expose sensitive internal state. **Vulnerable Code:**

```
public fun view_config() returns (Config) {
    config
}
```

Fix: Sanitize returned data and avoid exposing raw structs.

MWC-132: Use of 'panic'-Like Error Reporting in Observable Contexts

Description: Emitting panics or aborts leaks internal conditions to observers. **Vulnerable Code:**

```
assert(balance > 0, 42); // Code 42 reveals
    logic flow
```

Fix: Use generic error codes and do not encode logic semantics in abort values.

MWC-133: Bridge Logic Bypassing Move Module Access Restrictions

Description: Bridges inject data or calls that override access restrictions. **Vulnerable Code:**

```
Bridge::send(payload); // Payload contains
    privileged operation
```

Fix: Re-validate all inputs and restrict bridge-entry modules.

MWC-134: Inconsistent State Replication Across L2-L1 Environments

Description: L2 and L1 states drift due to asynchronous replication logic. **Vulnerable Code:**

```
state_l2 = fetch(); // But not committed to L1
```

Fix: Use commit/confirm pattern with state proofs across layers.

MWC-135: Vulnerabilities from Partial Migration to zk-Move Systems

Description: Partially migrated contracts break assumptions in zk-runtime. **Vulnerable Code:**

```
// zk-incompatible resource behavior
```

Fix: Migrate atomically or wrap legacy logic in zk-safe proxies.

MWC-136: Oracles Injecting Inconsistent State via Unverified Calls

Description: Oracles introduce unvalidated or manipulated state into contracts. **Vulnerable Code:**

```
price = Oracle::get(); // No signature or
    source check
```

Fix: Validate oracle source via multi-sig, timestamp, or proof.

6. Comparison of SWC and MWC Taxonomies

Both the proposed **MWC (MoveEVM Weakness Classification)** taxonomy and the **SWC (Smart Contract Weakness Classification)** registry offer structured categorization schemes for smart contract vulnerabilities. However, as Table .1 in the Appendix section shows, they vary greatly in terms of their origin, scope, and contextual relevance.

The Ethereum community maintains the SWC registry, which is built around *Ethereum Virtual Machine (EVM)* and *Solidity language*. It focuses on both EVM-specific problems like misuse of *delegatecall* and transaction-ordering dependencies, as well as common programming errors like integer overflows and unhandled exceptions.

The MWC taxonomy, on the other hand, is designed for the *hybrid MoveEVM environment*, where the strict type system and resource-oriented programming model of Move interact with

the semantics of EVM bytecode. Traditional SWC codes are unable to identify new vulnerability surfaces brought about by this fusion, such as cross-module resource leakage and ABI deserialization mismatches.

Tools like Mythril and Slither frequently use SWC, a flat listing of more than 100 categories (e.g., SWC-100: Function Visibility, SWC-110: Assert Violation). In contrast, MWC defines 37 vulnerability codes (MWC-100 to MWC-136), which are categorized into six high-level categories, such as *Meta-Transaction and Signature Spoofing (MTS)* and *Bytecode Model Inconsistencies (BMI)*.

MWC can better represent the semantic complexity of hybrid execution paths and conform to new toolchains that support both Move and EVM codebases thanks to this structural grouping.

Some vulnerability types, like *SWC-101 (Integer Overflow)* and *MWC-101 (Numeric Edge Conditions)*, are present in both taxonomies; however, SWC does not include representation for important Move-specific issues, like *MWC-110 (Linear Resource Violations)* or *MWC-125 (Hybrid Gas Semantics Mismatch)*. On the other hand, vulnerabilities that stem solely from Solidity constructs, like `fallback` misuse, are not covered by MWC.

SWC is widely used in educational materials, IDEs, and static analyzers with a Solidity focus. Although MWC is more recent, it is becoming more and more incorporated into MoveEVM-adapted tools (such as MoveScan and Smartify LLM agents), and because Move places a strong emphasis on accuracy and safety, it is consistent with formal verification frameworks.

Although SWC is still necessary for conventional EVM-based systems, it is not expressive enough to capture MoveEVM's hybrid semantics. By offering an organized, empirically derived framework specifically designed for the MoveEVM context, MWC closes this gap. In cross-VM audit environments and tooling pipelines, both taxonomies can coexist and enhance one another.

7. Case Studies

We examined real-world smart contracts that were implemented on well-known Move-based networks, such as **Aptos** and **Sui**, in order to validate the suggested taxonomy and demonstrate its applicability. Public disclosures, reproducible proofs-of-concept, or availability in open-source repositories were the criteria used to choose each case study.

7.1. Case Study 1: Sui-based Lending Protocol Invariant Violation

A resource mismanagement flaw in a Sui-based lending contract made it possible for the same token resource to be claimed repeatedly across modules. This vulnerability was classified as **MWC-112 (Cross-Module Resource Leakage)**. Inadequate boundary enforcement between modules that exposed a shared borrow function without adequate checks on token ownership constraints was the main contributing factor.

7.2. Case Study 2: Aptos-EVM Bridge ABI Deserialization Mismatch

External contracts were able to create distorted payloads that passed Move verification but resulted in corrupted resource states in a hybrid Aptos-EVM bridge due to a serialization mismatch. This was mapped to **MWC-102 (ABI Deserialization Flaw)**. The flaw made clear the dangers of invoking Move functions without sufficient validation using EVM-style calldata.

7.3. Case Study 3: Modular MoveEVM Chain Meta-Transaction Replay

Replay attacks resulted from a meta-transaction implementation that reused Ethereum-style signatures without the necessary domain separation or nonce checks. This issue, which was classified under **MWC-120 (Signature Replay Without Nonce)**, highlighted the necessity of domain isolation and cryptographic hygiene in hybrid execution environments.

7.4. Summary of Findings

We found **21 distinct vulnerabilities** across 42 reviewed contracts across different MoveEVM-enabled environments (such as Aptos, Sui, and hybrid bridges), of which **11 directly aligned with newly proposed MWC categories** that are not captured by the traditional SWC registry. This supports the central claim of this paper, which is that the hybrid execution model of MoveEVM presents new and non-trivial vulnerability surfaces that go beyond the purview of current EVM-centric taxonomies.

The uncovered vulnerabilities were disproportionately concentrated in the following frames:

- **Bytecode Model Inconsistencies (BMI):** covering four vulnerabilities, especially those pertaining to unsafe raw payload handling and ABI encoding/decoding discrepancies.
- **State Reentrancy and Synchronization (SRS):** Five flaws were discovered in contracts that used unsynchronized resource mutation logic and cross-runtime callbacks.
- **Meta-Transaction and Signature Spoofing (MTS):** When domain separation, replay protection, or cryptographic nonce hygiene were lacking, three problems were noted.

Crucially, current static analysis tools like Slither, Mythril, or the Move Prover, which are made for Solidity or pure Move environments, were unable to identify eight of the eleven MWC-classified vulnerabilities. Future development of hybrid-aware auditing and verification frameworks must address this significant *tooling gap*.

Furthermore, trends showing systemic risks surfaced in:

- Type safety and capability enforcement are frequently circumvented in the bridge and interoperability layers.
- **Gas accounting mismatches,** which lead to unpredictable execution costs and security regressions in cross-language batching.

- **Standard library imports,** which frequently assume safe defaults but introduce hidden state transitions in composable modules.

These findings not only validate the necessity of the MWC taxonomy but also demonstrate its utility as a practical framework for identifying and categorizing emergent smart contract vulnerabilities in next-generation blockchain platforms. The classification system provides a granular lens through which developers and auditors can detect, mitigate, and prevent sophisticated exploits that span both Move and EVM semantics.

Future studies may build upon these results by conducting large-scale, longitudinal vulnerability scans across MoveEVM projects to quantify evolving trends and to further refine or expand the taxonomy based on observed exploitability patterns.

8. Real-World Deployment Case Studies

We examined a number of publicly available MoveEVM-based contracts and platforms on chains like Aptos, Sui, and MoveVM-enabled testnets in order to validate our taxonomy and identify useful security vulnerabilities.

8.1. Aptos NFT Mint Vulnerability (MWC-102, MWC-112)

Because of insufficient resource validation and cross-module boundary interactions, the vulnerabilities reported in Mitenkov et al. (2024) during NFT minting on Aptos can be classified under our suggested taxonomy as MWC-102 and MWC-112.

8.2. MWC-112 (Module Boundary Escapes) and MWC-111 (Capability Verification) for Dex Exploit

A Sui-based decentralized exchange (DEX) (Blokworks (2025)) was found to have a flaw in which the withdrawal logic failed to enforce single-use constraints on capability objects. This allowed an attacker to reuse the same capability multiple times, violating Move's resource linearity in practice. This vulnerability is now classified under MWC-111 and MWC-112.

8.3. Bridge Adapter Reentrancy in MoveEVM (MWC-130)

We examined a MoveEVM-to-EVM bridge adapter prototype that permitted EVM calls to return to Move module logic prior to state commitments being fulfilled. Hybrid reentrancy and Move state manipulation were made possible by this cross-runtime callback, which falls under **MWC-130 (Hybrid Reentrancy)**.

These examples demonstrate how hybrid interactions, shoddy standard library patterns, or EVM compatibility layers can jeopardize Move's theoretical guarantees.

9. Logic-Driven AI Agents for MoveEVM Auditing

New capabilities for smart contract security are provided by recent developments in AI agent pipelines and logic-driven language models (LLMs), particularly in hybrid environments like MoveEVM. Compared to conventional tools, logic-guided systems audit contracts more successfully by utilizing prompt engineering, symbolic reasoning, and agent modularity.

9.1. Multi-Agent Architectures

A multi-agent framework for analyzing Move and EVM hybrid contracts using specialized LLMs was introduced by Smartify Karanjai et al. (2025). Each agent focuses on a particular kind of vulnerability, like asset leakage or reentrancy, and collaborates to generate potential fixes. Without depending on extensive Move-specific corpora, these modular agents are customized to the semantics of MoveEVM and reason using security specifications.

In a similar vein, LLM-SmartAudit Wei and Kumar (2024) showed that a collaborative architecture was superior to static analyzers in identifying logic errors in contracts with multiple languages. By designating MWC categories as analysis tasks for specific agents, these systems conform to the MWC framework.

9.2. Structured Prompt Pipelines

Prompt pipelines are used in AuditGPT Xia et al. (2024) to demonstrate structured auditing. It breaks down auditing into phases: applying function-specific prompts after formal properties have been extracted. By matching each prompt to a MWC rule, this divide-and-conquer strategy improves explainability and precision while adhering to MWC categorization. Per-MWC-category prompting can be used in future MoveEVM pipelines to provide more precise diagnosis.

9.3. Natural-Language and Proof-Based Reporting

LLMs can function as "proof scribes," producing specifications and proofs from source code, as PropertyGPT (Liu et al. (2024)) showed. Theorem provers are used to verify the invariant suggestions that these systems convert Move contracts into. Both Smartify and AuditGPT produce reports in natural language for human auditors, offering automated code fixes or MWC-aware diagnostics.

10. Formal Verification Tools and MWC Taxonomy

10.1. MoveProver and SMT-Based Invariant Checking

The Move language's formal verifier is called MoveProver Bartoletti et al. (2025). It achieves high scalability by using SMT solving (Z3) to verify global invariants and pre/postconditions (e.g., full verification of the 8,800-line Diem framework). MWC vulnerabilities like resource leaks, absent access checks, or invariant violations can be statically discharged by MoveProver.

10.2. Model Checking and K-Framework Applications

VeriMove Keilty et al. (2022) uses NuSMV and CTL specifications to model-check Move. Beyond function-local proofs, it allows cross-function invariant enforcement and validation of temporal properties. This method can be applied to MoveEVM to validate cross-module behaviors in the MWC style.

10.3. Symbolic Execution and Property Specifications

Move's specification language allows for expressive assertions, despite the fact that its symbolic execution tools are limited. Move's linear memory and resource model makes it easier to prove class-level MWC properties like ownership safety or no-loss guarantees than Solidity's Certora or Slither Bartoletti et al. (2025). This suggests that a large number of MWC rules can be statically confirmed without the need for sophisticated auxiliary tools.

11. Future Directions in AI-Augmented MoveEVM Auditing

While still essential, traditional auditing methods are increasingly being supplemented by artificial intelligence, especially Large Language Models (LLMs) and logic-driven agents, as MoveEVM-based smart contracts become more complex and widely used. In order to create a security analysis framework that is more scalable, accurate, and automated for MoveEVM environments, this section examines promising research and development avenues that combine formal verification techniques with AI-driven tools.

11.1. LLMs as Theorem-Proving Assistants

According to recent research, LLMs can help theorem provers by summarizing failed invariants or converting counterexamples into fixes. As a copilot for MoveProver, GPT-4 could rank vulnerable code paths by MWC severity, explain Z3 outputs, and offer fixes.

Recent developments suggest that Large Language Models (LLMs) like Claude and GPT-4 can serve as intelligent theorem-proving assistants to enhance the performance of formal verifiers. These models can translate abstract counterexamples into useful developer instructions and interpret and explain outputs from SMT solvers like Z3, which are used internally by tools like MoveProver, in the context of Move smart contracts.

For example, if MoveProver is unable to release an invariant or post-condition, an LLM may:

- Examine the particular proof failure and find the line in the code that corresponds to it.
- Give an overview of the violated property in natural language.
- Make suggestions for changes to the code or different requirements that would guarantee the success of the proof.
- Rank issues based on MWC (Move Weakness Classification) severity levels.

11.2. Interactive Audit Pipelines

Prompt pipelines, inspired by AuditGPT, can separate tasks according to MWC categories: some query MoveProver for supporting invariants, while others detect the class (e.g., GSM or MTS). AutoGen or LangChain Langchain (2025); Wu et al. (2022, 2023) agents can be used to orchestrate such workflows.

Building interactive, category-specific audit pipelines, motivated by programs such as AuditGPT, is another fascinating avenue. These pipelines can divide auditing into a number of specific stages by utilizing prompt engineering and modular agent design:

- **Classification Stage:** The pipeline uses static indicators or semantic patterns to determine the general class of vulnerability, such as Meta-Transaction Spoofing (MTS), Bytecode Model Inconsistency (BMI), or Generalized State Manipulation (GSM).
- **Specification Extraction:** The system uses pre-defined templates linked to MWC categories or automatically extracts formal specifications or invariants from code.
- **Formal Verification Querying:** LLM agents generate hypotheses that are used to invoke MoveProver or VeriMove. Failed proofs are returned to the agent for correction and interpretation.
- **Generating Developer Feedback:** The system generates a human-readable description of the problem, the failed proof, and potential fixes.

Frameworks like LangChain, AutoGen, or PromptChainer Langchain (2025); Wu et al. (2022, 2023) could be used to orchestrate such modular pipelines. They successfully combine formal rigor with LLM adaptability, enabling scalable and explicable auditing across sizable codebases.

11.3. MWC-Based Benchmarks and Datasets

The absence of labeled Move vulnerability datasets is a major obstacle. Real contracts could be annotated by MWC category using a MoveEVM analog to SWC. These data sets can be used to standardize audit metrics across tools, assess agent accuracy, and improve LLMs.

The absence of annotated datasets specifically designed to address the special characteristics of Move's execution model is a significant obstacle to developing AI-augmented security tools for MoveEVM. There is currently no corpus of real-world contracts categorized by vulnerability type in MoveEVM, unlike Solidity, where supervised learning and benchmarking are made possible by SWC-tagged datasets.

We suggest developing an open-source benchmark suite comprising the following elements to get around this restriction:

- **Labeled Vulnerability Corpus:** Move contracts marked with their MWC category—e.g., MWC-106 for hybrid reentrancy, MWC-128 for cryptographic usage—are gathered here.
- **Fix Pairings:** For every vulnerable contract, a "fixed" variant and justification for the fix accompany it.
- **Formal Properties Dataset:** Formally defined and linked to every contract are specifications, pre/postconditions, and invariants.

- **Execution Traces:** Symbolic or literal traces showing attack situations and proof failures.

These datasets have several uses: fine-tuning LLMs on security-specific tasks, assessing prompt engineering strategies, training reinforcement learning agents for autoremediation, and creating shared metrics for tool comparison.

11.4. Toward Human-AI Co-Auditing Pipelines

Human-AI co-auditing is envisioned as creating cooperative settings whereby automated agents and human auditors cooperate. Under such systems, humans concentrate on contextual judgment, ambiguous logic, and final decision-making while AI agents handle high-throughput tasks including code scanning, classification, and proof validation.

Key elements of co-auditing pipelines might include:

- **Agent Responsibility Assignment:** Every LLM agent is given particular MWC categories or formal properties to track.
- **Real-Time Auditing Dashboards:** Interfaces that let developers interactively see agent recommendations, proof attempts, and vulnerability reports.
- **Feedback Loops:** Suggestions can be accepted, turned down, or changed by developers; the model adjusts depending on the responses (active learning).
- **Certification Support:**Verified results from AI+human co-audits can be included into audit reports or presented as machine-checkable certificates.

Especially as MoveEVM smart contracts get more sophisticated and widely used, this approach scales security expertise, lowers manual effort, and over time creates institutional knowledge.

11.5. Summary and Outlook

For auditing MoveEVM smart contracts, this vision combines formal methods and logical based LLMs. Structured prompts, agent roles, and verification targets find a semantic backbone in MWC categories. Future studies should investigate multi-agent orchestration, prompt-based audits per category, and human-AI co-auditing pipelines anchored in MWC reasoning. Integration of formal verification methods with LLM-powered assistants marks a paradigm change in the auditing of MoveEVM contracts. Aligning MWC categories with AI processes will help the community to reach more consistency, repeatability, and efficiency in vulnerability identification and remediating.

12. Conclusions, Recommendations, and Future Work

Comprising six main categories and 37 subtypes (MWC-100 to MWC-136), this paper proposed a disciplined taxonomy of MoveEVM vulnerabilities. This all-encompassing classification system is meant to systematize security evaluations across hybrid smart contracts, so offering a strong framework that improves the accuracy and efficiency of security assessments. Establishing separate categories helps us to guarantee that all stakeholders—auditors, developers, tool builders—are in agreement on possible vulnerabilities and their consequences, so facilitating better communication among them.

Apart from spotting weaknesses, the taxonomy facilitates the creation of customized mitigating techniques, so promoting a more safe MoveEVM ecosystem. Crucially as the terrain of smart contracts and blockchain technology keeps changing fast, the methodical character of this taxonomy enables it to evolve over time, adjusting to new threats and technological developments. Through classification of vulnerabilities, we have opened the path for the creation of uniform procedures applicable to several MoveEVM projects.

Based on our results, we advise developers of MoveEVM smart contracts to implement the following steps to improve their security:

- **Use Capabilities and Resource Guards Explicitly:** Developers should specifically enforce security limits using capabilities and resource guards. This proactive strategy reduces the risk connected with implicit type checks, which might cause unanticipated actions and weaknesses. Clearly defined ensures that every interaction altering state or access to resources enhances the general security posture of the application. Moreover, considering how every element interacts with others and the possible hazards connected with those interactions, developers should have a perspective that gives security top priority in the design process.
- **Avoid Cross-Module State Dependencies:** Designing contracts depending on state dependencies across modules calls for careful consideration from developers. Unless they are thoroughly tested and confirmed, such dependencies can bring complexity and raise vulnerability potential. To guarantee dependability, a comprehensive testing program should be developed whereby automated tests covering all pertinent interactions. This covers security-oriented tests modeled on several attack paths against the contract, integration tests, and unit tests.
- **Rely on Formal Specifications:** Designers of public interfaces are urged to define expected behaviors by depending on formal specs. Move Prover should be used to confirm compliance and correctness by means of these criteria. By means of formal verification, one can effectively identify discrepancies and possible weaknesses before implementation, so lowering the risk of exploitation in the living environment. Including formal techniques into the development process helps developers produce more dependable contracts following the given criteria.
- **Conduct Regular Security Audits:** Regular security audits should be given top priority by developers all through the process. Involving independent auditors guarantees

that possible weaknesses are found outside the immediate awareness of the development team and adds a new viewpoint. To give thorough coverage, audits should be iterative, following major code changes, and ideally include both static and dynamic analysis.

- **Adopt a Security-First Development Philosophy:** Every phase of the development life should include security issues into account by developers. This covers doing threat modeling during the design process, doing frequent code reviews with an eye toward security, and keeping current on the most recent vulnerabilities and attack paths that the MoveEVM system has revealed. Stressing a culture of security inside development teams will help to identify and reduce hazards early on.

Recommendations for Auditors

Auditors should follow these best practices to improve the MoveEVM contract auditing process:

- **Include MoveEVM-Specific Categories in Checklists:** Auditors have to include the MoveEVM-specific categories our taxonomy defines into their regular audit forms. This inclusion guarantees that the security evaluation process leaves no stone unturned by methodically assessing all possible weaknesses. Comprehensive audits covering not only the code but also the underlying architecture and interaction with outside systems.
- **Utilize Both Static and Dynamic Tools:** A Throughout their audits, auditors should combine dynamic testing tools with static analysis tools. Using both methods will help auditors to fully grasp the behavior of the contract and find weaknesses that might not be clear from static analysis by itself. Particularly useful in exposing runtime vulnerabilities that static analysis might overlook is dynamic testing—fuzzing.
- **Monitor for New Hybrid Behaviors:** Auditors should be alert as the MoveEVM platform and its underlying technologies change for new hybrid behaviors that might surface. This guarantees that auditing procedures remain pertinent and efficient by means of constant education and adaptation to new environmental changes. Participating in forums, seminars, and debates with the community will help one gain important understanding of newly developing hazards.
- **Engage in Knowledge Sharing:** Auditors should take part actively in blockchain and smart contract communities' seminars, conferences, and discussions. Auditors can add to the body of knowledge by sharing ideas and experiences, so promoting an attitude of ongoing auditing practice improvement. Furthermore fostering creativity in auditing techniques is cooperation with academic institutions and researchers.
- **Develop Incident Response Plans:** Working with companies, auditors should help them create and preserve incident response strategies that specify the actions to be followed should a security breach arise. These strategies for containment, research, communication, and remedial action should guarantee that companies are ready to react properly to weaknesses and attacks.

Based on empirical audits and recorded vulnerabilities as of 2025, our taxonomy points future directions. But as the field of smart contracts and blockchain technology develops, several future paths that call for more research surface.

- **Extending the Framework to zkEVM–Move Integrations and Rollups:** Extending our vulnerability taxonomy to include zkEVM–Move integrations will be essential as zero-knowledge technologies acquire popularity inside the larger blockchain ecosystem. This will entail spotting special weaknesses related to zero-knowledge proofs and rollups so that auditors and developers might better grasp the security consequences of these technologies. Research should concentrate on how zero-knowledge proofs interact with current security paradigms and how these interactions might be securely carried out.
- **Automating Vulnerability Classification Using LLM-Based Tools:** Large language models' (LLMs') integration into security tools offers a chance to automatically classify vulnerabilities. Training these models on our repository of vulnerabilities and their traits will help us to build systems that quickly find and classify fresh vulnerabilities in MoveEVM contracts, so improving accuracy and efficiency in security evaluations. The evolution of these automated tools should also take into account the interpretability of model outputs to guarantee developers may rely on the recommendations given by artificial intelligence systems.
- **Standardizing MoveEVM ABI and Gas Accounting:** Establishing a standardized Application Binary Interface (ABI) for MoveEVM along with consistent gas accounting methods is crucial to enable interoperability and lower differences between several runtimes. By means of consistency, this standardization will help to reduce security concerns resulting from discrepancies and guarantee a better development experience for programmers operating inside the Move ecosystem. It will also offer better instructions for auditors, so simplifying audits conducted on several systems.
- **Collaborative Development of Tools and Frameworks:** To create tools leveraging the taxonomy for automated vulnerability detection and remediation, we urge cooperation among researchers, developers, and auditors. The community can develop strong answers that meet the changing security needs of MoveEVM contracts by aggregating resources and knowledge. Encouragement of open-source projects will help to promote contributions from many different stakeholders, so improving the variety and strength of the tools at disposal.

- **Contribute to the Community Knowledge Base:** We urge the MoveEVM community to provide additional case studies, tool evaluations, and research results that might enable the taxonomy to be developed into a useful guide for MoveEVM security. Through knowledge and experience sharing, practitioners can improve the security environment of MoveEVM smart contracts together and help to create a more safe blockchain ecosystem. By means of a centralized repository for best practices and case studies, knowledge sharing and ongoing development can be promoted.
- **Conduct Longitudinal Studies on Security Trends:** Longitudinal studies tracking the development of vulnerabilities in MoveEVM smart contracts over time should form part of future work. Analyzing past data helps scientists spot trends, grasp the success of mitigating techniques, and project future vulnerabilities. More resilient smart contracts and more efficient auditing techniques will be developed in part by this data-driven approach.

Future research may focus on:

- Designing benchmark datasets and simulation environments tailored for MWC-based learning.
- Extending tools like MoveProver and VeriMove with LLM backends.
- Experimenting with fully autonomous audit agents in testnets.
- Evaluating the reliability and false positive rates of AI-generated proofs and fixes.

The ultimate aim is to build an ecosystem in which MWC categories not only as classification labels but also as building blocks for automated reasoning, agent training, and secure-by-construction development. By means of ongoing investment in these domains, the auditing and validation of MoveEVM smart contracts can become not only more scalable but also more intelligent, trustworthy, and easily available.

Ultimately, the suggested taxonomy is a basic first step toward enhancing MoveEVM smart contract security. Implementing the suggestions advised above and actively supporting group projects will help developers and auditors greatly improve the resilience of their applications against new risks, so building more confidence in the MoveEVM system. All stakeholders must pledge to a culture of security-first thinking, ongoing education, and community collaboration as we progress, so building the foundation for a safer blockchain future.

Acknowledgements

The author ...

References

Abrahimi, N., 2023. Applying a modular execution environment with movevm in a blockchain-agnostic. Thesis .

Antonopoulos, A.M., Wood, G., 2018. Mastering Ethereum. O'Reilly Media.

Atzei, N., Bartoletti, M., Cimoli, T., 2017. A survey of attacks on ethereum smart contracts (sok). International Conference on Principles of Security and Trust , 164–186.

Bartoletti, M., Carta, S., Cimoli, T., Serusi, S., 2020. The defi ecosystem: Challenges, opportunities and vulnerabilities. arXiv preprint arXiv:2002.08099 .

Bartoletti, M., Crafa, S., Lipparini, E., 2025. Formal verification in solidity and move: insights from a comparative analysis. arXiv preprint arXiv:2502.13929 .

Bauer, D.P., 2022. Solidity, in: Getting Started with Ethereum: A Step-by-Step Guide to Becoming a Blockchain Developer. Springer, pp. 13–16.

Blackshear, S., Cheng, E., Dill, D.L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Rain, D.R., Sezer, S., et al., 2019. Move: A language with programmable resources. Libra Assoc 1.

Blokworks, 2025. Dex exploit. Available at: `https://blockworks.co/news/suis-decentralization-dex-put-to-the-te` Accessed: 2025-05-23.

Buterin, V., 2013. Ethereum whitepaper: A next-generation smart contract and decentralized application platform .

Buterin, V., 2018. Vyper documentation. Vyper by Example , 13.

Diem, 2019. Diem developers. `https://diem.github.io/move/introduction.html` Accessed: 2025-05-12.

Dill, D., Grieskamp, W., Park, J., Qadeer, S., Xu, M., Zhong, E., 2022. Fast and reliable formal verification of smart contracts with the move prover, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer. pp. 183–200.

Facebook, 2019. Libra white paper. Available at: `https://developers.libra.org/docs/assets/papers/the-libra-blockcha`

Feist, J., Grieco, G., Groce, A., 2019a. Slither: a static analysis framework for smart contracts, in: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), IEEE. pp. 8–15.

Feist, J., Grieco, G., Groce, A., 2019b. Slither: a static analysis framework for smart contracts, in: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), IEEE. pp. 8–15.

Foundation, E., a. Erc-20 token standard. `https://eips.ethereum.org/EIPS/eip-20`.

Foundation, E., b. Erc-721 non-fungible token standard. `https://eips.ethereum.org/EIPS/eip-721`.

Fu, Y., Ren, M., Ma, F., Yang, X., Shi, H., Li, S., Liao, X., 2024. Evmfuzz: Differential fuzz testing of ethereum virtual machine. Journal of Software: Evolution and Process 36, e2556.

Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A., 2020. Echidna: effective, usable, and fast fuzzing for smart contracts, in: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp. 557–560.

HackenProof, 2023. Sui vulnerability: Crafted bytecode causes node crash. `https://hackenproof.com/reports/sui-node-crash-move`. Accessed: 2025-05-12.

He, W., Xia, Y., Zhang, Z., Wang, W., 2020. Vslc: Automatically identifying logic vulnerabilities in smart contracts. ACM Transactions on Software Engineering and Methodology .

Karanjai, R., Blackshear, S., Xu, L., Shi, W., 2025. A multi-agent framework for automated vulnerability detection and repair in solidity and move smart contracts. arXiv preprint arXiv:2502.18515 .

Keilty, E., Nelaturu, K., Wu, B., Veneris, A., 2022. A model-checking framework for the verification of move smart contracts, in: 2022 IEEE 13th International Conference on Software Engineering and Service Science (ICSESS), IEEE. pp. 1–7.

Labs, M., 2022a. Sui white paper. `https://docs.sui.io/paper/sui.pdf`.

Labs, N.C., 2022b. Aptos vulnerability report: Move vm denial of service. https://www.numencyber.com/analysis-of-the-first-critical-vulnerability-of-aptos-move-vm/. Accessed: 2025-05-12.

Langchain, 2025. Langchain. Available at: https://python.langchain.com/docs/introduction/. Accessed: 2024-9-12.

Li, W., Bu, J., Li, X., Peng, H., Niu, Y., Zhang, Y., 2022. A survey of defi security: Challenges and opportunities. Journal of King Saud University-Computer and Information Sciences 34, 10378–10404.

Liu, Y., Xue, Y., Wu, D., Sun, Y., Li, Y., Shi, M., Liu, Y., 2024. Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. arXiv preprint arXiv:2405.02580 .

Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A., 2016. Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM. pp. 254–269.

Mitenkov, G., Kabiljo, I., Li, Z., Spiegelman, A., Vusirikala, S., Xiang, Z., Zlateski, A., Lopes, N.P., Gelashvili, R., 2024. Deferred objects to enhance smart contract programming with optimistic parallel execution. arXiv preprint arXiv:2405.06117 .

Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A., 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE. pp. 1186–1189.

Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A., 2018. Finding the greedy, prodigal, and suicidal contracts at scale, in: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 653–663.

Patrignani, M., Blackshear, S., 2023. Robust safety for move, in: 2023 IEEE 36th Computer Security Foundations Symposium (CSF), IEEE. pp. 308–323.

Pierro, G.A., Ibba, G., Tonelli, R., 2023. A study on diem and aptos distributed ledger technology. International Journal of Parallel, Emergent and Distributed Systems , 1–17.

Praitheeshan, P., Pan, L., Zheng, X., Jolfaei, A., Doss, R., 2021. Solguard: Preventing external call issues in smart contract-based multi-agent robotic systems. Information Sciences 579, 150–166.

Sayeed, S., Marco-Gisbert, H., Caira, T., 2020. Smart contract: Attacks and protections. Ieee Access 8, 24416–24427.

Security, Z., 2023a. Move fast and break things, aptos. https://www.zellic.io/blog/move-fast-and-break-things-pt-1. Accessed: 2024-11-12.

Security, Z., 2023b. Move fast and break things, sui. https://www.zellic.io/blog/move-fast-break-things-move-security-part-2. Accessed: 2024-11-12.

Sharma, N., Sharma, S., 2022. A survey of mythril, a smart contract security analysis tool for evm bytecode. Indian J Natural Sci 13, 39–41.

Song, S., Chen, J., Chen, T., Luo, X., Li, T., Yang, W., Wang, L., Zhang, W., Luo, F., He, Z., et al., 2024. Empirical study of move smart contract security: Introducing movescan for enhanced analysis , 1682–1694.

Songsom, N., Werapun, W., Suaboot, J., Rattanavipanon, N., 2022. The swc-based security analysis tool for smart contract vulnerability detection, in: 2022 6th International Conference on Information Technology (InCIT), IEEE. pp. 74–77.

Soud, M., Liebel, G., Hamdaqa, M., 2024. A fly in the ointment: an empirical study on the characteristics of ethereum smart contract code weaknesses. Empirical Software Engineering 29, 13.

swcregistry.io, 2024. Swc registry: Smart contract weakness classification and test cases. https://swcregistry.io/. Accessed: 2024-04-12.

Szabo, N., 1997. Formalizing and securing relationships on public networks. First Monday 2.

Torres, C., Schütte, J., State, R., McLaughlin, K., 2021. The art of the scam: Demystifying honeypots in ethereum smart contracts. USENIX Security .

Tsankov, P., Dan, A., Drachsler-Cohen, D., et al., 2018. Securify: Practical security analysis of smart contracts, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 67–82.

Wei, Z., Kumar, N., 2024. Llm-smartaudit: Collaborative multi-agent contract analysis. arXiv preprint arXiv:2404.00689 .

Werner, S.M., Perez, D., Gudgeon, L., Klages-Mundt, A., Knottenbelt, W., Livshits, B., 2021. Sok: Decentralized finance (defi), in: IEEE Euro S&P.

Wood, G., 2014. Ethereum: A secure decentralised generalised transaction ledger Ethereum Yellow Paper.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., Wang, C., 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. arXiv preprint arXiv:2308.08155 3.

Wu, T., Jiang, E., Donsbach, A., Gray, J., Molina, A., Terry, M., Cai, C.J., 2022. Promptchainer: Chaining large language model prompts through visual programming, in: CHI Conference on Human Factors in Computing Systems Extended Abstracts, pp. 1–10.

Wu, X., Xing, J., Li, X., 2025. Exploring vulnerabilities and concerns in solana smart contracts. arXiv preprint arXiv:2504.07419 .

Xia, S., Shao, S., He, M., Yu, T., Song, L., Zhang, Y., 2024. Auditgpt: Auditing smart contracts with chatgpt. arXiv preprint arXiv:2404.04306 .

Zhang, X., Li, Y., Sun, M., 2020. Towards a formally verified evm in production environment, in: Coordination Models and Languages: 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15–19, 2020, Proceedings 22, Springer. pp. 341–349.

Zheng, Z., Xie, S., Dai, H.N., Chen, X., Wang, H., 2020. Blockchain challenges and opportunities: A survey. International Journal of Web and Grid Services .

Zhong, J.E., Cheang, K., Qadeer, S., Grieskamp, W., Blackshear, S., Park, J., Zohar, Y., Barrett, C., Dill, D.L., 2020. The move prover.

Table .1: Comparison of SWC and MWC Taxonomies

Aspect	SWC (Solidity/EVM)	MWC (MoveEVM)
Purpose	Classify common vulnerabilities in Solidity smart contracts on EVM	Frame-based classification of MoveEVM-specific vulnerabilities in hybrid Move+EVM environments
Granularity	Moderate: 36 categories with varying detail	High: 37 detailed weakness codes mapped to 6 semantic frames
Language Model Assumptions	Stack-based execution, no linear type enforcement	Resource-oriented with linear type system and deterministic behavior
Type of Execution Environment	EVM-only; assumes Solidity bytecode behavior	Hybrid (EVM front-end with Move core semantics)
Signature Handling	Assumes `msg.sender` and `msg.value` based flows	Explicit signature verification; supports multi-signer and domain-separated signatures
Gas Model	Native EVM gas model; well-studied	Dual-layer gas semantics between EVM opcodes and Move logic
Reentrancy Modeling	Classical reentrancy on `call`, `delegatecall` etc.	Hybrid reentrancy via callbacks between EVM $\leftrightarrow$ Move modules (MWC-106 to MWC-109)
Module System	Monolithic; modularity via libraries or interfaces	Strongly modular, formalized resource encapsulation with inter-module invariants (MWC-103–105)
Tool Coverage	Broad coverage via MythX, Slither, Manticore	Emerging support in MoveScan, Move Prover, LLM-based agents; partial tool gaps identified
Cryptographic Context Awareness	Mostly assumes Solidity-level cryptographic primitives	Cryptographic misuse across hybrid environments (MWC-128–129) addressed explicitly
Meta-Transaction Support	Covered under replay attacks, but abstracted	Dedicated frame for Meta-Tx and signature spoofing (MWC-110–112)
Standard Compatibility	Anchored in ERC standards (e.g., ERC-20, ERC-721)	Considers compatibility with both ERC and MIP (Move Improvement Proposals) standards
Specification Awareness	Not explicitly tied to formal verification tools	Integrated with Move Prover and formal postcondition violations (MWC-120–121)
Targeted Projects	Ethereum mainnet, L1/L2 rollups using Solidity	MoveEVM chains (e.g., Aptos EVM, SuiEVM, zkMove)
Unique Contributions	Provides foundational classification for Solidity audit tooling	First structured hybrid vulnerability taxonomy accounting for EVM-Move execution intersections