Formal Analysis of Arithmetic Vulnerabilities in EVM Smart Contracts: Exploits and Mitigation Strategies

Author: Dr. Franciny Salles Rojas Marin

Date: September 23, 2025

Abstract

Smart contracts, particularly within the Ethereum ecosystem, form the backbone of decentralized finance (DeFi) applications, managing digital assets of substantial value. The correctness and security of these programs are therefore of critical importance. Arithmetic vulnerabilities, such as integer overflow and underflow, represent a class of programming flaws that, while subtle, can lead to catastrophic consequences, including unauthorized token creation and fund drainage. This whitepaper presents a formal and in-depth analysis of these vulnerabilities in the context of the Ethereum Virtual Machine (EVM). Using a state transition system model, we dissect the fundamental anatomy of an overflow exploit. We analyze in depth the historical case of the batchOverflow exploit (CVE-2018-10299) in the Beauty Chain (BEC) token, mathematically demonstrating how EVM's modular arithmetic was exploited. Finally, we conduct a comparative study of mitigation strategies, ranging from compile-time solutions such as SafeMath libraries and native Solidity v0.8+ compiler checks, to advanced formal verification methodologies such as Symbolic Execution and Bounded Model Checking (BMC). The objective of this work is to provide developers, auditors, and researchers with a rigorous understanding of these vulnerabilities and a framework for building more robust and secure smart contracts.

1. Introduction

The rise of blockchain technology, driven by Ethereum, introduced the concept of smart contracts — autonomous and deterministic programs that execute on a replicated and decentralized state machine, the Ethereum Virtual Machine (EVM) [1]. These contracts govern the logic of decentralized applications (DApps) that already move billions of dollars, making their security a primary concern. However, the immutability of code deployed on the blockchain means that security flaws, once exploited, can have permanent and devastating financial consequences.

One of the most persistent and dangerous classes of vulnerabilities in smart contracts is arithmetic errors, specifically integer overflow and underflow. These flaws occur when a mathematical operation results in a value that exceeds the storage space of the data type, causing the value to "wrap around" silently in older versions of the Solidity compiler. Attackers exploit this predictable behavior to subvert contract logic, bypass security checks, and manipulate contract state in unintended ways, such as minting an infinite amount of tokens.

This paper offers a rigorous technical analysis of these vulnerabilities. We begin with the formalization of arithmetic in the EVM and model how an overflow violates contract invariants. We then conduct a detailed case study of the batchOverflow exploit (CVE-2018-10299), which serves as a canonical example of the real-world impact of these attacks [2]. Finally, we evaluate a spectrum of defense techniques, from secure coding practices to the application of formal verification methods such as Bounded Model Checking [3], which offer a higher level of security assurance. This work aims to deepen the community's knowledge, promoting the development of more secure and resilient DApps.

2. Formal Model of Arithmetic and Vulnerabilities in the EVM

The Ethereum Virtual Machine (EVM) operates with fixed-size integer data types, with `uint256` (256-bit unsigned integer) being the most common for representing monetary values and balances. Arithmetic in the EVM for these types is modular. For a `uintN`, operations are performed modulo 2^N. This means that for a `uint256`, any result of an operation that exceeds `2^256 - 1` (the maximum representable value) will undergo a wrap-around.

Formally, for two `uint256` values *a* and *b*, addition is defined as:

`a + b := (a + b) mod 2^256`

And multiplication as:

`a * b := (a * b) mod 2^256`

An **integer overflow** occurs when `a + b >= 2^256` (for addition) or `a * b >= 2^256` (for multiplication). In Solidity versions prior to 0.8.0, this operation did not generate an exception, but rather resulted in the modular value, which was then used in subsequent contract logic. It is this discrepancy between the expected mathematical result and the actual computational result that creates the attack vector.

State Transition Model of the Vulnerability

We can model a smart contract as a State Transition System (STS), where state `S` is the set of all values stored in the contract's state variables (e.g., `mapping(address => uint) balances`). A contract function `f` induces a state transition `S -> S'`. Contract security depends on maintaining certain **invariants** across all possible state transitions. A critical invariant in a token contract is the conservation of totalSupply.

Consider the vulnerable batchTransfer function:

Plain Text

```
function batchTransfer(address[] memory _receivers, uint256 _value) public
returns (bool) {
    uint cnt = _receivers.length;
    uint256 amount = uint256(cnt) * _value; // Potential overflow
    require(balances[msg.sender] >= amount);

    balances[msg.sender] -= amount;
    for (uint i = 0; i < cnt; i++) {
        balances[_receivers[i]] += _value;
    }
    // ...
}
```

The contract invariant that should be preserved is that the sum of all users' balances remains constant (or, if there is burning/minting, changes in a controlled manner). The overflow exploit breaks this invariant. An attacker chooses _receivers and _value such that the mathematical product cnt * _value is greater than or equal to 2^256, but the modular product (cnt * _value) mod 2^256 is a small value, amount_overflowed, ideally zero.

The check require(balances[msg.sender] >= amount_overflowed) is then satisfied, as the attacker's balance is greater than a small value (or zero). However, the subsequent for loop credits each of the cnt recipients with the original _value, which is large. The result is that the sum of credited balances (cnt * _value) is much larger than the value debited from the attacker (amount_overflowed). This violates the token conservation invariant, effectively creating tokens out of thin air.

3. Case Study: The batchOverflow Exploit (CVE-2018-10299)

In April 2018, blockchain security company PeckShield identified a series of anomalous transactions involving the Beauty Chain (BEC) token [2]. These transactions transferred astronomically large amounts of tokens, exploiting an integer overflow vulnerability in the contract's batchTransfer function. This incident serves as a paradigmatic case study of the dangers of arithmetic flaws.

The vulnerable code in the BEC contract was functionally identical to the example in the previous section. The attacker executed a call to the batchTransfer function with the following parameters:

- `_receivers` : an array with two addresses.
- `_value` : `0x8000000000000000000000000000000000000000000000000000000000000000`

This `_value` is `2^255`. The variable `cnt` (length of `_receivers`) was `2`. The critical operation was the calculation of `amount`:

`amount = 2 * 2^255 = 2^256`

In EVM's 256-bit arithmetic, `2^256 mod 2^256 = 0`. Therefore, the `amount` variable became `0`. The subsequent security check, `require(balances[msg.sender] >= amount)`, became `require(balances[msg.sender] >= 0)`, which is trivially true for any balance. The contract then proceeded to debit `0` from the attacker's balance and credit `2^255` tokens to each of the two destination addresses. The result was the creation of `2^256` new BEC tokens, massively inflating the total supply and destroying the token's value. The incident forced exchanges like OKEx to suspend token trading, illustrating the systemic impact that a single smart contract flaw can have.

4. Mitigation Strategies and Formal Verification

Defense against arithmetic vulnerabilities requires a layered approach, combining secure coding practices with advanced analysis tools. The following table compares the main strategies:

Strategy	Automation Level	Gas Overhead	Type of Guarantee
Solidity ⩾ 0.8.0	Total (compiler)	Moderate	Reverts on overflow/underflow at runtime
SafeMath Library	Manual (code)	Moderate	Reverts on overflow/underflow at runtime
Symbolic Execution	High (tool)	N/A (static analysis)	Proves existence of vulnerable paths
Bounded Model Checking	High (tool)	N/A (static analysis)	Proves absence of bugs up to depth `k`

4.1. Code-Level and Compiler Mitigation

The most direct and widely adopted solution was the improvement of development tools.

SafeMath Library: Before Solidity 0.8.0, the standard practice was to use OpenZeppelin's SafeMath library. This library wraps each arithmetic operation (`add`, `sub`, `mul`, `div`) within a function that first checks if the operation would cause an overflow or underflow. If the check fails, the function reverts the transaction. While effective, it requires developer discipline to be applied consistently.

Solidity v0.8.0+: Starting with version 0.8.0, the Solidity compiler began including overflow and underflow checks by default in the generated bytecode. Any arithmetic operation that results in a wrap-around now automatically reverts the transaction. This is the currently recommended mitigation, as it provides compiler-level security without the need for external libraries.

4.2. Formal Verification

For high-criticality contracts, formal verification offers a superior level of security assurance. These methods use mathematical analysis to prove or disprove properties about the code.

Symbolic Execution: Tools like Mythril and Manticore execute the contract with symbolic rather than concrete values. They explore the program's execution paths and use an SMT (Satisfiability Modulo Theories) solver to determine if there exists any concrete input that could lead the contract to an unsafe state (e.g., a state where an overflow occurs). This is powerful for finding exploits but can suffer from path explosion in complex contracts.

Bounded Model Checking (BMC): As described by Biere et al. [3], BMC is a technique that unrolls the contract's state transition system for a finite number of steps, `k`, and translates the problem of finding a property violation into a Boolean satisfiability (SAT) problem. For overflow vulnerabilities, a model checker can be used to prove that, for any sequence of up to `k` transactions, the token conservation invariant is never violated. This does not prove total security (unless `k` is greater than the longest possible execution path), but can provide strong assurance that no short-term exploits exist.

Alternating-time Temporal Logic (ATL) Model Checking: Research such as that by Nam et al. [4] proposes using Alternating-time Temporal Logic (ATL) to model the interaction between users and the contract as a multi-agent game. Verification can then prove properties such as "there exists no strategy for a malicious agent (the attacker) to force the contract into a state where tokens are improperly created."

5. Advanced Formal Methods and Future Directions

Beyond the established techniques discussed above, emerging formal methods show promise for providing even stronger security guarantees for smart contracts.

Abstract Interpretation: This technique involves analyzing programs by computing over-approximations of their possible behaviors. For arithmetic vulnerabilities, abstract interpretation can track the ranges of integer variables throughout program execution, identifying points where overflow might occur without requiring exhaustive path exploration.

Theorem Proving: Interactive theorem provers like Coq and Isabelle/HOL allow for the construction of machine-checked proofs of program correctness. While requiring significant expertise and effort, they can provide the highest level of assurance. Projects like the DeepSEA compiler demonstrate how high-level smart contract specifications can be compiled to EVM bytecode with formal correctness guarantees.

Compositional Verification: As DeFi protocols become increasingly complex and interdependent, verifying individual contracts in isolation becomes insufficient. Compositional verification techniques allow reasoning about the security properties of contract compositions, ensuring that the interaction between multiple contracts does not introduce new vulnerabilities.

6. Empirical Analysis and Tool Evaluation

To provide practical guidance for developers and auditors, we present a comparative analysis of existing vulnerability detection tools based on their effectiveness in identifying arithmetic vulnerabilities:

Tool	Detection Method	False Positive Rate	Coverage	Academic Validation
Slither	Static Analysis	Low	High	Extensive [5]
Mythril	Symbolic Execution	Medium	Medium	Moderate [6]
Manticore	Symbolic Execution	Low	Medium	Limited
CBMC	Bounded Model Checking	Very Low	High	Extensive [7]
Solidity SMTChecker	SMT-based	Very Low	Medium	Moderate

The empirical evidence suggests that a combination of static analysis tools (like Slither) for broad coverage and symbolic execution tools (like Mythril) for deep analysis provides the most effective vulnerability detection strategy. However, the ultimate recommendation remains the use of Solidity v0.8.0+ with its built-in arithmetic checks, supplemented by formal verification for high-value contracts.

7. Conclusion and Future Work

Integer overflow and underflow vulnerabilities in the EVM are not merely theoretical bugs; they are fundamental flaws with demonstrated real-world financial consequences, as evidenced by the batchOverflow exploit. The root cause lies in the semantics of fixed-size modular arithmetic, which can violate programmer assumptions about mathematical operations.

The evolution of the Ethereum ecosystem has brought effective and easy-to-use mitigations, primarily the arithmetic checks integrated into the Solidity compiler from version 0.8.0 onwards. For the vast majority of projects, adhering to modern compiler versions is a necessary and sufficient defense.

However, for protocols managing exceptionally high values or possessing complex logic, defense in depth is prudent. The integration of static analysis tools and, crucially, the application of formal verification techniques such as Symbolic Execution and Bounded Model Checking, provide a level of assurance that simple code review or conventional testing cannot achieve.

Future research directions include the development of more scalable formal verification techniques that can handle the increasing complexity of DeFi protocols, the creation of standardized security specifications for common contract patterns, and the integration of formal methods into the smart contract development lifecycle. As the DeFi ecosystem matures, the adoption of these rigorous software engineering practices will be indispensable for ensuring the trust and stability of the next generation of decentralized finance.

The lessons learned from arithmetic vulnerabilities extend beyond smart contracts to any system where mathematical correctness is critical. The principles of formal specification, automated verification, and defense in depth represent fundamental pillars of secure system design that will remain relevant as blockchain technology continues to evolve and find new applications across various domains.

References

[1] Wood, G. (2014). *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Ethereum Project Yellow Paper. Available at: https://ethereum.github.io/yellowpaper/paper.pdf

[2] PeckShield. (2018, April 22). *New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018–10299)*. Medium. Available at: https://peckshield.medium.com/alert-new-batchoverflow-bug-in-multiple-erc20-smart-contracts-cve-2018-10299-511067db6536

[3] Biere, A., Cimatti, A., Clarke, E. M., & Zhu, Y. (2003). *Bounded Model Checking*. Advances in Computers, 58, 117-148. Available at: https://www.cs.cmu.edu/~emc/papers/Books%20and%20Edited%20Volumes/Bounded%20Model%20Checking.pdf

[4] Nam, W., & Kil, H. (2022). *Formal Verification of Blockchain Smart Contracts via ATL Model Checking*. IEEE Access, 10, 8151-8162. doi:10.1109/ACCESS.2022.3142819. Available at: https://ieeexplore.ieee.org/document/9681884/

[5] Feist, J., Grieco, G., & Groce, A. (2019). *Slither: A Static Analysis Framework for Smart Contracts*. 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 8-15.

[6] Mueller, B. (2018). *Smashing Ethereum Smart Contracts for Fun and Real Profit*. 9th Annual HITB Security Conference.

[7] Kroening, D., & Tautschnig, M. (2014). *CBMC – C Bounded Model Checker*. In Tools and Algorithms for the Construction and Analysis of Systems (pp. 389-391). Springer.