Formal Verification of EVM Bytecode in Dafny

How smart contracts are supposed to work

Ethereum is a distributed state machine. Thousands of nodes maintain identical copies of the world state—account balances, contract storage, deployed code. When a transaction arrives, every node executes the same bytecode, applies the same state changes, and reaches the same result. This determinism is required for consensus.

Smart contracts are programs deployed to addresses on this network. Once deployed, a contract's bytecode is immutable. The code executes exactly as written, forever. A developer writes source code in Solidity or Vyper, compiles it to bytecode, and deploys that bytecode to the blockchain.

Security audits review the source code. Formal verification tools analyze the source code. Testing exercises the source code. The entire security effort concentrates on the source-level artifact.

However, Ethereum nodes do not execute source code. They execute bytecode.

The problem: compilers are not verified

Between source code and bytecode sits a compiler—a complex program that parses, type-checks, optimizes, and generates code. The Solidity compiler contains approximately 500,000 lines of C++. The Vyper compiler is written in Python.

Neither compiler is formally verified.

Vyper reentrancy bug (July 2023): Curve Finance lost $26 million. The attack exploited contracts that used Vyper's `@nonreentrant` decorator. This decorator prevents a function from being called recursively—if a function calls an external contract, and that contract calls back, the second call should fail.

The source code was correct. Developers applied the decorator. Auditors verified it was present. The protection appeared to be in place.

The Vyper compiler (versions 0.2.15, 0.2.16, 0.3.0) generated incorrect bytecode. The lock check and lock set operations executed in the wrong sequence. An attacker could reenter despite the decorator.

Solidity optimizer bug (2022): The optimizer performs dead code elimination—removing operations whose results are never used. A bug caused the analysis to misclassify live storage writes as dead. The source code contained the writes. The compiled bytecode did not.

These bugs share a characteristic: source-level analysis cannot detect them. The source correctly expresses the intended behavior. The compiler fails to preserve that behavior in bytecode.

Why this problem matters

The verification gap means that proving properties about source code does not guarantee those properties hold for the executing bytecode.

A contract can be: - Audited by experts - Tested extensively - Formally verified at the source level

And still contain vulnerabilities that exist only in the bytecode.

The only way to verify bytecode behavior is to verify bytecode directly.

Formal semantics of the EVM in Dafny

The Dafny-EVM project defines a formal, executable specification of the Ethereum Virtual Machine. This specification enables direct reasoning about bytecode.

Formal: Every opcode is defined as a mathematical function with preconditions and postconditions. The Dafny verifier proves these specifications hold for all inputs.

Executable: The specification compiles to runnable code. It can execute bytecode and compare results against the Ethereum reference tests.

How the EVM executes bytecode

The stack machine

The EVM is stack-based. There are no registers. All operands pass through a single stack with maximum depth 1024. Each element is 256 bits.

To add two numbers:

```
PUSH1 0x05    // Stack: [5]
PUSH1 0x03    // Stack: [3, 5]
ADD           // Pop 3 and 5, compute 8, push result. Stack: [8]
```

Stack access is limited. DUP1-DUP16 duplicate elements at positions 1-16. SWAP1-SWAP16 exchange the top with positions 2-17. Accessing deeper elements requires shuffling.

Arithmetic

All arithmetic operates on unsigned 256-bit integers in the range [0, 2^256 - 1]. Overflow wraps modulo 2^256:

```
(2^256 - 1) + 1 = 0
```

Memory and storage

Memory is a volatile byte array that expands on access. Cost is quadratic in size.

Storage is a persistent mapping from 256-bit keys to 256-bit values. Writing to storage costs 2,900-20,000 gas depending on whether the slot was previously zero. This high cost reflects that storage must be replicated across every node.

Gas

Every operation consumes gas. ADD costs 3 gas. SSTORE (storage write) costs up to 20,000 gas. If execution exhausts the gas limit, the EVM halts and all state changes revert.

Gas is deducted before execution. Either the operation completes or it never starts.

How the Dafny-EVM represents state

The complete EVM state is a datatype:

```
datatype Raw = EVM(
    fork: Fork,           // Protocol version (Berlin, London, Cancun, etc.)
    context: Context.T,   // Caller, value, calldata, block info
    world: WorldState.T,  // All account balances and storage
    stack: EvmStack,      // Current stack contents
    memory: Memory.T,     // Current memory contents
    code: Code.T,         // Bytecode being executed
    gas: nat,             // Remaining gas
    pc: nat               // Program counter
)
```

Execution produces one of four outcomes:

```
datatype State =
    EXECUTING(evm: T)           // Still running
  | RETURNS(gas, data, ...)     // Normal termination via STOP or RETURN
  | ERROR(error, gas, data)     // Exception (out of gas, stack underflow, etc.)
  | CONTINUING(Continuation)    // Nested call waiting for callee
```

Bytecode execution and gas accounting

EVM bytecode is a sequence of bytes of type `u8` (unsigned 8-bit integers). To interpret bytecode, the EVM must decode each byte into an opcode and apply its semantics to the current state.

Executing a single opcode

The `ExecuteBytecode` function maps each byte to its corresponding operation:

```
function ExecuteBytecode(op: u8, st: ExecutingState): State {
  match op
    case STOP  => Bytecode.Stop(st)
    case ADD   => Bytecode.Add(st)
    case MUL   => Bytecode.Mul(st)
    case SUB   => Bytecode.Sub(st)
    case DIV   => Bytecode.Div(st)
    ...
    case CALL         => Bytecode.Call(st)
    case RETURN       => Bytecode.Return(st)
    case DELEGATECALL => Bytecode.DelegateCall(st)
    case REVERT       => Bytecode.Revert(st)
    case SELFDESTRUCT => Bytecode.SelfDestruct(st)
    case _            => ERROR(INVALID_OPCODE)
}
```

This function assumes gas has already been deducted. It matches the opcode byte to its semantic function and applies it to the state. If the opcode is not recognized, the result is `ERROR(INVALID_OPCODE)`. If the opcode requires operands that are not present (e.g., ADD with fewer than 2 stack elements), the semantic function returns an error state.

The execution loop

The `Execute` function orchestrates a single step of execution:

```
function Execute(st: ExecutingState): State {
  var opcode := Code.DecodeUint8(st.evm.code, st.evm.pc as nat);
  if st.evm.fork.IsBytecode(opcode)
```

```
  then
    match DeductGas(opcode, st)
      case EXECUTING(vm) => ExecuteBytecode(opcode, EXECUTING(vm))
      case s => s
  else
    ERROR(INVALID_OPCODE)
}
```

This function performs two checks before executing:

1. **Fork validity**: The opcode must be valid for the current fork. EVM opcodes can be added or removed at hard forks—specific points in time where changes are applied to the Ethereum protocol. The `fork` field identifies which version of the EVM semantics applies.
2. **Gas availability**: The `DeductGas` function checks whether sufficient gas remains. If not, it returns `ERROR(OUT_OF_GAS)`. If gas is available, it deducts the cost and returns the updated state.

Only after both checks pass does execution proceed to `ExecuteBytecode`.

Separation of concerns

The Dafny-EVM separates functional semantics from gas accounting. Each opcode's behavior (what it computes) is defined independently from its cost (how much gas it consumes).

This separation provides several benefits:

- **Clarity**: The semantic function for ADD describes only addition, not gas metering
- **Modularity**: Gas costs can be updated for new forks without modifying opcode semantics
- **Verification**: Properties about computation can be proven independently from properties about gas

The bytecode being executed is stored in `st.evm.code` as a sequence of bytes. The program counter `st.evm.pc` indicates the current position. After each instruction, the program counter advances—either sequentially or via a jump.

How opcodes are specified

Each opcode is a function from state to state with a formal specification. The ADD opcode:

```
function Add(st: State): (st': State)
    requires st.IsExecuting()
    ensures st'.OK? || st' == INVALID(STACK_UNDERFLOW)
```

```
    ensures st'.OK? <==> st.Operands() >= 2
    ensures st'.OK? ==> st'.Operands() == st.Operands() - 1
{
    if st.Operands() >= 2
    then
        var lhs := st.Peek(0) as int;
        var rhs := st.Peek(1) as int;
        var res := (lhs + rhs) % TWO_256;
        st.Pop().Pop().Push(res as u256).Next()
    else
        INVALID(STACK_UNDERFLOW)
}
```

The `requires` clause states the precondition: the machine must be executing.

The `ensures` clauses state postconditions: 1. The result is either valid or specifically STACK_UNDERFLOW—no other error is possible 2. The result is valid if and only if there were at least 2 operands 3. On success, the stack has one fewer element

---

Verifying overflow detection

Expected behavior

When adding two 256-bit integers, overflow occurs if the mathematical sum exceeds 2^256 - 1. The EVM wraps the result modulo 2^256. Solidity 0.8+ generates bytecode that detects overflow and reverts the transaction.

Expected: The bytecode should revert if and only if overflow occurred.

The mathematical property

For unsigned x, y in [0, 2^256 - 1]:

`(x + y) > 2^256 - 1` `(x + y) mod 2^256 < x`

If overflow occurs, the wrapped result is x + y - 2^256, which is less than x (since y < 2^256). If no overflow, the result equals x + y, which is at least x.

Dafny proves this equivalence automatically:

```
lemma AddOverflowNSC(x: u256, y: u256)
    ensures x as nat + y as nat > MAX_U256 <==> (x as nat + y as nat) % TWO_256 < x as nat
{ }
```

The bytecode

```
Address | Instruction | Stack state
```

```
--------|-------------|--------------------------
0x00    | DUP2        | [x, y, x]
0x01    | ADD         | [(x+y) mod 2^256, x]
0x02    | LT          | [1 if wrapped < x, else 0]
0x03    | PUSH1 0x07  | [0x07, flag]
0x05    | JUMPI       | [] - jump if flag  0
0x06    | STOP        | normal return
0x07    | JUMPDEST    | overflow handler
0x08-0C | ...         | REVERT
```

The bytecode computes (x + y) mod 2^256, compares it to x, and jumps to REVERT if the comparison indicates overflow.

The verification

```
method OverflowCheck(st: State, x: u256, y: u256) returns (st': State)
    requires st.OK? && st.PC() == 0
    requires st.Gas() >= 6 * Gas.G_VERYLOW + Gas.G_HIGH + Gas.G_JUMPDEST
    requires st.GetStack() == Stack.Make([x, y])
    requires st.evm.code == OVERFLOW_CHECK

    ensures st'.REVERTS? || st'.RETURNS?
    ensures st'.REVERTS? <==> x as nat + y as nat > MAX_U256
    ensures st'.RETURNS? <==> x as nat + y as nat <= MAX_U256
{
    st' := ExecuteN(st, 4);
    if st'.Peek(1) == 0 {
        st' := Execute(st');
        st' := ExecuteN(st', 1);
    } else {
        st' := Execute(st');
        st' := ExecuteN(st', 4);
    }
}
```

The method executes the bytecode with symbolic inputs x and y. The postconditions state that execution terminates, and that it reverts if and only if overflow occurred.

Dafny verifies this for all 2^512 possible (x, y) pairs.

Verifying loop termination

Expected behavior

A loop should terminate after a fixed number of iterations. Infinite loops would allow denial-of-service attacks (though gas limits bound actual execution).

The bytecode

A loop that iterates c times:

```
0x00  | PUSH1 c     | push counter
0x02  | JUMPDEST    | loop header
0x03  | DUP1        | duplicate counter
0x04  | PUSH1 0x08  |
0x06  | JUMPI       | if counter > 0, jump to body
0x07  | STOP        | counter == 0, exit
0x08  | JUMPDEST    | loop body
0x09  | PUSH1 0x01  |
0x0B  | SWAP1       |
0x0C  | SUB         | decrement counter
0x0D  | PUSH1 0x02  |
0x0F  | JUMP        | back to header
```

The verification

```
method Loopy(st: State, c: u8) returns (st': State)
    requires st.OK? && st.PC() == 0 && st.Capacity() >= 3
    requires st.Gas() >= [formula involving c]
    requires st.evm.code == LOOP_CODE
    ensures st'.RETURNS?
{
    st' := ExecuteN(st, 4);
    ghost var n: nat := 0;

    while st'.Peek(2) > 0
        invariant st'.OK?
        invariant st'.PC() == 0x06
        invariant [other invariants]
        decreases st'.Peek(2)
    {
        st' := ExecuteN(st', 10);
        n := n + 1;
    }
    assert n == c as nat;
    st' := ExecuteN(st', 2);
}
```

The `decreases st'.Peek(2)` clause proves termination. The counter is non-negative and decreases by 1 each iteration. It must eventually reach 0.

The ghost variable `n` tracks iteration count. The final assertion `n == c as nat` proves exactly c iterations occurred.

Dafny verifies this for all c in [0, 255].

Verifying optimization correctness

Expected behavior

Compiler optimizations should preserve semantics. If original code produces state S, optimized code should produce state S (except for gas consumption and program counter).

The optimization

Original: `SWAP_n` followed by n+1 `POP` operations Optimized: n+1 `POP` operations (SWAP removed)

The SWAP rearranges elements that are subsequently discarded. Removing it should produce the same final stack.

The verification

```
method Proposition12b(n: nat, s: seq<u256>, g: nat)
    requires 1 <= n <= 16
    requires n + 1 <= |s| <= 1024
    requires g >= (n + 1) * Gas.G_BASE + Gas.G_VERYLOW
{
    var vm := Init(gas := g, stk := s, code := []);

    var vm1 := vm;
    for i := 0 to n + 1 { vm1 := ExecuteOP(vm1, POP); }

    var vm2 := vm;
    vm2 := Swap(vm2, n).UseGas(Gas.G_VERYLOW);
    for i := 0 to n + 1 { vm2 := ExecuteOP(vm2, POP); }

    assert vm1.GetStack() == vm2.GetStack();
    assert vm2.Gas() < vm1.Gas();
}
```

Two EVMs start from identical state s. One executes the optimized sequence, the other the original. The assertions verify identical final stacks and that the

optimization saves gas.

Dafny verifies this for all n in [1, 16] and all valid stack configurations.