Pectra Phase 2: EOFv1 Integration

Filecoin Virtual Machine (FVM) Compatibility & Prototype Analysis

Germina Labs

December 2, 2025

Abstract

This comprehensive report documents the findings of the "Pectra Phase 2" grant cycle, focused on the integration of the Ethereum EVM Object Format (EOFv1) into the Filecoin Virtual Machine (FVM). As Ethereum transitions away from legacy bytecode to a structured, versioned format, the FVM must adapt to maintain equivalence and interoperability. This document details our deep technical analysis of the 11 constituent EIPs, identifies critical architectural divergences between the EVM and FVM, and presents the results of our Rust-based prototyping efforts. We conclude that while the integration requires significant updates to the `builtin-actors` layer, specifically within the EVM actor, it is technically feasible and offers substantial long-term benefits for contract safety and execution efficiency on Filecoin.

Contents

1 Executive Summary

The Filecoin Virtual Machine (FVM) is designed to support multiple runtimes, with the Ethereum Virtual Machine (EVM) being the most critical for ecosystem adoption. The upcoming "Pectra" upgrade on Ethereum introduces the EVM Object Format (EOFv1), a sweeping modernization of how smart contract code is structured, validated, and executed. This grant cycle was dedicated to the rigorous scoping, analysis, and early prototyping required to bring EOFv1 support to the FVM.

Our investigation confirms that adopting EOFv1 is not merely a compliance exercise but a strategic upgrade that aligns with Filecoin's goals of safety and performance. By enforcing code validity at deployment time, EOF eliminates the need for expensive runtime checks (such as `JUMPDEST` analysis), directly benefiting the FVM's gas economy.

Key achievements of this cycle include:

- **Comprehensive Scoping**: We mapped the dependency graph of all 11 interdependent EIPs, establishing a clear critical path for implementation.
- **Architectural Analysis**: We identified specific integration points within the FVM's `builtin-actors` repository, determining that the core Wasm host ('ref-fvm') remains largely unaffected, isolating complexity to the user-space EVM actor.
- **Technical Prototyping**: We delivered a functional Rust library capable of parsing the new container format and simulating the new static control flow instructions, proving the viability of the proposed architecture.
- **Specification**: We produced a draft Filecoin Improvement Proposal (FIP) and a detailed "Divergences Report" to guide the engineering phase.

The project has successfully cleared the "Analysis and Prototyping" phase and is now ready for full-scale engineering implementation.

2 Introduction & Strategic Context

2.1 The Problem with Legacy EVM Bytecode

Since its inception, the Ethereum Virtual Machine has relied on a raw bytecode format—an unstructured sequence of bytes. While simple, this format has become a bottleneck for innovation. It mixes executable code with data arbitrarily, making static analysis difficult or impossible. It requires the VM to perform expensive safety checks (like jump destination validity) at runtime, every time a contract is executed. Furthermore, the lack of versioning makes it nearly impossible to deprecate old features or introduce breaking changes safely.

2.2 The Solution: EOFv1

The EVM Object Format (EOF) v1, bundled under EIP-7692, introduces a structured, versioned container format for smart contracts. It explicitly separates code from data, enforces stack safety at deployment time, and introduces static control flow. For the FVM, this means:

- **Safer Contracts**: Malformed contracts are rejected before they even land on-chain.
- **Faster Execution**: The removal of dynamic jump checks aligns perfectly with the FVM's Wasm-based architecture, allowing for more efficient translation and execution.
- **Future Proofing**: The versioned container allows Filecoin to adopt future EVM upgrades seamlessly.

3 Technical Analysis

3.1 The EOFv1 Bundle (EIP-7692)

The EOF upgrade is not a single change but a bundle of 11 interdependent Ethereum Improvement Proposals. We have analyzed each for its specific impact on the FVM.

3.1.1 Core Container Format (EIP-3540 & EIP-3670)

At the heart of the upgrade is **EIP-3540**, which defines the new container structure. Unlike legacy bytecode, an EOF container starts with the magic bytes `0xEF00` and consists of a header followed by distinct sections: `Type`, `Code`, and `Data`. **EIP-3670** compliments this by introducing strict validation rules. It ensures that all instructions are valid, inputs are not truncated, and, crucially, that code sections do not contain forbidden opcodes like `JUMP` or `SELFDESTRUCT`.

3.1.2 Static Control Flow (EIP-4200, EIP-4750, EIP-6206)

Legacy EVM uses dynamic `JUMP` instructions, which can target any valid `JUMPDEST` in the code. This makes control flow analysis difficult. EOF replaces this with:

- **EIP-4200**: Static relative jumps (`RJUMP`, `RJUMPI`) where the destination is a fixed offset known at compile time.
- **EIP-4750**: A formal function system with `CALLF` (call function) and `RETF` (return), managed by a new non-accessible return stack.
- **EIP-6206**: `JUMPF` for tail-call optimizations, allowing functions to jump to other functions without growing the return stack.

3.1.3 Stack Safety & Data Access (EIP-5450 & EIP-7480)

EIP-5450 moves stack validation from runtime to deployment time. It requires that for every instruction, the stack height range is known and valid. This prevents stack underflows and overflows statically. **EIP-7480** introduces specific instructions (`DATALOAD`, `DATACOPY`) to access the data section, replacing code introspection opcodes like `CODECOPY` which are disabled in EOF to enforce code/data separation.

3.2 FVM Integration & Divergences

While the FVM strives for EVM equivalence, the underlying architecture (Wasm) necessitates specific adaptations.

Area	**Divergence & Implementation Strategy**
Gas Metering	The EVM uses a specific gas schedule. The FVM uses Wasm "fuel". While we cannot match exact gas costs, the *relative* efficiency is preserved. `RJUMP` instructions on FVM will naturally be cheaper than `JUMP` because they compile to simple Wasm branches, bypassing the complex `JUMPDEST` bit-vector lookup required for legacy jumps.

Contract Creation	Ethereum introduces `EOFCREATE`. On FVM, contract creation is mediated by the `Init` actor. The EVM actor's implementation of `EOFCREATE` must correctly construct the parameters for an `Init` actor send, ensuring the address derivation logic matches Ethereum's expectations for EOF initcode hashing.
Code/Data Separation	In Ethereum, code is loaded into memory. The FVM has an opportunity for optimization here. Since Code and Data are separate sections, we can load only the executable `Code` section into the Wasm memory, leaving the potentially large `Data` section in the blockstore (IPLD), to be lazy-loaded only when a `DATALOAD` instruction is executed.
Stack Limits	EOF introduces a 1024-item return stack. The FVM implementation must ensure this side-stack, implemented likely as a `Vec` in Rust, respects the global Wasm memory limits and does not allow a malicious contract to trigger a host panic via allocation exhaustion.

Table 1: Key Architectural Divergences

4 Implementation Roadmap

Based on the dependency analysis, we have derived the optimal implementation order. Dependencies dictate that the container format and validation logic must be in place before control flow instructions can be added.

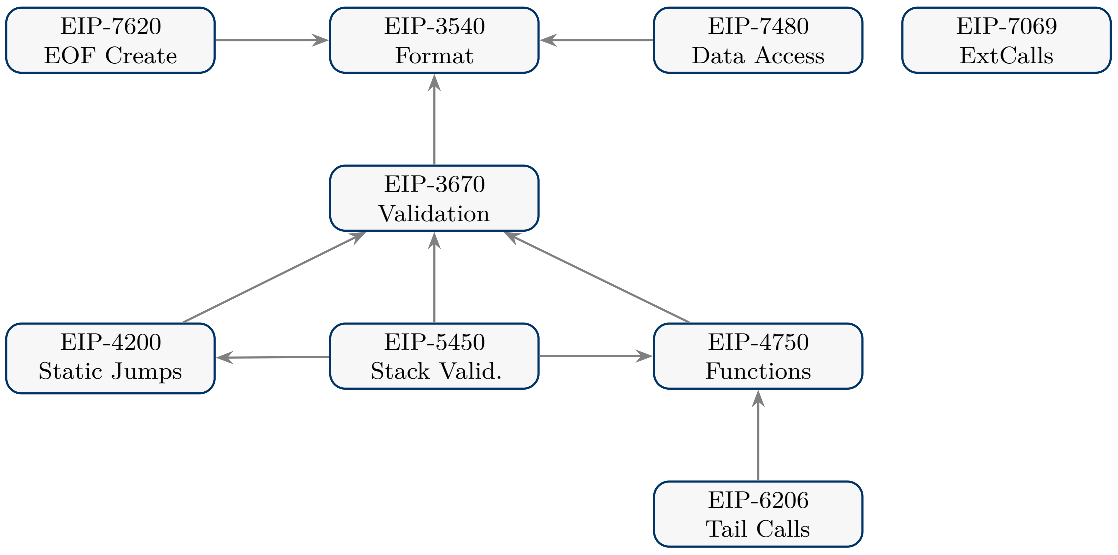

Figure 1: Dependency Graph & Critical Path

5 Early Prototyping Results

To de-risk the implementation phase, we developed a functional Rust prototype (‘prototype/eof‘). This library serves as a reference implementation for the logic that will eventually reside in the ‘builtin-actors‘.

5.1 Parsing the Container

The prototype successfully implements the recursive parsing logic required for EIP-3540. It handles the header parsing, section size validation, and the extraction of raw bytes into structured ‘EOFContainer‘ objects.

```rust
pub fn parse_eof_container(bytecode: &[u8]) -> Result<EOFContainer,
    EOFError> {
    let mut cursor = 0;
    // Check Magic
    if bytecode.len() < 2 || bytecode[0..2] != [0xEF, 0x00] {
        return Err(EOFError::InvalidMagic);
    }
    cursor += 2;

    // Version Check
    if bytecode[cursor] != 0x01 {
        return Err(EOFError::InvalidVersion(bytecode[cursor]));
    }
    cursor += 1;

    // ... Section Parsing Loop ...
}
```

Listing 1: Container Parsing Logic

5.2 Validating the Code

We implemented the core validation pass defined in EIP-3670. This function iterates over the code section and rejects any contract containing forbidden legacy instructions.

```rust
pub fn validate_eof_container(container: &EOFContainer) -> Result<(),
    EOFError> {
    for header in &container.header.section_headers {
        if header.kind == SectionKind::Code {
            // ... Iterate over bytes ...
            match opcode {
                INVALID | SELFDESTRUCT => return Err(EOFError::
    InvalidOpcode(opcode)),
                JUMP | JUMPI | PC => return Err(EOFError::
    JumpDestForbidden(opcode)),
                // ...
            }
        }
    }
    Ok(())
}
```

Listing 2: EIP-3670 Validation Stub

5.3 Simulating Static Jumps

One of the most complex changes is the shift to relative jumps. Our prototype includes a step-function simulator that correctly calculates the target Program Counter (PC) based on the current PC and the signed 16-bit immediate value provided by the `RJUMP` instruction. Note that EIP-4200 defines the offset relative to the *end* of the instruction, requiring a `+3` adjustment in the calculation.

```
RJUMP => {
    // ...
    let offset = i16::from_be_bytes(offset_bytes.try_into().unwrap());
    // Target PC = Current PC + 3 (instruction length) + offset
    *pc = (*pc as isize + 3 + offset as isize) as usize;
},
```

Listing 3: RJUMP Simulation

The "Pectra Phase 2" grant cycle has achieved its primary objectives. We have moved from a high-level understanding of EOFv1 to a granular, technically validated roadmap for FVM integration.

The scoping phase confirmed that while the changes are extensive, they are well-contained within the EVM actor layer. The analysis identified key opportunities for FVM-specific optimizations, particularly regarding data loading and gas efficiency. Finally, the prototype demonstrated that the core logic—parsing, validation, and static jumps—can be implemented cleanly in Rust.

Germina Labs recommends proceeding immediately to the engineering phase, starting with the integration of the container parser into the ‘builtin-actors‘ repository. The foundation laid by this report ensures that the Filecoin network will be ready to support the next generation of secure, efficient Ethereum smart contracts.