EVM Object Format (EOF)

Managing the Bytecode Chaos

Valerian Callens

Senior Security Auditor

Alex Murashkin

Senior Software Engineer

Quantstamp

Section 1

What is EOF and why?

What is EOF?

EOF = “EVM Object Format”

- New format of the EVM executable bytecodes
- Bytecode starting with **0xEF00** = “EOF Magic”

```
EF00 01 01 0004 02 0001 0003 04 0000 00 00 80 0000 E0 FF FD
```

Considered for the Pectra upgrade

- But postponed [https://eips.ethereum.org/EIPS/eip-7600]
- Next one is Fusaka (?)

But: are we ready?

- Does everyone understand it well enough?
- What are the upsides and downsides?
- How is the security side?

Why EOF?

Currently, any bytecode can be deployed as a smart contract:

- No meta-structure
- No bytecode versioning
- No thorough deploy-time validation

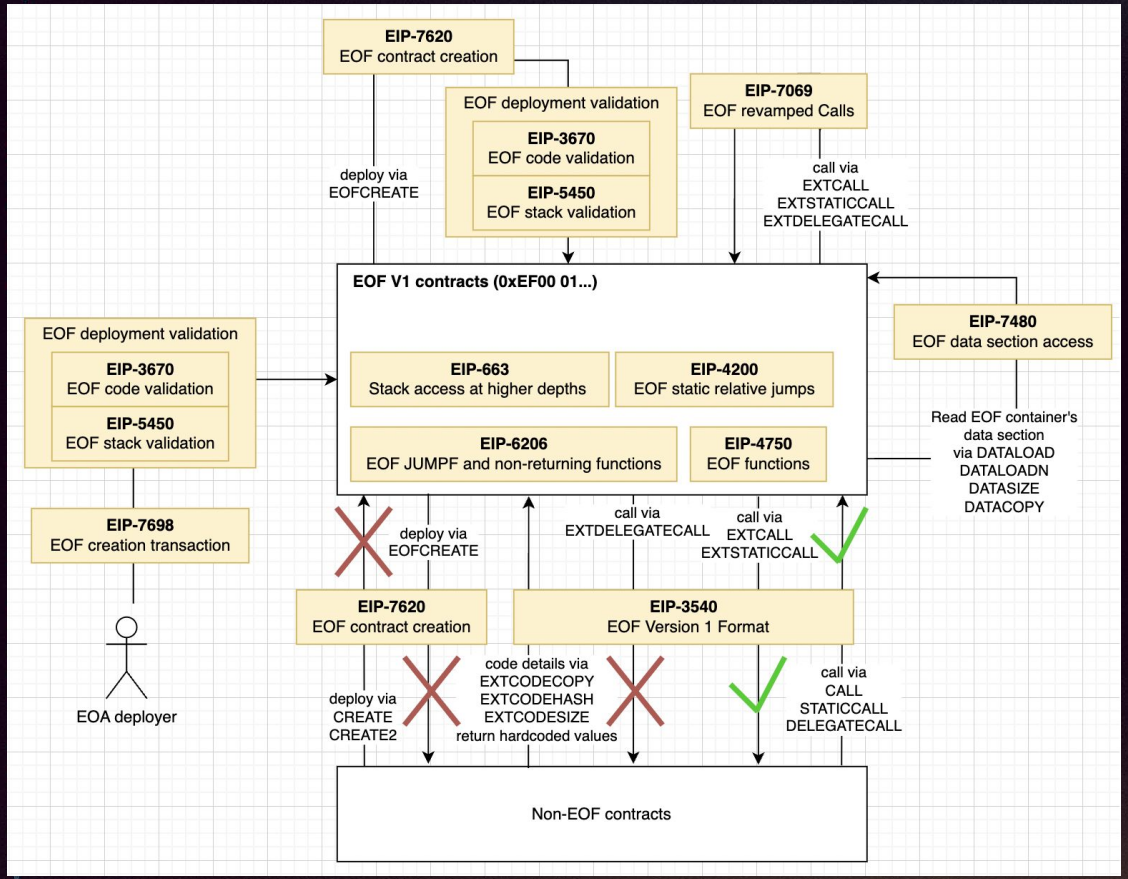
`5b600056`

Problems:

- Difficulty deprecating/adding features
- Hard for code analyzers and compilers
- Hard to fully solve legacy EVM opcode issues

Solution: add structure, versioning + make EVM better in the **NEW** version

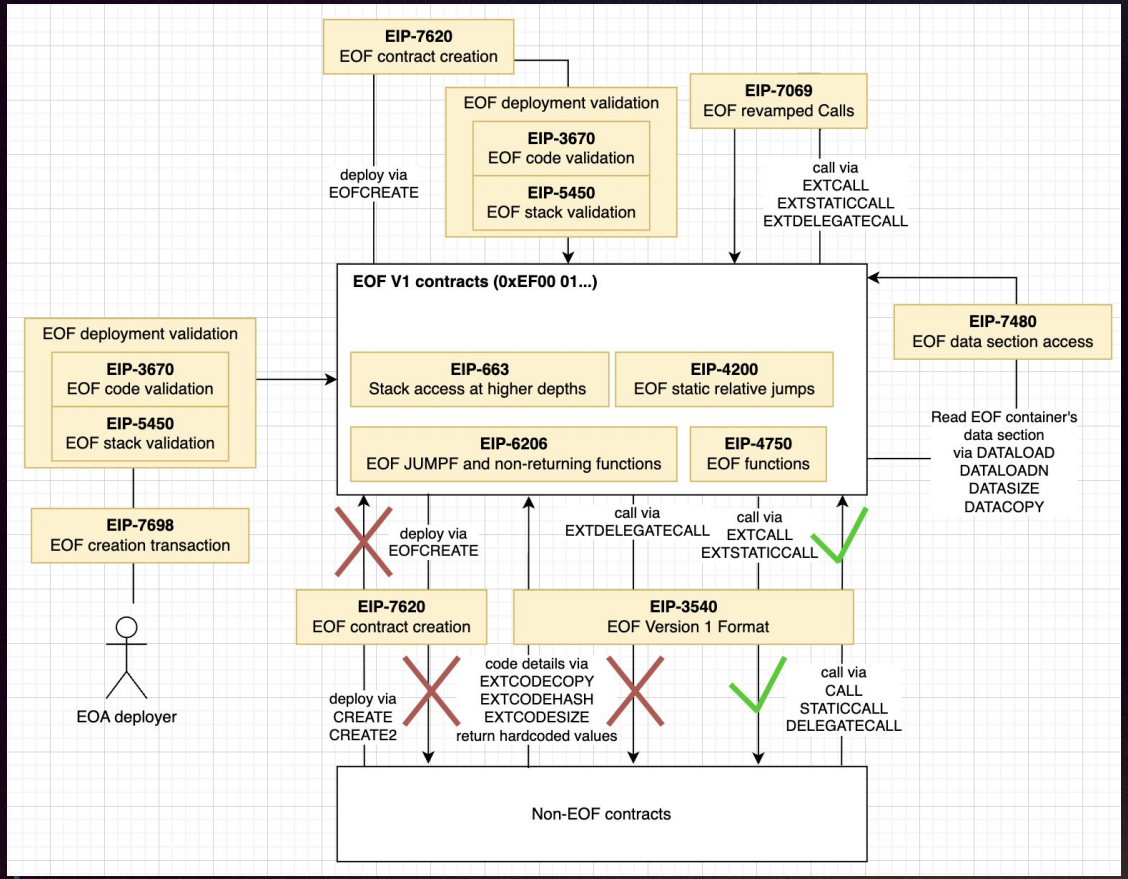
EOF Upgrade In A Nutshell



EIP-3540 EOF - EVM Object Format v1
EIP-3670 EOF - Code Validation
EIP-4200 EOF - Static relative jumps
EIP-4750 EOF - Functions
EIP-5450 EOF - Stack Validation
EIP-6206 EOF - JUMPF and non-returning functions
EIP-7480 EOF - Data section access instructions
EIP-7069 EOF - Revamped CALL instructions
EIP-7620 EOF - Contract Creation
EIP-7698 EOF - Creation Transaction

EIP-663 SWAPN, DUPN and EXCHANGE instructions

EOF Upgrade In A Nutshell



EIP-3540 EOF - EVM Object Format v1

EIP-3670 EOF - Code Validation

EIP-4200 EOF - Static relative jumps

EIP-4750 EOF - Functions

EIP-5450 EOF - Stack Validation

EIP-6206 EOF - JUMPF and non-returning functions

EIP-7480 EOF - Data section access instructions

EIP-7069 EOF - Revamped CALL instructions

EIP-7620 EOF - Contract Creation

EIP-7698 EOF - Creation Transaction

EIP-663 SWAPN, DUPN and EXCHANGE instructions

Section 2

Examples and EIP Highlights

Smart Contract Structure (EIP-3540) - Example

Solidity

```
contract InfiniteLoop {
  fallback() external payable {
    while (true) {}
}}
```

Legacy (Simplified)

```
5b600056 // the actual executable code *


5b   - JUMPDEST: Destination for the JUMP
6000 - PUSH1 0x00: Push value 0x00 onto stack
56   - JUMP: Jump to the location on the stack
(in this case, 0x00).
```

Smart Contract Structure (EIP-3540) - Example

Solidity

```
contract InfiniteLoop {
  fallback() external payable {
    while (true) {}
}}
```

Legacy (Simplified)

```
5b600056 // the actual executable code *


5b   - JUMPDEST: Destination for the JUMP
6000 - PUSH1 0x00: Push value 0x00 onto stack
56   - JUMP: Jump to the location on the stack
(in this case, 0x00).
```

EOF (Simplified)

```
[header]
EF00          // EOF “Magic”
01            // Header: version=1
01 0004       // Header: types  (01) section 4 bytes long
02 0001 0004  // Header: 1 code (02) section 4 bytes long
04 0000       // Header: data section (04)   0 bytes long
00            // Header: terminator

[body: types section]
00    // number of inputs (0)
80    // number of stack elements OR 0x80 if none
0002  // max stack height: 2

[body: code section 1]
5b600056 // the actual executable code *


* this code needs to replace dynamic jumps with the static jump
```

Static Relative Jumps (EIP-4200)

- In the legacy EVM, jumps are dynamic
- Requires destination code analysis (**JUMPDEST**)
- EOF introduces "**static relative jumps**"

Solidity	**Legacy (Simplified)**	**EOF (Simplified)**
	Dynamic (absolute) jumps: JUMP (0x56), JUMPI, JUMPDEST (0x5b)	**Static** (**relative**) jumps: RJUMP, RJUMPI, and RJUMPV

pragma solidity 0.8.24;
 	
contract InfiniteLoop {
  fallback() external {
  while (**true**) {}
  }
}		

https://github.com/ethereum/evmone/blob/master/test/unittests/eof_example_test.cpp

Static Relative Jumps (EIP-4200)

- In the legacy EVM, jumps are dynamic
- Requires destination code analysis (**JUMPDEST**)
- EOF introduces "**static relative jumps**"

Solidity	Legacy (Simplified)	EOF (Simplified)
	Dynamic (absolute) jumps: JUMP (0x56), JUMPI, JUMPDEST (0x5b)	**Static** (**relative**) jumps: RJUMP, RJUMPI, and RJUMPV

pragma solidity 0.8.24;
 	
contract InfiniteLoop {
  fallback() external {
  while (**true**) {}
  }
}
	5b600056
	
	5b – JUMPDEST: the destination of JUMP
	
	6000 – PUSH1 0x00: Push the value 0x00 onto the stack
	
	56 – JUMP: Jump to 0x00 - the contract start
		EF00 01 01 0004 02 0001 0003 04 0000 00 00 80 0000 E0 FF FD
		
		E0 – RJUMP: Relative jump
		
		FFFD – 0xFFFD: (-1) The 16-bit offset for the relative jump instruction

https://github.com/ethereum/evmone/blob/master/test/unittests/eof_example_test.cpp

Smart Contract Creation (EIP-7620) - Example

Legacy (Simplified)

CREATE or CREATE2

[initcode]

```
60 04     // PUSH 4 (length of contract bytecode)
60 0A     // PUSH 0x0A (memory location where the contract
bytecode will start)
60 00     // PUSH 0x00 (offset in memory to copy to)
39        // CODECOPY (copies the code to memory)
60 04     // PUSH 4 (length of contract bytecode)
60 00     // PUSH 0x00 (memory location to return from)
F3        // RETURN (stores the bytecode as the contract)
```

[actual code]

```
5b 6000 56 // The actual contract bytecode
```

Smart Contract Creation (EIP-7620) - Example

Legacy (Simplified)

CREATE or CREATE2

```
[initcode]

60 04     // PUSH 4 (length of contract bytecode)
60 0A     // PUSH 0x0A (memory location where the contract
bytecode will start)
60 00     // PUSH 0x00 (offset in memory to copy to)
39        // CODECOPY (copies the code to memory)
60 04     // PUSH 4 (length of contract bytecode)
60 00     // PUSH 0x00 (memory location to return from)
F3        // RETURN (stores the bytecode as the contract)

[actual code]

5b 6000 56 // The actual contract bytecode
```

EOF (Simplified)

EOFCREATE

```
[header]
EF00           // EOF “Magic”
01             // Header: version=1
01 0004        // Header: types  (01) section 4 bytes long
02 0001 0004   // Header: 1 code (02) section 4 bytes long
03 0001 0016   // Header: 1 subcontainer (03) 22(0x16) bytes long
04 0000        // Header: data section (04)   0 bytes long
00             // Header: terminator

[body: types section]
00      // number of inputs (0)
80      // number of stack elements OR 0x80 if none
0002    // max stack height: 2

[body: code section]
5F          // PUSH0 [aux data size]
5F          // PUSH0 [aux data offset]
EE 00       // RETURNCONTRACT (0xEE) first subcontainer (0x00)

[body: subcontainer]
EF00 01 01 0004 02 0001 0003 04 0000 00 00 80 0000 E0 FF FD
// EXACTLY like on the last slide!
```

Smart Contract Creation (EIP-7620) - Example

“cast decode-eof <bytecode>”

```
Header:
+-----------------------+------+
| type_size             | 4    |
|-----------------------+------|
| num_code_sections     | 1    |
|-----------------------+------|
| code_sizes            | [4]  |
|-----------------------+------|
| num_container_sections | 1    |
|-----------------------+------|
| container_sizes       | [22] |
|-----------------------+------|
| data_size             | 0    |
+-----------------------+------+

Code sections:
+---+--------+---------+-----------------+------------+
|   | Inputs | Outputs | Max stack height | Code       |
+=================================================+
| 0 | 0      | 128     | 2                | 0x5f5fee00 |
+---+--------+---------+-----------------+------------+

Container sections:
+---+-----------------------------------------------+
| 0 | 0xef000101000402000100030400000000800000e0fffd |
+---+-----------------------------------------------+
```

EOF (Simplified)

```
[header]
EF00          // EOF “Magic”
01            // Header: version=1
01 0004       // Header: types  (01) section 4 bytes long
02 0001 0004  // Header: 1 code (02) section 4 bytes long
03 0001 0016  // Header: 1 subcontainer (03) 22(0x16) bytes long
04 0000       // Header: data section (04)   0 bytes long
00            // Header: terminator

[body: types section]
00      // number of inputs (0)
80      // number of stack elements OR 0x80 if none
0002    // max stack height: 2

[body: code section]
5F          // PUSH0 [aux data size]
5F          // PUSH0 [aux data offset]
EE 00       // RETURNCONTRACT (0xEE) first subcontainer (0x00)

[body: subcontainer]
EF00 01 01 0004 02 0001 0003 04 0000 00 00 80 0000 E0 FF FD
```

EOF Functions (EIP-4750): CALLF and RETF

Solidity (pseudo-code)

```
function() external payable {
  subroutine(42); // 42 = 0x2A
}

function subroutine(uint256 arg) returns (uint256) {
  return arg;
}
```

Legacy

```
// the main code (caller)
60 2A // PUSH1 (60) 0x2A
60 09 // PUSH1 (60) 0x07 (address of the subroutine)
56    // JUMP

// back to the main code (the caller)
5b    // JUMPDEST
f3    // RETURN

// the subroutine
60 0F // PUSH1 (60) 0x05 (return address of the caller)
56    // JUMP
```

EOF Functions (EIP-4750): CALLF and RETF

Solidity (pseudo-code)

```
function() external payable {
  subroutine(42); // 42 = 0x2A
}

function subroutine(uint256 arg) returns (uint256) {
  return arg;
}
```

Legacy

```
// the main code (caller)
60 2A // PUSH1 (60) 0x2A
60 09 // PUSH1 (60) 0x07 (address of the subroutine)
56    // JUMP

// back to the main code (the caller)
5b    // JUMPDEST
f3    // RETURN

// the subroutine
60 0F // PUSH1 (60) 0x05 (return address of the caller)
56    // JUMP
```

EOF (Version 1)

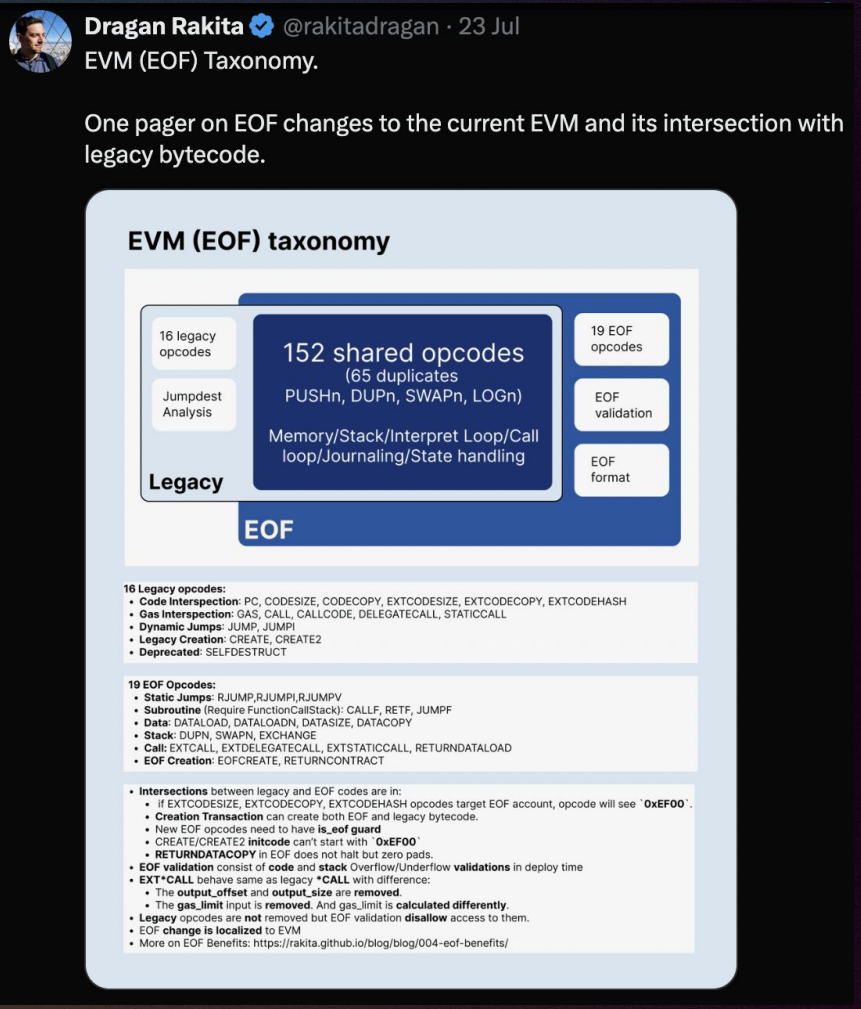
```
[header]
EF00          // EOF “Magic”
01            // Header: version=1
01 0008       // Header: types  (01) section 8 bytes long
02 0002       // Header: 2 code (02) sections
0006 0001     // one is 6 bytes long, another is 1 byte long
04 0000       // Header: data section (04)   0 bytes long
00            // Header: terminator

[body: types section for code section 1 - the caller]
00    // number of inputs (0)
80    // number of stack elements OR 0x80 if none
0001  // max stack height: 1

[body: types section for code section 2 - the subroutine]
01    // number of inputs (1)
01    // number of stack elements
0001  // max stack height (1)

[body: code section 1 - the caller]
60 2A   // PUSH1 (60) 0x2A
E3 0001 // CALLF (E3) code at section 0001
00      // terminator

[body: code section 2 - the subroutine]
E4      // RETF
```

EOF: Code Validation (EIP-3670)

EOF: Code Validation

1. Checking for valid instructions and disallowing the invalid ones (**SELFDESTRUCT** and all legacy instructions)
2. **JUMPDEST** analysis is not necessary - this is already done at deployment time
3. Expected to have linear computational and space complexity
4. Old rules for the legacy contracts

Question: what about **interactions** between legacy and EOF contracts, which rules apply?



EOF <> Legacy Interactions

Can a legacy contract deploy an EOF contract?
- No, **CREATE/CREATE2** fail when the target bytecode starts with **0xEF**

Can an EOF contract deploy a legacy contract?
- No, **CREATE/CREATE2** are considered invalid instructions in EOF

Can an EOF contract call a legacy contract?
- EXTCALL, EXTSTATICCALL - ALLOWED
- EXTDELEGATECALL (DELEGATECALL replacement) - NOT ALLOWED

Can a legacy contract call an EOF contract?
- DELEGATECALL - ALLOWED (for existing proxy contracts to use EOF upgrades)

Note:
- EOF **must not** be enabled on chains with 0xEF00-prefixed bytecodes that are not valid EOF (EIP-3541 in London fork)

Revamped CALL Instructions (EIP-7069)

- “Gas observability” has been a problem
- Breaking changes due to gas “re-pricings” across EIPs

Legacy	**EOF**
1. The caller can pass an arbitrary gas limit into function call 2. *63/64th* rule 3. Max call depth: *1024*	1. The caller **has no control over the amount of gas** passed in as part of the function call 2. *63/64th* rule 3. Max call depth: *1024* (considered for removal) 4. MIN_RETAINED_GAS introduced 5. New opcodes with simplified semantics introduced: EXTCALL, EXTDELEGATECALL, and EXTSTATICCALL 6. Return code becomes non-boolean

Section 3

Support, Implementation, and Community

EOF Support and Implementation

Implemented:

1. Hyperledger Besu
2. Nethermind (https://github.com/NethermindEth/nethermind/commits/feature/evm/eof) - NET Client
3. EVM One (https://github.com/ethereum/evmone): C++ Implementation
4. REVM (https://github.com/bluealloy/revm): Rust

Not Fully Implemented:

1. Geth: partial, under development
2. Solc (Solidity): partial, under development

EOF: Related Tools

EOF fuzzer: https://github.com/marioevz/eoffuzzer

EOF bytecode parsers:

- https://github.com/AmadiMichael/EOF-Bytecode-Parser
- https://github.com/malik672/EOF-Parser

EOF code performance benchmark:

- https://github.com/cairoeth/sp1-eof

EOF: Useful Resources

Discussion thread about EOF by Ethereum Magicians community
Consolidated specs for EOF V1 by Ipsilon
https://evmobjectformat.org by Ipsilon

Individual analyses:

1. EOF benefits by Dragan Rakita
2. Why I am against EOF in Pectra by Marius Van Der Wijden
3. EOF explained - What developers need to know by BuildBear
4. A complete guide to EOF by Marko Veniger
5. Functional changes of EOF by jtriley.eth
6. Technical overview of EOF by 0xnightfall.eth
7. Ethereum set for overhaul of crucial programming standard with EOF by Margaux Nijkerk

Videos:

1. PEEPanEIP #121: EOF with Danno Ferrin by Ethereum Cat Herders
2. EOF Upgrade by Uttam Singh

EOF: Performance Analysis

Highlights:

1. Faster transactions and execution, and smaller size
2. Results by **BuildBear Labs:**

	Legacy	EOF	Delta
Init Code Size	31,202 bytes	29,236 bytes	~6.5% smaller
Deployed Code Size (Optimized)	7,058 bytes	6,388 bytes	~10% smaller
Deploy Gas Usage	6,832,734 gas	5,925,377 gas	~14% less gas
Call gas usage	8,815,561 gas	8,094,095 gas	~9% less gas
Execution Time (100 calls)	18,539 microseconds	13,870 microseconds	~15% faster

https://medium.com/buildbear/eof-explained-what-developers-need-to-know-179091e21c03

Section 4

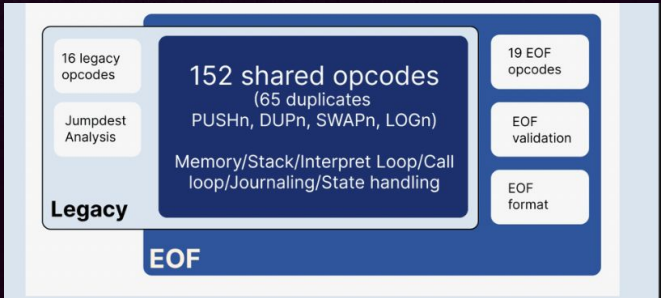
Security

EOF: Security Benefits

1. **Removal of JUMPDEST analysis:**
    a. JUMDEST analysis initially was security-related
    b. There is only so much can be done at runtime
    c. DOS attack vector of deployment complicated initcodes (patched by EIP-3860)
2. **Stack operations revision (SWAPN, JUMPN opcodes)**
    a. Not EOF-specific
    b. “Stack too deep” previously required re-writing your code
    c. More intuitive code leads to better security
3. **Static jumps and better structure for bytecode**
    a. Potentially faster computational complexity for static analysis: **O(n)** vs **O(n^2)**
    b. Easier analysis due to better structure
4. **Gas observability revised**

EOF: Potential Drawbacks and Risks

1. The opcode space is **more complex**:
   a. 19 EOF opcodes
   b. 152 shared opcodes
   c. 16 legacy opcodes
2. No publicly known security analysis yet
3. Not enough adoption yet



Conclusion

EOF manages the bytecode chaos by:
versioning, structure, relative jumps, better stack management, and deploy-time validation

EOF adds complexity:
19 new opcodes, lack of knowledge and thorough analyses

EOF is seen as:
investment in the future, more benefits in long-term rather than immediate

EOF (EVM Object Format)

Thank you!

Alex Murashkin

Valerian Callens

alex@quantstamp.com

valerian@quantstamp.com