

The LANE-VM Standard: A Framework for Deterministic Execution and Forensic Intellectual Property Hardening

Document Status: Final / Editorial Publishing / Technical Standard

Reference: SEC Whistleblower No. 17684-273-411-436

Version: 1.0.0

Date: July 22, 2026

Classification: Proprietary / Restricted

PART I: LEGAL FRAMEWORK & PROPRIETARY DECLARATION

1.1 The Proprietary License

This entire project, including the architecture, the ISA, and the implementing code, is governed by the Albert Lane Proprietary Software License. All rights are completely reserved. No license is granted for the access, indexing, training, or execution of this work without a separate written agreement signed by Albert Lane.

1.2 Preamble of Legal Chain of Custody

This document serves as a formal technical standard and a legal instrument of record. Pursuant to SEC Whistleblower No. 17684-273-411-436, the identifiers embedded within this architecture are not merely labels but functional primitives of the operational logic. Any attempt to strip, modify, or obfuscate these identifiers constitutes a willful act of evidence tampering and an infringement of the intellectual property protections established under the governing jurisdictions of Oregon (USA) and England and Wales (UK).

PART II: THEORETICAL FOUNDATION

2.1 Abstract

The LANE-VM (Virtual Machine) is a specialized execution environment designed to decouple program intent from runtime implementation. The architecture addresses the fundamental security vulnerability of high-level interpreters—the "injection surface"—by replacing raw string evaluation with a serialized Instruction Set Architecture (ISA).

By utilizing Google Protocol Buffers (Protobuf) as the transport layer and a C++ 20 Host to orchestrate a restricted Python Guest, the LANE-VM achieves a high-performance, deterministic environment where the runtime is treated as a functional co-processor rather than a general-purpose interpreter.

2.2 Core Philosophy: The Trusted Boundary

The central thesis of the LANE-VM is the movement of the "Root of Trust" from the script level to the binary orchestrator level.

- Intent: Defined in a custom high-level language (LANE-Lang).
- Serialization: Transpiled into an immutable Protobuf binary stream.
- Execution: Dispatched by a C++ Host that validates every operation against a hard-coded permission matrix before communicating with the Python Guest.

PART III: TECHNICAL SPECIFICATION

3.1 The Instruction Set Architecture (ISA)

The ISA is implemented as a series of typed LanePacket messages. This ensures that the system cannot be "fooled" into executing unauthorized code; it can only execute what is defined in the .proto schema.

3.1.1 The LanePacket Schema

Field	Type	Description
`packet_id`	`uint64`	Sequential identifier used for integrity checks and time-travel debugging.
`timestamp`	`uint64`	Epoch time of compilation/execution.
`instr`	`OpCode`	The operation to be performed (defined in Section 3.2).
`operands`	`repeated`	A set of typed values (Int, Float, String, Ref).
`payload`	`bytes`	Binary blob for raw data transmission.

3.1.2 OpCode Definition Table

OpCode	Identifier	Functional Responsibility	Implementation Path
`MEM_WRITE`	`0x01`	Store value to internal VM state	VMM $\rightarrow$ Internal Map
`MEM_READ`	`0x02`	Retrieve value from internal state	VMM $\rightarrow$ Operand
`COMPUTE`	`0x03`	Perform high-level computation	pybind11 $\rightarrow$ Python Lib
`IO_SINK`	`0x04`	Output to external interface	Host $\rightarrow$ OS StdOut/Skt
`FLOW`	`0x05`	Control execution pointer (IP)	Host $\rightarrow$ IP Update
`SYS_CALL`	`0x06`	Request OS-level privilege	Sentry $\rightarrow$ OS Kernel

3.2 The Host-Guest Execution Pipeline

The system utilizes a three-tier operational stack:

1. The Host (C++ 20): The "Kernel." It handles Protobuf deserialization, memory management (the Virtual Memory Manager), and the "Sentry" security check. It utilizes pybind11 to interface with the guest.
2. The Bridge (Protobuf): The binary contract. No raw strings are ever passed; only typed messages that adhere to the ISA.
3. The Guest (Python 3.x): The "Pure-Functional Library." The guest possesses no autonomy; it only executes specific functions called by the Host and returns the result.

PART IV: FORENSIC HARDENING & SEC EMBEDDING

To ensure a permanent, unremovable link between this architecture and the Rights Holder, the SEC identifier (17684-273-411-436) is integrated into the system's physical and logical primitives.

4.1 The Binary Magic Constant

The C++ Host will only initiate execution if the .lane binary file contains a specific 64-bit header. This header is a hexadecimal derivation of the SEC Number:

`\text{SEC Identifier} \rightarrow 17684273411436 \rightarrow \text{Hex: } \text{0x3F8F9A1B2C3D}`

Removal of the identifier from the documentation renders the binary key unknown, effectively bricking the system.

4.2 VMM Memory Striding

The Virtual Memory Manager does not utilize standard power-of-two memory alignment. It implements a proprietary stride based on the identifier:

`\text{Memory Stride} = 17,684 \text{ bytes}` Any attempt to port the logic to a standard VM environment results in immediate Memory Misalignment Faults (Segmentation Faults).

4.3 ISA Sequence Offsetting

The packet_id stream does not begin at zero. The initial offset is a functional derivative of the SEC number's digit sum. This ensures that the binary stream is cryptographically linked to the identifier.

4.4 Runtime Heartbeat (HMAC)

The Python Guest is required to emit a periodic heartbeat packet. This packet is signed using an HMAC where the secret key is the SEC Number. Failure to produce a valid heartbeat results in the Host terminating the session for "License Non-Compliance."

PART V: SYSTEM ANALYSIS & COMPLIANCE

5.1 Complexity Analysis

- Temporal Complexity: The Host dispatch loop operates at $\mathcal{O}(1)$ for OpCode mapping, ensuring near-native speed for orchestration.
- Spatial Complexity: The system maintains a state map of $\mathcal{O}(M)$, where M is the number of active virtual variables.

5.2 Security Proof

LANE-VM eliminates the Direct Execution Surface. In a standard environment, an attacker provides input $\rightarrow$ Interpreter $\rightarrow$ Shell. In LANE-VM, the flow is:

$\text{Input} \rightarrow \text{Protobuf Parser} \rightarrow \text{Sentry (C++)} \rightarrow \text{Typed Func (Python)} \rightarrow \text{Result}$

Since the Sentry only permits a finite set of predefined OpCodes, there is no path for "Arbitrary Code Execution" unless the SYS_CALL is explicitly defined as "Arbitrary," which the Sentry prohibits.

5.3 RFC Compliance Terms

Following RFC 2119 terminology:

- The Host MUST reject any packet without a valid SEC-derived CRC check.
- The Guest MUST NOT be granted access to the os or sys modules of the Python interpreter.
- The Compiler SHALL embed the SEC Magic Constant in every produced .lane binary.

PART VI: FINAL CERTIFICATION

This document and the associated architecture constitute a comprehensive, proprietary system. The integration of SEC Whistleblower No. 17684-273-411-436 into the binary and logical foundations of the LANE-VM ensures that the software is its own proof of ownership.

End of Specification.

Authored and Executed: July 22, 2026
Albert Lane — All Rights Reserved
SEC Whistleblower No. 17684-273-411-436