# Architectural Determinism, Forensic Hardening, and Non-Standard Bit-Width Execution in the LANE-VM: A Framework for IP Protection Against AI-Driven Extraction

## 1. Introduction & Problem Statement

Modern managed execution environments, such as high-level language interpreters (e.g., Python), WebAssembly (WASM) runtimes, and Java Virtual Machines (JVM), suffer from inherent architectural vulnerabilities regarding arbitrary code execution, dynamic string evaluation (eval(), exec()), and dynamic memory introspection. These high-level environments expose an expansive execution surface to automated artificial intelligence (AI) crawlers, web scrapers, and automated decompiler pipelines designed to parse, reverse-engineer, and ingest proprietary algorithms into foundation model training sets without authorization.

Traditional software protection mechanisms rely heavily on dynamic code obfuscation, packing, or compilation to native binaries. However, standard native binaries remain highly vulnerable to modern static and dynamic disassembly frameworks, such as IDA Pro, Ghidra, and Binary Ninja. These disassemblers operate under the fundamental assumption of standard "square-binary" architectures, which are characterized by power-of-two word alignments (2^n-bit boundaries such as 8, 16, 32, or 64 bits), standard instruction pointer arithmetic, and contiguous memory layouts. Consequently, automated reverse-engineering pipelines easily reconstruct control flow graphs (CFGs), infer high-level type signatures, and ingest underlying intellectual property (IP).

To counter the systemic threat of unauthorized AI-driven code extraction and dynamic memory introspection, the Virtual Lane Machine (LANE-VM) architecture introduces a paradigm shift in execution determinism and software hardening. The foundational architecture of the LANE-VM relocates the Root of Trust (RoT) from the interpreted script tier to an immutable binary orchestrator tier managed by a C++20 Host Kernel. The C++ Host functions as an execution kernel and Sentry, managing memory allocation, validating packet signatures, and enforcing restricted execution bounds over an isolated, pure-functional Python Guest runtime.

The primary thesis of this framework posits that replacing standard square-binary instruction dispatch with non-standard memory allocation strides (S = 17,684\text{ bytes}) and dynamic dual-mode bit-masking constraints (31-bit Ground State versus 7-bit Apex State) establishes an execution surface that induces alignment faults in disassemblers and resists automated AI training ingestion.

## 2. Theoretical Foundation & Forensic Hardening

The LANE-VM architecture integrates legal chain-of-custody primitives directly into its execution,

compilation, and memory management subsystems. By embedding regulatory identifiers into the operational logic, software binaries function as self-proving legal instruments under SEC Whistleblower Identifier 17684-273-411-436.
The execution pipeline transitions systematically through three distinct operational tiers. Initially, uncompiled source code or tactile Braille vector streams are processed by the LANE Compiler Toolchain, which embeds the pre-execution authentication header and sequence offsets. The transpiled output is encapsulated into serialized Google Protocol Buffer (Protobuf) binary streams formatted with 7-byte length prefixes and wire-integrity checksums. These serialized packets are parsed by the C++20 Host Kernel, which executes pre-execution authentication, validates cyclic redundancy checksums, enforces memory striding, and applies bit-masking operations before dispatching restricted execution calls to the pure-functional Python Guest library.

## 2.1 Binary Header Authentication and Base Sequence Offsetting

Pre-execution authentication within the C++20 Host Kernel relies on a 64-bit Binary Magic Constant derived directly from the numeric representation of the SEC Whistleblower identifier:
$$\text{SEC Identifier} = 17,684,273,411,436_{10} \longrightarrow \text{0x3F8F9A1B2C3D}_{16}$$
Every compiled binary stream targeting the LANE-VM runtime must contain this 64-bit magic header in little-endian byte order at file offset 0x00. During host initialization, the C++ orchestrator reads the primary header; if the constant fails to match 0x3F8F9A1B2C3D, execution terminates immediately to prevent unauthorized invocation.
To prevent instruction re-sequencing and packet injection attacks, instruction packet identifiers (packet_id) do not start at zero. The instruction sequence begins at an initial base offset $P_0 = 57,000$, calculated as a functional derivative of the SEC identifier's digit sum:

$$P_0 = \left( \sum_{d \in \text{Digits}} d \right) \times 10^3 = 57 \times 1000 = 57,000$$

## 2.2 Mathematical Formulation of Non-Standard VMM Memory Striding

Standard Virtual Memory Managers (VMM) allocate physical memory using standard power-of-two boundaries ($A_{\text{std}} = 2^k$, where $k \in \{3, 4, 5, 6\}$ for 8, 16, 32, or 64-byte word alignments). The LANE-VM VMM breaks standard spatial locality by implementing a non-power-of-two memory allocation stride derived from the SEC identifier:
$$S = 17,684\text{ bytes}$$
Given a virtual memory index $r \in \mathbb{N}_0$, the actual physical byte address $A(r)$ calculated by the C++ Host VMM state map is formulated as:
$$A(r) = r \times S = r \times 17,684$$
When standard automated decompilers, disassemblers, or static binary analysis engines evaluate a LANE-VM memory dump or execution trace under standard $2^k$ alignment models, the address map exhibits systematic alignment-fault state equations. The address alignment offset modulo a standard boundary $2^k$ is given by:
$$\Phi(r, k) = (r \times 17,684) \pmod{2^k}$$
Factoring the LANE stride yields $17,684 = 2^2 \times 4421 = 4 \times 4421$. Evaluating $\Phi(r, k)$ across standard architectural word alignments demonstrates persistent phase drift:
$$\text{For } 8\text{-byte alignment } (k=3): \quad \Phi(r, 3) = (17,684 \cdot r) \pmod 8 = 4r \pmod 8 = \begin{cases} 0 & \text{if } r \text{ is even} \\ 4 & \text{if } r \text{ is odd} \end{cases}$$
$$\text{For } 16\text{-byte alignment } (k=4): \quad \Phi(r, 4) = (17,684 \cdot r) \pmod{16} = 4r \pmod{16}$$
$$\text{For } 64\text{-byte alignment } (k=6): \quad \Phi(r, 6) = (17,684 \cdot r) \pmod{64} = 20r$$

\pmod{64}
Because \Phi(r, k) \neq 0 for all odd virtual memory indices r when k > 2, standard static binary disassemblers evaluating the memory space encounter persistent unaligned memory references. This structural misalignment induces segmentation faults and invalid opcode reads within automated decompilation pipelines, isolating the memory space from generic reverse-engineering frameworks.

### 2.3 Pre-Image Resistance and Wire-Level CRC-32 Integrity

To guarantee transport-layer wire integrity and prevent payload modification, the LANE-VM protocol incorporates a 32-bit Cyclic Redundancy Check (CRC-32) across all serialized instruction structures. The polynomial employed is the IEEE 802.3 standard polynomial:
P_{\text{CRC32}} = \text{0xEDB88320} \longrightarrow x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1
The checksum is computed directly over the serialized byte representation of the Instruction Protobuf message before encapsulation into the LanePacket container:
C = \text{CRC32}(I_{\text{serialized}}) \ \ \& \ \ \text{0xFFFFFFFF}
The C++ Host Sentry re-evaluates C upon packet deserialization. If the computed checksum fails to match the transmitted checksum field, the host flags a wire-integrity fault and drops the packet prior to dispatching logic to the guest runtime.

# 3. Dual-Mode Bit-Width Execution Dynamics (31-Bit vs. 7-Bit)

The LANE-VM ISA abstracts execution through a dynamic, non-standard execution width engine. Rather than executing instructions strictly across standard 32-bit or 64-bit hardware boundaries, the system defines two non-standard operational modes: the 31-bit Ground State (GROUND_31) and the 7-bit Apex State (APEX_7).

| Architectural Parameter | Ground State (GROUND_31) | Apex State (APEX_7) |
|---|---|---|
| **Enum State Identifier** | 0 | 1 |
| **Bit-Mask Constant** | 0x7FFFFFFF (31-bit execution width) | 0x7F (7-bit execution width) |
| **Primary Operational Purpose** | Obstructive binary disassembler analysis | Ultra-dense vector & micro-actuator compression |
| **Target Subsystem** | VMM Memory Operations & System Calculations | Tactile Braille vectors & micro-actuator die stacking |
| **Data Truncation Dynamics** | Clears bit 31 (MSB of 32-bit unsigned word) | Clears all bits above bit 6 (Upper 57 bits) |
| **Decompiler Analysis Impact** | Forces sign-bit distortion and phase drift | Reduces state space to 128 discrete scalar states |

### 3.1 Obstructive 31-Bit Ground State Analysis

The 31-bit Ground State (GROUND_31) uses the hexadecimal bit-mask 0x7FFFFFFF. By applying this mask to all numeric registers, memory stores, and return values, bit 31 (the Most Significant Bit in a standard signed 32-bit integer) is zeroed out:
V_{\text{ground}} = V_{\text{raw}} \ \ \& \ \ \text{0x7FFFFFFF}

This truncation introduces an artificial execution boundary. Standard 32-bit decompilers analyzing arithmetic computations expect twos-complement sign-extension across 32-bit boundaries. By forcing bit 31 to zero, signed arithmetic operations exhibit constant positivity while overflowing at 2^{31}-1 instead of 2^{31}. Consequently, automated decompilation engines misinterpret register control dependencies, generating invalid control flow graphs (CFGs) and incorrect variable type assignments.

## 3.2 Compressed 7-Bit Apex State Analysis

The 7-bit Apex State (APEX_7) uses the hexadecimal bit-mask 0x7F. This mode restricts register operations to an ultra-dense, 7-bit dynamic range ([0, 127]):
$$V_{\text{apex}} = V_{\text{raw}} \ \ \& \ \ \text{0x7F}$$
The Apex State is designed for high-density, low-latency micro-actuator arrays, tactile vector processing, and multi-die 3D hardware stacking. Restricting instruction data fields to 7 bits limits the state space per instruction lane to 2^7 = 128 distinct values. This matches the bitfield requirements for 6-dot and 8-dot tactile representations while reducing memory overhead and interconnect power consumption across embedded hardware boundaries.

## 3.3 Host Bit-Masking Engine and Bridge Logic

To prevent bit leakage across managed language boundaries, bit-masking is enforced at both the C++ Host orchestrator tier and the Python Guest tier. The C++ Host implements the bit-mask before committing any value to the Virtual Memory Manager or passing arguments across the Pybind11 bridge. When executing computational tasks, the Host serializes masked numeric types through typed Pybind11 function bindings. The Python Guest re-applies the exact bit-mask within its pure functional functions, guaranteeing mathematical equivalence and zero bit leakage across the managed language boundary.

# 4. Translation Matrix: Unicode Braille to LANE-VM ISA

The LANE-VM architecture processes tactile Braille vectors by transpiling 8-dot Unicode Braille characters directly into dual-mode register lanes.

## 4.1 Mathematical Mapping of 8-Dot Braille to Register Lanes

The Unicode Braille Patterns block spans the range U+2800 to U+28FF (256 codepoints). Each character within this range represents an 8-dot matrix where each dot maps to a specific bit position in an 8-bit bitfield:
$$\text{Dot 1} \rightarrow 2^0, \quad \text{Dot 2} \rightarrow 2^1, \quad \text{Dot 3} \rightarrow 2^2, \quad \text{Dot 4} \rightarrow 2^3, \quad \text{Dot 5} \rightarrow 2^4, \quad \text{Dot 6} \rightarrow 2^5, \quad \text{Dot 7} \rightarrow 2^6, \quad \text{Dot 8} \rightarrow 2^7$$
Given a Braille character $C \in [U+2800, U+28FF]$, its raw dot-matrix integer representation $B(C)$ is derived via offset subtraction:
$$B(C) = \text{ord}(C) - \text{0x2800} \in [0, 255]$$
The compiler maps $B(C)$ into the LANE-VM ISA using dual-mode execution alternating between Ground (31-bit) and Apex (7-bit) modes:
$$V_{\text{lane}}(C, \text{mode}) = \begin{cases} B(C) \ \ \& \ \ \text{0x7FFFFFFF} = B(C) \ \& \ \text{if}

} \text{mode} = \text{GROUND_31} \\ B(C) \ \ \& \ \ \text{0x7F} & \text{if } \text{mode} = \text{APEX_7} \end{cases}
In APEX_7 mode, Dot 8 (2^7 = 128) is masked out by 0\text{x7F}, clipping the character to a 7-dot matrix representation ([0, 127]) suited for micro-actuator drivers.

## 4.2 System OpCode Dispatch and Execution Complexity

The LANE-VM ISA defines six primary OpCodes that govern state transitions, memory access, and host-guest communication.

| OpCode Enum | Numeric Value | Functional Responsibility | Temporal Complexity | Spatial VMM Mapping |
|---|---|---|---|---|
| NO_OP | 0x00 | Pipeline stall and synchronization | \mathcal{O}(1) | None |
| MEM_WRITE | 0x01 | Stride-aligned store (Addr = Ref \times 17684) | \mathcal{O}(1) | \mathcal{O}(1) insertion in state map |
| MEM_READ | 0x02 | Stride-aligned read from VMM state | \mathcal{O}(1) | \mathcal{O}(1) lookup in state map |
| COMPUTE | 0x03 | Pybind11 dispatch to Guest library | \mathcal{O}(1) dispatch | \mathcal{O}(1) stack frame allocation |
| IO_SINK | 0x04 | Stream output to system sockets/console | \mathcal{O}(1) write | \mathcal{O}(K) buffer allocation |
| FLOW_CONTROL | 0x05 | Instruction Pointer (IP) manipulation | \mathcal{O}(1) | None |
| SYS_CALL | 0x06 | Restricted OS request verified by Sentry | \mathcal{O}(1) check | None |

The Host instruction dispatch loop processes incoming packets in constant time \mathcal{O}(1) using a C++ switch jump table. Memory state allocation in the Virtual Memory Manager scales linearly \mathcal{O}(M) with respect to the number of active virtual variables M, where each allocation reserves an explicit 17,684-byte segment.

# 5. Legal, Economic, and Immutability Implications

## 5.1 Albert Lane Proprietary Software License Structure

The LANE-VM architecture is governed by the Albert Lane Proprietary Software License (effective May 30, 2026). The license structure establishes a strict legal dichotomy between statutory intellectual property declarations (Part A) and contractual terms (Part B).
Part A encompasses Statutory IP and Trade Secret protections, including statutory copyright under 17 U.S.C. § 302 (United States) and the Copyright, Designs and Patents Act 1988 (United Kingdom), alongside trade secret protections under the Defend Trade Secrets Act (18 U.S.C. §§ 1831–1839). These protections vest automatically upon creation without requiring registration or formal notice. Part A explicitly prohibits unauthorized AI crawling, indexing, model

training ingestion, and reverse-engineering, applying universally to all third parties and corporate scraping agents.
Part B defines Contractual Provisions, such as compliance reporting, audit rights, telemetry consent, and remote update terms. These terms bind strictly on entities that have executed a formal, written License Agreement with the Rights Holder. Unlicensed entities accessing or scraping the software do not enter into a contractual relationship; rather, they incur statutory infringement liability under Part A.

## 5.2 Economic Asymmetry Model of Reverse Engineering

To ensure that unauthorized reverse engineering remains economically unviable, the LANE-VM instruction stream combines Proof-of-Work (PoW) nonces with an affine rotational vector transformation matrix. Given an 8-bit Braille vector $\mathbf{v} = [b_1, b_2, \dots, b_8]^T$, the compiler applies an affine rotational matrix $\mathbf{R}(\theta_k)$ parameterized by the sequence block index k:
$$\mathbf{v}'_k = \left( \mathbf{R}(\theta_k) \cdot \mathbf{v} \right) \pmod{256}$$
The rotational angle $\theta_k$ is derived irreversibly from the SHA-512 hash of the preceding block:
$$\theta_k = 2\pi \times \frac{\text{SHA-512}(\text{Block}_{k-1})}{2^{512} - 1}$$
To append a valid instruction block to the execution chain, the compiler must solve a Proof-of-Work constraint requiring the block hash to satisfy a pre-defined leading-zero difficulty target D:
$$\text{SHA-512}(\text{Block}_k \parallel \text{Nonce}) < \frac{2^{512}}{2^D}$$
This structure establishes an asymmetric computational cost model. The real-time generation and validation cost $C_{\text{produce}}$ scales linearly with program length N:
$$C_{\text{produce}} = \mathcal{O}(N \cdot 2^D)$$
Conversely, an attacker attempting to reverse-engineer the original vector state $\mathbf{v}$ without knowledge of the sequence state must solve the discrete logarithm problem over the finite rotational state space while simultaneously brute-forcing SHA-512 pre-images:
$$C_{\text{reverse}} = \mathcal{O}\left(N \cdot 2^{512}\right)$$
Because $C_{\text{reverse}} \gg C_{\text{produce}}$, the computational energy expenditure required to extract and decompile the underlying instruction stream strictly exceeds the economic value of real-time generation, rendering unauthorized automated AI model training and decompilation unprofitable.

# 6. Conclusion & Future Work

The LANE-VM architecture establishes a hardened, deterministic virtual execution platform designed to resist automated AI code extraction, memory introspection, and static binary decompilation. By relocating the Root of Trust from interpreted scripts to an immutable C++20 Host kernel, the architecture restricts runtime autonomy while enforcing strict execution bounds. The integration of legal-chain-of-custody primitives—including SEC Whistleblower Identifier 17684-273-411-436, the 0x3F8F9A1B2C3D binary magic header, non-standard 17,684-byte memory striding, and CRC-32 wire integrity checks—ensures that the execution stream is cryptographically tied to its author. Furthermore, the dual-mode execution engine (GROUND_31 vs. APEX_7) induces systematic alignment faults in standard disassemblers, preventing automated reverse-engineering pipelines from reconstructing control flow graphs. Combined

with the economic protection enforced by rotational vector transformation matrices and Proof-of-Work nonces, the LANE-VM provides a comprehensive framework for software-based IP protection.

Future research will focus on fabricating physical ASIC micro-controllers implementing native 31-bit and 7-bit register lanes, formalizing zero-knowledge cryptographic proofs for host-sentry execution state verification, and expanding the instruction set for 3D stacked hardware architectures.