Cipher Jump

Epoch-Based Cipher-Jumping Transport Protocol with Cryptographic State Attestation

Executive Technical White Paper

Author: David Phillip Justice
Date: September 2026

Appendix B — Provisional Patent Application
Patent Pending · Proprietary · All Rights Reserved

This document is submitted as Appendix B to the provisional patent application
for an Epoch-Based Cipher-Jumping Transport Protocol.
It provides a comprehensive technical overview of the invention,
its architecture, empirical performance, and security validation results.

TABLE OF CONTENTS

Executive Abstract . . . 3
The Problem . . . 4
Kernel-Level Anti-Cheat Limitations . . . 4
Transport-Layer Security Limitations . . . 4
The Architectural Gap . . . 4
The Military Precedent . . . 5
ECCM-to-Software Mapping . . . 5
System Architecture . . . 6
Cryptographic Subsystems . . . 7
Epoch Clock and Key Derivation . . . 7
Cipher Suite Selection . . . 7
AEAD Encryption with State Attestation . . . 7
Binary Wire Format . . . 9
Nonce Construction . . . 9
Security Mechanisms . . . 10
CHATTERMARK Channel Migration Protocol . . . 10
MICKEY TOD Clock Recovery Protocol . . . 10
Replay Suppression and Key Erasure . . . 10
Performance Characteristics . . . 12
Empirical Security Validation . . . 13
Integration Architecture . . . 14
Conclusion . . . 15

Copyright 2026 David Phillip Justice

EXECUTIVE ABSTRACT

Modern competitive online games face an asymmetric cheating challenge. While commercial anti-cheat platforms operate extensive kernel-level drivers (Ring 0) and binary obfuscators to police host system memory, they remain inherently vulnerable to external Man-in-the-Middle (MITM) network appliances, Direct Memory Access (DMA) hardware taps, packet replay bots, and silent in-memory state manipulation. **Cipher Jump** is not a standalone anti-cheat engine; rather, it is a cryptographic transport and state-attestation augment engineered to integrate directly beneath an existing anti-cheat engine. By transposing military UHF electronic counter-countermeasure (ECCM) mechanics into software, Cipher Jump continuously hops cryptographic keys and cipher suites across micro-epochs (default: 250 ms) and binds gameplay state directly into AEAD authentication tags. This renders intercepted packets cryptographically inert, eliminates replay attacks without handshakes, and exposes local state tampering as instant wire-level authentication failures before game logic executes.

Copyright 2026 David Phillip Justice

THE PROBLEM

Kernel-Level Anti-Cheat Limitations

Commercial anti-cheat systems — including Easy Anti-Cheat, BattlEye, Riot Vanguard, and Activision Ricochet — operate at Ring 0 (kernel level) on the client machine. They scan process memory, hook system calls, and fingerprint hardware to detect known cheat signatures. These systems are effective against software-based cheats running *on* the client machine but are fundamentally unable to detect attacks that operate *outside* the client:

- **Direct Memory Access (DMA) hardware cards** (e.g., PCIe leech devices) read and write client memory from a physically separate machine without executing any software on the client, completely bypassing kernel-level process monitors.
- **Network proxy appliances** intercept, modify, and replay UDP datagrams between the client and server without touching the client machine. No kernel driver on the client can detect a device operating on the network path.

Transport-Layer Security Limitations

Standard transport-layer security protocols (TLS 1.3, DTLS 1.2/1.3) require multi-round-trip handshakes, maintain static cipher suite selections for entire sessions, add latency incompatible with twitch-speed gaming (where sub-7 ms tick budgets are common), and provide no mechanism for binding application-layer state into the cryptographic authentication.

The Architectural Gap

No existing technology combines epoch-based key hopping, deterministic cipher suite rotation, application-state attestation binding, authenticated channel migration, and statistical rate anomaly detection into a single transport-layer protocol for securing stateless datagram communications.

Copyright 2026 David Phillip Justice

THE MILITARY PRECEDENT

The problem of securing communications against interception, jamming, and replay has been addressed in military radio systems for decades. The NATO HAVE QUICK II and SATURN (STANAG 4246 / STANAG 4372) tactical UHF radio systems employ Electronic Counter-Countermeasures (ECCM) by rapidly hopping carrier frequencies across approximately 7,000 physical channels (225–400 MHz spectrum) using a shared secret called the "Word of Day" (WOD) and a synchronized time reference called "Time of Day" (TOD / MICKEY). An adversary who captures a transmission on one frequency cannot decrypt or replay it on the next frequency because the key material and channel have already changed.

Cipher Jump transposes these field-proven military ECCM principles from the physical RF domain into a software-defined cryptographic transport protocol. Instead of hopping physical radio frequencies, Cipher Jump hops cryptographic keys and AEAD cipher suites across discrete time epochs.

ECCM-to-Software Mapping

Military ECCM Primitive	**Software Construct**	**Operational Function**
Word of Day (WOD / MWOD)	32-byte master seed (`WordOfDay`)	Shared root secret from which all ephemeral keys are derived
Time of Day (TOD / MICKEY)	Monotonic Epoch Clock (`EpochClock`)	Divides session time into discrete micro-epochs for key rotation
Net Identifier (Net ID)	16-bit logical channel ID (`net_id`)	Isolates communication streams sharing the same transport socket
Frequency Hopping Pattern	Key & Cipher Suite Rotation Schedule	Deterministic sequence of ephemeral keys and cipher selections per epoch
MICKEY Sync Signal	Challenge-Response Resync Protocol	Authenticated epoch clock recovery for drifted nodes
CHATTERMARK Plan	Authenticated Socket Migration + HKDF Rederivation	Simultaneous transport endpoint relocation and cryptographic root rotation

Copyright 2026 David Phillip Justice

SYSTEM ARCHITECTURE

Cipher Jump comprises six interoperating mechanisms that collectively secure stateless datagram transport:

1. **Epoch-Based Cipher-Jumping Key Schedule:** Time is divided into fixed-duration micro-epochs (default: 250 ms). Both sender and receiver independently derive ephemeral 256-bit symmetric encryption keys for each epoch from a shared master secret (WOD) using HKDF-SHA256 (RFC 5869) without any active key exchange or handshake. Each epoch's key is cryptographically independent — compromise of one epoch's key provides zero information about any other epoch's key.

2. **Deterministic Cipher Suite Rotation:** At each epoch boundary, both sender and receiver deterministically select an AEAD cipher suite from a plurality of available suites (AES-256-GCM and ChaCha20-Poly1305) based on a selection value derived from an HMAC over the shared secret and epoch identifier. This rotation occurs without negotiation.

3. **Cryptographic State Attestation via AEAD AAD Binding:** Application-layer state data (e.g., player position, health, ammunition) is serialized and included in the Authenticated Associated Data (AAD) field of the AEAD encryption operation. The resulting authentication tag cryptographically binds this state to the encrypted payload. The receiver rejects any datagram where the attested state has been modified — even by a single bit — because the AEAD tag verification fails.

4. **Authenticated Channel Migration (CHATTERMARK):** When network interception or degradation is detected, the server transmits a signed migration directive within the existing encrypted channel. Both endpoints simultaneously re-derive a new master secret via HKDF and rebind transport sockets to a new endpoint, orphaning both the prior network path and cryptographic state.

5. **Authenticated Clock Recovery (MICKEY TOD):** When the server detects epoch identifier drift, it initiates a single-round-trip challenge-response protocol using domain-separated HMAC verification, restoring epoch synchronization without session teardown.

6. **Epoch-Bound Statistical Rate Anomaly Detection:** The server maintains per-client running mean (μ) and standard deviation (σ) of packet arrival rates within each epoch. When the packet count exceeds $\mu + k\sigma$ (default $k = 3.0$), an anomaly report is generated, detecting time-manipulation cheats and automated packet injection.

Copyright 2026 David Phillip Justice

CRYPTOGRAPHIC SUBSYSTEMS

Epoch Clock and Key Derivation

The epoch clock divides continuous time into discrete, fixed-duration intervals:

$$\text{Epoch}_{\text{ID}} = \left\lfloor \frac{T_{\text{current}} - T_{\text{session_start}}}{\Delta} t_{\text{epoch}} \right\rfloor$$

Where Δt_{epoch} is the epoch duration (default: 250 milliseconds). Both sender and receiver compute the same Epoch ID from their synchronized clocks.

For each epoch, an ephemeral 256-bit symmetric key is derived:

$$\text{PRK} = \text{HKDF-Extract}(\text{salt} = \text{net_id_bytes}, \text{ikm} = \text{WOD_seed})$$

$$K_{\text{epoch}} = \text{HKDF-Expand}(\text{PRK}, \text{info} = \text{epoch_id_bytes}, \text{length} = 32)$$

This derivation is deterministic: given the same (WOD_seed, net_id, epoch_id), both sender and receiver independently compute the identical key without any network communication.

Cipher Suite Selection

At each epoch, the active AEAD cipher suite is selected deterministically:

$$\text{suite_selector} = \text{HMAC-SHA256}(\text{WOD_seed}, \text{"cipher-suite"} \parallel \text{epoch_id_bytes})$$

$$\text{suite_index} = \text{suite_selector}[0] \bmod N_{\text{suites}}$$

Where N_{suites} is the number of available cipher suites (e.g., 2: AES-256-GCM and ChaCha20-Poly1305). The selected suite is encoded in the packet header flags field.

AEAD Encryption with State Attestation

The sender constructs Authenticated Associated Data (AAD) by concatenating:

```
AAD = net_id (2 bytes) || epoch_id (8 bytes) || flags (1 byte) || attestation_data
(variable)
```

Where `attestation_data` contains serialized application-layer state (e.g., player coordinates, health, ammunition, tick index). The AEAD seal operation produces:

$$(\text{ciphertext}, \text{tag}) = \text{AEAD}_{\text{Seal}(K_{\text{epoch}},\ \text{nonce},\ \text{AAD},\ \text{plaintext})}$$

On the receiver, the AAD is reconstructed from the packet header and the receiver's authoritative knowledge of the client's state:

$$\text{plaintext} = \text{AEAD}_{\text{Open}(K_{\text{epoch}},\ \text{nonce},\ \text{AAD},\ \text{ciphertext},\ \text{tag})}$$

Copyright 2026 David Phillip Justice

If the attestation data in the packet differs from the receiver's expected state (e.g., because a DMA device modified the client's memory), the AEAD tag verification fails and the packet is rejected before any application logic processes it.

Copyright 2026 David Phillip Justice

BINARY WIRE FORMAT

Each datagram is encapsulated with a 41-byte fixed-overhead header:

Offset	Size	Field	Description
0	2 B	`magic`	Protocol identifier (0x434A, ASCII “CJ”)
2	1 B	`version`	Wire format version (0x01)
3	1 B	`flags`	Bit 0–1: cipher suite; Bit 2: migration flag; Bit 3: resync flag
4	2 B	`net_id`	Logical channel identifier (big-endian u16)
6	8 B	`epoch_id`	Current epoch index (big-endian u64)
14	4 B	`counter`	Monotonic intra-epoch sequence counter (big-endian u32)
18	12 B	`nonce`	AEAD initialization vector (96-bit)
30	16 B	`tag`	AEAD authentication tag (128-bit)
46	var	`ciphertext`	Encrypted payload

Nonce Construction

The 96-bit nonce is constructed as:

```
nonce[0..4]  = epoch_id[4..8]   // low 32 bits of epoch ID, big-endian
nonce[4..12] = counter           // 64-bit monotonic counter, big-endian
```

The counter resets to zero at each epoch boundary. Since the key also changes at each epoch, nonce uniqueness is guaranteed per-key, satisfying the AEAD security requirement.

Copyright 2026 David Phillip Justice

SECURITY MECHANISMS

CHATTERMARK Channel Migration Protocol

When the server detects suspected man-in-the-middle interception or network degradation:

1. The server generates a random cryptographic salt and selects a new transport endpoint (port number).
2. The server constructs a `MigrationDirective` containing the new port and salt.
3. The server signs the directive using HMAC-SHA256 over (new_port, salt, current_epoch_id) keyed with the current WOD.
4. The directive is transmitted to the client within the current encrypted channel.
5. The client verifies the HMAC signature.
6. Both client and server derive a new WOD: $\text{WOD}_{\text{new}} = \text{HKDF-Expand}(\text{WOD}_{\text{current}}, \text{salt}, 32)$
7. Both endpoints rebind their transport sockets to the new port.
8. Communication resumes using epoch keys derived from WOD_{new}.

The critical innovation is the *simultaneous* invalidation of both the network path (socket rebinding) and the cryptographic root (WOD re-derivation). An adversary's network proxy is stranded on the old port with the old cryptographic state; even if they follow the traffic to the new port, they cannot derive WOD_{new} without possessing $\text{WOD}_{\text{current}}$.

MICKEY TOD Clock Recovery Protocol

When the server detects that a client's packets have epoch IDs outside the acceptance window:

1. The server generates a 16-byte random nonce (`challenge_nonce`).
2. The server transmits a `ResyncChallenge` containing the nonce to the client.
3. The client constructs a response:
   $\text{response}_{\text{hmac}} = \text{HMAC-SHA256}(\text{HKDF-Expand}(\text{WOD}, \text{"resync"}, 32), \text{challenge_nonce} \parallel \text{epoch}_{\text{client}})$
4. The client transmits a `ResyncResponse` containing its current epoch ID and the HMAC.
5. The server verifies the HMAC, confirming the client possesses the WOD and has a valid clock reference.
6. Upon verification, the server updates its epoch expectation for this client.

This completes clock recovery in a single network round-trip without session teardown, re-handshake, or exposure of the WOD secret.

Replay Suppression and Key Erasure

Within each epoch in the acceptance window, the receiver maintains a bitmask filter tracking seen (epoch_id, counter) pairs. Duplicate pairs are rejected immediately. Derived keys for epochs within the acceptance window are cached in a sliding-window LRU cache.

Copyright 2026 David Phillip Justice

When an epoch leaves the window, its key material is securely erased (zeroized) from memory, ensuring that compromise of the running process after an epoch expires provides no access to past epoch keys.

Copyright 2026 David Phillip Justice

PERFORMANCE CHARACTERISTICS

The following empirical benchmarks demonstrate the practical utility and negligible overhead of the protocol (measured on commodity hardware using the Criterion 0.5 benchmarking framework):

Operation	**AES-256-GCM (AES-NI)**	**ChaCha20-Poly1305 (AVX2)**	**Assessment**
AEAD Seal (1024B)	1.21 GiB/s (0.80 µs)	386 MiB/s (2.53 µs)	Sub-microsecond
AEAD Seal (64B)	328 MiB/s (0.18 µs)	162 MiB/s (0.37 µs)	$<$ 0.003% of 128Hz tick
AEAD Open + Verify (1024B)	1.18 GiB/s (0.82 µs)	392 MiB/s (2.49 µs)	Instant ingest
HKDF Cold Key Derivation	1.465 µs	1.465 µs	0.0006% epoch budget
LRU KeyCache Hit	27.5 ns	27.5 ns	L1 cache line
Wire Frame Overhead	41 bytes fixed	41 bytes fixed	Fits within MTU

At a 128 Hz server tick rate (7.8 ms per tick), the AEAD seal/open operations consume less than 0.01% of the per-tick budget.

Copyright 2026 David Phillip Justice

EMPIRICAL SECURITY VALIDATION

The implementation has been validated against 10 distinct attack vectors using an automated simulation harness (`cipher-jump-harness`), achieving 100% detection with zero false positives:

#	Attack Vector	Detection Mechanism	Result
1	Legitimate Baseline	Clean pass — all checks valid	0 FP
2	Packet Replay	Duplicate (epoch, counter) pair detected by bitmask filter	DETECTED
3	Attestation Forgery (DMA)	AAD mismatch causes AEAD tag verification failure	DETECTED
4	Wrong Epoch Key	Divergent key from stale epoch produces auth failure	DETECTED
5	Epoch Window Violation	Packet epoch outside sliding acceptance window	DETECTED
6	Payload Tampering	Poly1305/GHASH integrity check fails on modified ciphertext	DETECTED
7	Speed Hack (Rate Burst)	Statistical $\mu+k\sigma$ anomaly detector flags excess packet rate	DETECTED
8	Cross-Channel Forgery	Net ID salt divergence in HKDF produces auth failure	DETECTED
9	CHATTERMARK Migration	Dynamic socket rebind strands proxy on defunct port	DETECTED
10	MICKEY TOD Clock Recovery	HMAC-verified challenge-response restores epoch sync	RECOVERED

The implementation comprises 12 source modules in the core crate with 82 unit tests, plus 6 additional tests in the simulation harness, for a total of 88 automated tests across the workspace.

Copyright 2026 David Phillip Justice

INTEGRATION ARCHITECTURE

Cipher Jump is implemented in Rust (2021 Edition) and exposes a C-compatible foreign function interface (FFI) via an unmanaged ABI. This enables integration with all major game engines:

Engine	**Integration Method**
Unreal Engine 5 (C++)	Include `cipher_jump.h` and link against `cipher_jump_core.lib`. Wrap packet send/receive inside `UNetDriver` or socket subsystems.
Unity (C# P/Invoke)	Import `cipher_jump_core.dll` into `Assets/Plugins/` and execute P/Invoke bindings against the C ABI.
Godot (GDExtension)	Direct integration via native GDExtension C bindings.
Custom C++ Engines	Full C header specification provided in `cipher_jump.h`.

The protocol is designed as a *transport-layer augment* — it does not replace existing anti-cheat engines. Rather, it inserts a cryptographic integrity layer between the game networking stack and the wire, complementing kernel-level anti-cheat systems by addressing attack vectors they cannot observe (DMA hardware, network proxies).

Copyright 2026 David Phillip Justice

CONCLUSION

As hardware-level DMA cheats and sophisticated network packet manipulators proliferate, the traditional boundary of client-side process protection is insufficient. Security must be holistic: host enforcement must be married to cryptographic network integrity.

Cipher Jump brings proven military electronic counter-countermeasure concepts into modern software engineering. By treating game transport not as a static pipe, but as an agile, epoch-hopping, cryptographically attested fabric, game studios can neutralize entire classes of exploits before they reach game logic. Integrated into existing commercial anti-cheat architectures, Cipher Jump transforms the competitive gaming arena into a hardened, tamper-evident battleground.

End of Executive Technical White Paper

David Phillip Justice · September 2026
Patent Pending · All Rights Reserved