COLDSTAR

Ephemeral Security for Blockchain Signing

By ChainLabs Technologies

Abstract

ColdStar introduces a fundamentally new security paradigm for cryptocurrency key management. Rather than trusting permanent hardware devices, ColdStar shifts the trust anchor to open-source, auditable software and ephemeral processes. Private keys exist only in protected volatile memory for approximately 100 microseconds during signing operations, then are immediately zeroized. This approach eliminates the attack vectors inherent in permanent key storage while maintaining full automation capabilities for professional workflows.

The Hardware Wallet Paradox

A Permanent Key Creates a Permanent Target

Hardware wallets improved security but introduced a new class of risks centered on a single, permanent physical device. Trust is placed in hardware that can be compromised at multiple points:

Risk Category	Description
Supply Chain Risk	Compromise during manufacturing or shipping
Vendor Trust	Reliance on a single company's integrity and competence
Firmware Exploits	Vulnerabilities in the device's closed-source operating system
Poor Automation	Manual, GUI-driven workflows unsuitable for scripting or CI/CD
Physical Attack Surface	A persistent device that can be lost, seized, or physically manipulated

---

A New Philosophy

Trust Auditable Code, Not Opaque Hardware

ColdStar challenges the traditional model by removing permanent trusted hardware entirely. The trust anchor is shifted from a physical object to open-source software and ephemeral processes.

Instead of Trusting Devices...

We reject:

- Permanent key storage
- Vendor trust assumptions
- Proprietary chips

ColdStar Trusts:

- Open-source, auditable software
- User-controlled operating systems
- Extremely short-lived key exposure in RAM

> ***The USB drive is not a signing device. It is disposable, deniable, encrypted storage only.***

The ColdStar Model

Ephemeral Keys, Disposable Storage

```
USB Storage          Protected RAM          Zeroization
     |                    |                      |
     v                    v                      v
[Encrypted Key] --> [Decrypt to RAM] --> [Sign] --> [Overwrite with 0x00]
     |                    |                      |
 Permanent             ~100 microseconds      Instant
```

1. **A standard USB stores a single, encrypted key file**
2. **The key is decrypted *only* into a protected, locked segment of system RAM for the brief moment of signing**
3. **All plaintext key material is instantly and automatically zeroized (overwritten) from memory after use**

> *At no point does any powered device permanently store a usable private key.*

Key Generation Flow

From Hardware RNG to Encrypted File

Hardware: CPU + System RAM (Volatile)

```
Hardware RNG
    |
    v  32-byte Ed25519/secp256k1 seed generated in CPU registers
    |
    v
`mlock`'d RAM
    |
    v  Seed is copied to a locked, non-swappable memory buffer
    |                                              [PLAINTEXT IN RAM]
    v
Passphrase & KDF
    |
    v  Passphrase + Salt --> Argon2id --> AES Key
    |                                              [PLAINTEXT HERE]
    v
AES-256-GCM Encryption
    |
    v  The CPU encrypts the private key seed using the derived AES key
    |
    v
Immediate Zeroization
    |
    v  All plaintext materials (private key seed, derived AES key,
       passphrase) are overwritten with zeros in RAM
```

Hardware: USB Flash Storage (Non-Volatile)

```
{
  "version": 1,
  "ciphertext": "...",
  "salt": "...",
  "nonce": "...",
  "public_key": "..."
}
```

ENCRYPTED ONLY - The final output is a JSON file containing only the ciphertext, salt, nonce, and public key.

The Signing Flow

A 100 Microsecond Lifespan

Step 1: USB to Application

Encrypted data is read from the USB into the application's memory space.

NO PLAINTEXT KEY IN APPLICATION MEMORY

Step 2: FFI Call to Rust

The encrypted payload is passed to an isolated Rust environment via FFI.

Step 3: Key Re-derivation

In a new `mlock`'d Rust buffer, Argon2id re-derives the AES key.

```
Passphrase + Salt --> Argon2id
```

PLAINTEXT HERE (in locked RAM only)

Step 4: AES-256-GCM Decryption

The private key is decrypted into the locked Rust buffer.

PLAINTEXT KEY ONLY IN LOCKED RAM

Step 5: Ed25519/secp256k1 Signing

The CPU uses the plaintext key to sign the transaction.

```
Plaintext lifespan: ~100 microseconds
```

Step 6: Immediate Zeroization

The Rust `Drop` trait guarantees all plaintext (key, derived key, passphrase) is overwritten with zeros, even on panic/crash.

FFI Return

Only the public 64-byte signature is returned to the application.

Private key NEVER entered the application heap.

Security Mechanisms

`mlock()` - Preventing Swap to Disk

```
Standard RAM Page           `mlock`'d RAM Page
       |                           |
       v                           X
  Disk Swap File              Disk Swap File
```

By locking the memory page, the OS is prevented from swapping it out to disk, ensuring sensitive data remains only in volatile RAM.

Memory Zeroization - No Forensic Trace

```
Before                       After
0x42 0x1F 0x9A 0xE3   -->   0x00 0x00 0x00 0x00
0x5B 0xC0 0xD7 0xA7         0x00 0x00 0x00 0x00
```

Overwrites data before deallocation to prevent recovery from heap analysis or cold boot attacks.

Process Memory Isolation - Containing the Key

```
                    OS Kernel
     +--------------------|--------------------+
     |                    |                    |
     v                    v                    v
Application         Rust Library         Other Process
   heap          `mlock`'d buffer             X
     |           (plaintext key)         [blocked]
     +----FFI call--->|
```

The OS enforces memory isolation, ensuring that the key remains contained within the Rust environment and inaccessible to other processes.

Where the Private Key Lives

Location	Data State	Duration
USB NAND Flash	Encrypted	Permanent (until deleted)
Application RAM	Encrypted	Seconds
Rust `mlock`'d RAM	**PLAINTEXT**	**~100 microseconds**
CPU Registers/Cache	**PLAINTEXT**	**~10 nanoseconds**

Security Guarantees and Operational Assumptions

What IS Protected Against

- Long-lived key exposure
- Firmware backdoors & hardware supply-chain manipulation
- Persistent device compromise
- Physical seizure or fingerprinting of the signing medium

What is NOT Protected Against (Assumptions)

- Memory dumps while the key is in RAM (requires root + precise timing)
- A fully compromised operating system with kernel-level access
- Hardware keyloggers capturing the passphrase
- Cold boot attacks (if RAM is physically extracted within seconds)

> *ColdStar is explicit about its security boundaries and does not hide assumptions behind hardware abstractions.*

ColdStar vs. Hardware Wallets

Aspect	Hardware Wallet	ColdStar
Persistent key storage	Yes (secure element)	No (RAM only during signing)
Physical attack surface	Permanent device	Disposable USB + computer RAM
Decryption location	Inside secure chip	System RAM (`mlock`'d)
Key lifetime in plaintext	Years (powered on)	**Microseconds**
Supply chain risk	High (proprietary hardware)	**Low** (commodity USB + open source)
Automation Support	Poor / Manual	**CLI-first and fully scriptable**
Cost	$50-$200	~$5 commodity USB

Target Users

Engineered for Automation, Built for Experts

ColdStar is designed for developers, traders, and operators who require verifiability and explicit control. It is CLI-first by design, with no GUI dependency, browser extension, or background daemon.

Intended Workflows

- Headless Environments & CI/CD Pipelines
- Automated Trading Systems
- Air-gapped Workflows
- Deterministic Scripting

> *This is not a consumer wallet and does not aim to be one.*

Supported Blockchains

Solana (Ed25519)

- Native Solana transactions
- SPL tokens
- Stablecoins
- Tokenized commodities (e.g., PAXG)
- Tokenized equities (xStocks)
- Custom program instructions
- Solana staking and delegation

Ethereum (secp256k1)

- Native ETH transactions
- ERC-20 tokens
- Smart contract interactions
- EIP-712 typed data signing

Algorand (Ed25519)

- Native Algorand transactions
- ASA (Algorand Standard Assets)
- Smart contract calls
- Multisig operations

Intelligent Boot Process

Zero-Intervention Wallet Integrity

Every time a ColdStar USB is used, the system automatically verifies and maintains the integrity of critical wallet files, ensuring seamless and safe operation across different machines.

Automatic Actions

- Verifies critical wallet files
- Restores missing or corrupted files from on-device backup
- Creates/updates backups of valid files
- Tracks boot instances to optimize performance

USB Storage Structure

```
USB Drive/
  |-- wallet/
  |     |-- keypair.json
  |     +-- pubkey.txt
  |-- inbox/
  |-- outbox/
  +-- .coldstar/
        |-- last_boot_id
        +-- backup/
```

Getting Started

Verify Everything

ColdStar is open-source and designed to be fully inspectable. Security claims are meant to be verifiable, not trusted. The installer automatically handles all dependencies.

One-Command Installation

Windows (PowerShell)

```
.\install.ps1
```

macOS / Linux (Terminal)

```
chmod +x install.sh && ./install.sh
```

Repository

GitHub: `github.com/devsyrem/coldstar-distro-build-mc`

Conclusion

Hardware wallets were an improvement. But compared to open-source software, ColdStar removes the final dependency: **the device itself**.

> ***Decouple your security from your hardware.***

The ColdStar Philosophy

True sovereignty isn't owning a key.

It's creating one... and letting it disappear.

Resources

- **Main Repository**: github.com/devsyrem/coldstar-distro-build-mc
- **Technical Whitepaper**: Available in PDF format
- **Blockchain-Specific Guides**: Solana, Ethereum, Algorand

ColdStar by ChainLabs Technologies

Ephemeral Security for the Modern Blockchain Era