COLDSTAR

A Python-Based Air-Gapped Cold Wallet
Tool for Solana

Air-Gap Isolation

Ed25519 Cryptography

Alpine Linux (~50MB)

100% Open Source

</Syrem>

ChainLabs Technologies

COLDSTAR

Technical Whitepaper

TECHNICAL WHITEPAPER • V1.0

COLDSTAR

A Python-Based Air-Gapped Cold Wallet
Tool for Solana

</Syrem>

ChainLabs Technologies

Purple Squirrel Media

COLDSTAR: A Python-Based Air-Gapped Cold Wallet Tool for Solana
Technical Whitepaper — Current Implementation

Version 1.0.0
December 2025

Author: </Syrem>
Organization: ChainLabs Technologies

Published by Purple Squirrel Media

First Edition

Contents

ABSTRACT

Abstract

Coldstar is an open-source, command-line cold wallet management tool for the Solana blockchain that prioritizes security through air-gapped transaction signing. This whitepaper describes the current implementation—a Python-based CLI application that enables users to create bootable USB cold wallets, generate Ed25519 keypairs offline, and sign transactions without network exposure. By leveraging Alpine Linux as a minimal operating system and industry-standard cryptographic libraries, Coldstar provides enterprise-grade security with full code transparency and reproducibility.

Key Features

- Complete air-gap isolation during private key generation and transaction signing

- Minimal Alpine Linux (~50MB) bootable USB creation

- Python-based terminal interface with rich UI components

- Ed25519 cryptographic operations using audited libraries

- Offline transaction signing with online broadcasting separation

- Cross-platform support (Windows, Linux, macOS)

SECTION 1

Introduction

1.1 Motivation

The security of cryptocurrency holdings fundamentally depends on private key protection. Despite advances in wallet technology, users face a critical dilemma: hot wallets (online) offer convenience but expose keys to network attacks, while cold storage (offline) provides security at the cost of usability. Hardware wallets attempt to bridge this gap but introduce vendor dependencies, closed-source firmware risks, and cost barriers.

1.2 Coldstar's Solution

Coldstar addresses these challenges through a software-defined cold wallet approach:

- No Proprietary Hardware: Works with commodity USB drives ($5-10)

- Full Transparency: 100% open-source Python codebase

- Air-Gap Architecture: Private keys never touch internet-connected systems

- Minimal OS: Alpine Linux reduces attack surface to ~50MB

- Developer-Friendly: Command-line interface for power users and automation

1.3 Scope of This Document

This whitepaper documents the current implementation (v1.0) of Coldstar—a Python CLI tool. It covers:

- Architecture and component design

- Cryptographic implementation details

- Security model and threat mitigation

- Operational procedures and workflows
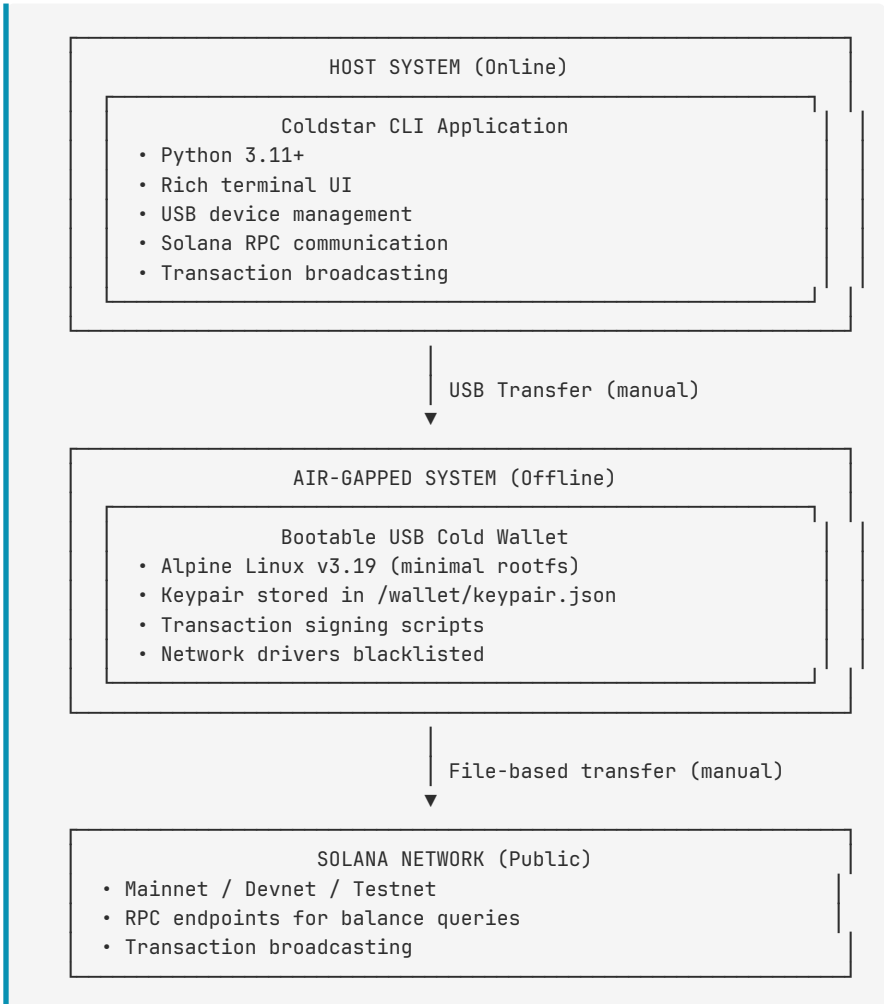
- Performance characteristics

This document does NOT cover future enhancements or planned features. It represents a snapshot of the production-ready system as deployed.

System Architecture

2.1 High-Level Overview

Coldstar employs a three-component architecture:

```
                    HOST SYSTEM (Online)

                   Coldstar CLI Application
          • Python 3.11+
          • Rich terminal UI
          • USB device management
          • Solana RPC communication
          • Transaction broadcasting


                       │  USB Transfer (manual)
                       ▼

                  AIR-GAPPED SYSTEM (Offline)

                   Bootable USB Cold Wallet
          • Alpine Linux v3.19 (minimal rootfs)
          • Keypair stored in /wallet/keypair.json
          • Transaction signing scripts
          • Network drivers blacklisted


                       │  File-based transfer (manual)
                       ▼

                   SOLANA NETWORK (Public)
          • Mainnet / Devnet / Testnet
          • RPC endpoints for balance queries
          • Transaction broadcasting
```

2.2 Component Breakdown

2.2.1 CLI Application (Python)

The command-line interface provides the following functionality:

Core Modules:

Module	Responsibility	Lines of Code
`main.py`	Application entry point, menu orchestration	~980
`src/wallet.py`	Keypair generation, storage, loading	~140
`src/transaction.py`	Transaction creation, signing, serialization	~210
`src/network.py`	Solana RPC client, balance queries	~150
`src/usb.py`	USB device detection, mounting	~200
`src/iso_builder.py`	Alpine Linux ISO creation	~250
`src/ui.py`	Terminal UI components (Rich library)	~240
`config.py`	Configuration constants	~50

Total: ~2,220 lines of Python code

Dependencies:

- `rich` - Terminal UI rendering
- `questionary` - Interactive prompts
- `solana` - Solana Python SDK
- `solders` - Rust-based Solana types (faster, safer)
- `pynacl` - Ed25519 cryptographic operations
- `httpx` - Async HTTP client for RPC
- `base58` - Address encoding

2.2.2 USB Cold Wallet Device

Operating System: Alpine Linux v3.19 x86_64

Filesystem Layout:

```
/
├── wallet/                     # Keypair storage
│   ├── keypair.json            # Ed25519 private key (64 bytes as JSO
N array)
│   └── pubkey.txt              # Base58-encoded public key
├── inbox/                      # Unsigned transactions (copied from h
ost)
├── outbox/                     # Signed transactions (copied to host)
├── scripts/
│   └── sign_tx.sh              # Transaction signing helper script
├── boot/                       # GRUB bootloader
├── etc/
│   ├── network.blacklist       # Disabled kernel modules
│   └── fstab                   # Filesystem mounting rules
└── [Alpine Linux minimal rootfs]
```

Size: ~50MB (vs. ~4GB for Ubuntu Server)

Security Customizations:

- Network kernel modules blacklisted (`e1000`, `iwlwifi`, `r8169`, etc.)

- Read-only root filesystem

- No package manager accessible

- Minimal process footprint

2.2.3 Data Flow

Wallet Creation Flow:

1. CLI detects USB device → Confirms with user

2. CLI downloads Alpine rootfs → Validates checksum

3. CLI creates custom ISO with wallet structure

4. User flashes ISO to USB using external tool (Rufus, dd, balenaEtcher)

5. User boots air-gapped PC from USB

6. Boot script generates Ed25519 keypair → Stores in `/wallet/`

7. Public key displayed on screen → User records it

Transaction Signing Flow:

1. User creates transaction on online host (CLI)

2. CLI builds unsigned transaction → Saves to file

3. User manually copies file to USB `/inbox/` (physical transfer)

4. User boots USB on air-gapped system

5. User runs `sign_tx.sh` script → Signs with private key

6. Signed transaction written to `/outbox/`

7. User copies signed TX back to online host

8. CLI broadcasts signed transaction → Solana network

Implementation Details

3.1 Keypair Management (`src/wallet.py`)

Generation:

```python
from solders.keypair import Keypair

class WalletManager:
    def generate_keypair(self) -> Tuple[Keypair, str]:
        """Generate Ed25519 keypair using OS entropy."""
        self.keypair = Keypair()  # Uses /dev/urandom on Linux/macOS
        public_key = str(self.keypair.pubkey())
        return self.keypair, public_key
```

Storage Format:

```
[
  123, 45, 67, 89, 12, 34, ...  // 64-byte array
  // Bytes 0-31: Ed25519 seed (private key)
  // Bytes 32-63: Public key (derived)
]
```

Security Measures:

- File permissions: `chmod 600` (owner read/write only)
- Keypair never logged or transmitted over network
- Optional: User can encrypt keypair file with GPG

3.2 Transaction Operations (`src/transaction.py`)

Creating Unsigned Transactions:

```python
def create_transfer_transaction(
    self,
    from_pubkey: str,
    to_pubkey: str,
    amount_sol: float,
    recent_blockhash: str
) -> Optional[bytes]:
    """Build unsigned SOL transfer transaction."""
    from_pk = Pubkey.from_string(from_pubkey)
    to_pk = Pubkey.from_string(to_pubkey)
    lamports = int(amount_sol * LAMPORTS_PER_SOL)
    blockhash = Hash.from_string(recent_blockhash)

    # Create transfer instruction
    transfer_ix = transfer(TransferParams(
        from_pubkey=from_pk,
        to_pubkey=to_pk,
        lamports=lamports
    ))

    # Build message and transaction
    message = Message.new_with_blockhash([transfer_ix], from_pk, bl
ockhash)
    tx = Transaction.new_unsigned(message)

    self.unsigned_tx = bytes(tx)
    return self.unsigned_tx
```

Signing Transactions:

```python
def sign_transaction(self, keypair: Keypair) -> Optional[bytes]:
    """Sign transaction with Ed25519 keypair."""
    if not self.unsigned_tx:
        return None

    # Deserialize unsigned transaction
    tx = Transaction.from_bytes(self.unsigned_tx)

    # Sign message
    tx.sign([keypair], tx.message.recent_blockhash)

    self.signed_tx = bytes(tx)
    return self.signed_tx
```

3.3 Network Communication (`src/network.py`)

RPC Client:

```python
from solana.rpc.async_api import AsyncClient
from solana.rpc.commitment import Confirmed

class SolanaNetwork:
    def __init__(self, rpc_url: str = SOLANA_RPC_URL):
        self.rpc_url = rpc_url
        self.client = AsyncClient(rpc_url, commitment=Confirmed)

    async def get_balance(self, public_key: str) -> Optional[float]:
        """Query wallet balance in SOL."""
        pubkey = Pubkey.from_string(public_key)
        response = await self.client.get_balance(pubkey)

        if response.value is not None:
            return response.value / LAMPORTS_PER_SOL
        return None

    async def broadcast_transaction(self, signed_tx: bytes) -> Optional[str]:
        """Submit signed transaction to network."""
        response = await self.client.send_raw_transaction(signed_tx)
        return str(response.value) if response.value else None
```

Supported Networks:

- Mainnet-Beta: `https://api.mainnet-beta.solana.com`

- Devnet: `https://api.devnet.solana.com`

- Testnet: `https://api.testnet.solana.com`

- Custom RPC: Configurable in `config.py`

3.4 USB Device Management (`src/usb.py`)

Detection (Windows):

```python
    def detect_usb_devices_windows(self) -> List[Dict]:
        """Use WMIC to detect USB drives."""
        cmd = 'wmic diskdrive where "InterfaceType=\'USB\'" get Caption
,DeviceID,Size'
        result = subprocess.run(cmd, shell=True, capture_output=True, t
ext=True)

        devices = []
        for line in result.stdout.splitlines()[1:]:
            if line.strip():
                parts = line.split()
                devices.append({
                    'device': parts[-2],
                    'model': ' '.join(parts[:-2]),
                    'size': self._format_size(int(parts[-1]))
                })
        return devices
```

Detection (Linux):

```python
    def detect_usb_devices_linux(self) -> List[Dict]:
        """Use lsblk to detect USB drives."""
        cmd = ['lsblk', '-d', '-o', 'NAME,SIZE,TRAN,MODEL', '-J']
        result = subprocess.run(cmd, capture_output=True, text=True)

        devices = []
        data = json.loads(result.stdout)
        for device in data.get('blockdevices', []):
            if device.get('tran') == 'usb':
                devices.append({
                    'device': f"/dev/{device['name']}",
                    'size': device['size'],
                    'model': device.get('model', 'Unknown')
                })
        return devices
```

3.5 ISO Building (`src/iso_builder.py`)

Alpine Linux Acquisition:

```python
def download_alpine_rootfs(self):
    """Download and verify Alpine Linux minimal rootfs."""
    url = ALPINE_MINIROOTFS_URL
    output_file = self.work_dir / "alpine-minirootfs.tar.gz"

    # Download with progress bar
    with httpx.stream("GET", url) as response:
        total = int(response.headers.get("content-length", 0))
        with open(output_file, "wb") as f:
            for chunk in response.iter_bytes():
                f.write(chunk)

    # Verify checksum (if available)
    self._verify_checksum(output_file)
    return output_file
```

Customization Process:

```python
def create_cold_wallet_iso(self, output_path: str):
    """Build bootable ISO with cold wallet structure."""
    # 1. Extract Alpine rootfs
    self._extract_rootfs()

    # 2. Create wallet directory structure
    create_wallet_structure(self.rootfs_dir)

    # 3. Blacklist network modules
    self._blacklist_network_drivers()

    # 4. Add signing scripts
    self._install_signing_scripts()

    # 5. Install Solana CLI tools
    self._install_solana_cli()

    # 6. Create bootloader
    self._create_grub_bootloader()

    # 7. Generate ISO image
    self._generate_iso(output_path)
```

3.6 User Interface (`src/ui.py`)

Terminal Rendering:

```
from rich.console import Console
from rich.panel import Panel
from rich.table import Table

def print_wallet_info(public_key: str, balance: Optional[float] = N
one):
    """Display wallet information in formatted panel."""
    table = Table(box=DOUBLE, show_header=False, border_style="cyan
")
    table.add_column("Field", style="dim")
    table.add_column("Value", style="green bold")

    table.add_row("Public Key", public_key)
    if balance is not None:
        table.add_row("Balance", f"{balance:.9f} SOL")

    panel = Panel(
        table,
        title="[bold cyan]WALLET INFORMATION[/bold cyan]",
        border_style="cyan"
    )
    console.print(panel)
```

Interactive Menus:

```
import questionary

def select_menu_option(options: List[str], message: str) -> str:
    """Display interactive selection menu."""
    return questionary.select(
        message,
        choices=options,
        style=CUSTOM_STYLE
    ).ask()
```

Security Model

4.1 Threat Model

Assumptions:

1. Host PC is compromised: Assume malware, keyloggers, network sniffers

2. Network is adversarial: Assume MITM attacks, DNS hijacking

3. USB device is physically secure: User maintains custody

4. Air-gapped PC is clean: No network connectivity, trusted hardware

Trust Boundaries:

- Trusted Zone: Air-gapped USB device, private keys

- Untrusted Zone: Online host PC, network infrastructure

- Transfer Mechanism: Manual file copying (physical air-gap)

4.2 Security Principles

Principle 1: Air-Gap Isolation

Implementation:

- Private keys generated ONLY on air-gapped device

- No network drivers loaded on Alpine Linux

- Physical separation between online and offline systems

- Manual file transfer prevents automated exfiltration

Threat Mitigation:

- ☒ Remote key theft impossible (no network)

- ☒ Malware cannot auto-exfiltrate keys

- ☒ Zero-day network exploits ineffective

Principle 2: Minimal Attack Surface

Implementation:

- Alpine Linux: ~50MB (vs. ~4GB Ubuntu)
- Only essential packages included
- No SSH daemon, no package manager, no unnecessary services
- Read-only filesystem prevents tampering

Threat Mitigation:

- ☒ Fewer vulnerabilities (smaller codebase)
- ☒ Limited post-exploitation options
- ☒ Reduced zero-day exposure

Principle 3: Code Transparency

Implementation:

- 100% open-source Python code (MIT License)
- No obfuscated binaries or compiled executables
- Reproducible ISO builds (checksums published)
- Community audit capability

Threat Mitigation:

- ☒ Supply chain backdoors detectable
- ☒ Hidden vulnerabilities discoverable
- ☒ Independent verification possible

Principle 4: Cryptographic Best Practices

Implementation:

- Ed25519 signatures (industry standard)

- NaCl library (audited, constant-time)

- OS entropy source (`/dev/urandom`)

- Deterministic signing (RFC 8032)

Threat Mitigation:

- ⊠ Timing attacks prevented

- ⊠ Weak RNG avoided

- ⊠ Signature forgery infeasible (2^128 security)

4.3 Attack Vectors & Mitigations

Attack	Likelihood	Impact	Mitigation	Effectiveness
Malware on Host PC	High	Critical	Keys never on host; only signed TXs transferred	⊠ Very High
Compromised Alpine ISO	Very Low	Critical	Checksum verification; reproducible builds	⊠ High
Evil Maid (USB tampering)	Low	High	Tamper-evident seals; boot verification	⊠⊠ Medium
Side-Channel (timing)	Very Low	Medium	Constant-time crypto (NaCl)	⊠ High
Physical USB Theft	Low	Critical	Backup USB in separate location	⊠⊠ Medium
Transaction Substitution	Medium	High	User verifies recipient on air-gapped screen	⊠ High
Dependency Supply Chain	Low	High	Pin package versions; verify hashes	⊠ Medium

4.4 Operational Security Recommendations

Best Practices:

1. Generate keys on truly air-gapped PC - Never connected to internet

2. Verify Alpine ISO checksum - Compare against published SHA-256

3. Use dedicated USB drive - Don't reuse for other purposes

4. Create backup USB - Store in separate physical location

5. Verify transaction details - Check recipient address on air-gapped display

6. Keep Python environment isolated - Use virtual environment for CLI

7. Update dependencies carefully - Pin versions, audit changes

User Guide

5.1 Installation

System Requirements:

- Python 3.11 or higher

- Windows 10+, macOS 11+, or Linux (kernel 5.0+)

- 4GB+ USB drive for cold wallet

- Administrator/root privileges (for USB operations)

Step 1: Install Python Dependencies

```
pip install rich questionary solana solders pynacl httpx aiofiles base58
```

Step 2: Clone Repository

```
git clone https://github.com/ChainLabs-Technologies/coldstar.git
cd coldstar
```

Step 3: Verify Installation

```
python main.py
```

5.2 Creating a Cold Wallet USB

Workflow:

1. Launch CLI:

```
python main.py
```

2. Select "Flash Cold Wallet OS to USB"

- CLI will download Alpine Linux rootfs (~50MB)

- Customize filesystem with wallet structure

- Generate bootable ISO

3. Flash ISO to USB:

- Windows: Use Rufus or balenaEtcher

- macOS: Use balenaEtcher or `dd`

- Linux: Use `dd` command:

```
sudo dd if=output/solana-cold-wallet.iso of=/dev/sdX bs=4M sta
tus=progress
sudo sync
```

4. First Boot (Air-Gapped):

- Boot dedicated PC from USB

- Alpine Linux loads automatically

- Keypair generated on first boot

- Public key displayed on screen → WRITE THIS DOWN

5.3 Creating and Signing Transactions

Transaction Flow:

Step 1: Create Unsigned Transaction (Online)

```
python main.py
# Select: "Create Unsigned Transaction"
# Enter recipient address: 5hP8g7...
# Enter amount: 1.5 SOL
# → Unsigned TX saved to file
```

Step 2: Transfer to USB (Manual)

```
# Copy unsigned transaction file to USB /inbox/
cp unsigned_tx.bin /media/usb/inbox/
```

Step 3: Sign Transaction (Air-Gapped)

```
# Boot USB on air-gapped PC
# Run signing script
cd /wallet
./sign_tx.sh /inbox/unsigned_tx.bin

# → Signed TX saved to /outbox/signed_tx.bin
# → Verify recipient address on screen before confirming
```

Step 4: Transfer Back to Host (Manual)

```
# Copy signed transaction from USB to host
cp /media/usb/outbox/signed_tx.bin ~/signed_tx.bin
```

Step 5: Broadcast Transaction (Online)

```
python main.py
# Select: "Broadcast Signed Transaction"
# → TX submitted to Solana network
# → Transaction signature displayed
```

5.4 Checking Wallet Balance

Method 1: Via CLI (Online)

```
python main.py
# Select: "View Wallet Information"
# Enter public key: 7gX...
# → Balance displayed
```

Method 2: Solana Explorer

```
https://explorer.solana.com/address/YOUR_PUBLIC_KEY?cluster=devnet
```

Cryptographic Operations

6.1 Ed25519 Key Generation

Algorithm: Edwards-curve Digital Signature Algorithm (EdDSA)

Process:

1. Entropy Collection: 256 bits from `/dev/urandom`

2. Seed Derivation: SHA-512(entropy) → 512-bit hash

3. Scalar Clamping: Ensure scalar is in valid range

4. Public Key: Scalar multiplication on Curve25519 (Edwards form)

Security Properties:

- Key Size: 256 bits (32 bytes)

- Security Level: 128-bit classical, ~64-bit quantum

- Signature Size: 512 bits (64 bytes)

- Signature Time: ~0.1ms on modern CPU

- Verification Time: ~0.3ms on modern CPU

6.2 Transaction Signing

Signing Process:

1. Message Hash: SHA-512(message)

2. Nonce Generation: Deterministic (RFC 8032)

```
nonce = SHA-512(hash(privkey_suffix) || message)
```

3. R Calculation: R = nonce * G (base point)

4. Challenge: h = SHA-512(R || pubkey || message)

5. Signature: s = (nonce + h * privkey) mod L

Signature Format:

```
[R (32 bytes) || s (32 bytes)] = 64 bytes total
```

Verification (by Solana validators):

```
s * G == R + h * pubkey
```

6.3 Encoding Schemes

Base58 (Public Keys):

- Alphabet:
  `123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz`

- Omits: `0`, `O`, `I`, `l` (visually ambiguous)

- Example: `7gX8jK...` (43-44 characters)

Base64 (Transaction Data):

- Used for JSON-RPC transmission

- More compact than hex

- Standard Base64 alphabet

JSON Array (Keypair Storage):

```
[123, 45, 67, ...]  // 64 bytes
```

- Compatible with Solana CLI format

- Easy to parse in any language

Threat Analysis

7.1 Sophisticated Attack Scenarios

Scenario 1: Compromised Host PC with Transaction Manipulation

Attack:

1. Malware intercepts transaction creation
2. Replaces recipient address with attacker's address
3. User unknowingly copies malicious TX to USB
4. User signs attacker's transaction

Mitigation:

- ⬚ Air-gapped system displays full transaction details before signing
- ⬚ User MUST verify recipient address on offline screen
- ⬚ Signing script shows: From, To, Amount
- ⬚⬚ User Responsibility: Visual verification is critical

Effectiveness: High (requires user vigilance)

Scenario 2: Backdoored Alpine Linux ISO

Attack:

1. Attacker compromises download server or MITM attack
2. Replaces legitimate ISO with backdoored version
3. Backdoor exfiltrates private key during generation
4. Key sent via covert channel (if user later connects USB online)

Mitigation:

- ⬚ SHA-256 checksum verification built into CLI

- ☒ Published checksums on multiple channels (GitHub, website)

- ☒ Reproducible builds (community can rebuild and compare)

- ☒ Offline key generation (no network = no exfiltration during creation)

Effectiveness: Very High

Scenario 3: Evil Maid Attack (Physical USB Tampering)

Attack:

1. Attacker gains temporary physical access to USB

2. Replaces signing script with key-logging version

3. Next signing operation captures private key

4. Attacker retrieves USB later to extract key

Mitigation:

- ☒☒ Read-only filesystem prevents script modification (partial)

- ☒☒ Tamper-evident seals on USB (user must check)

- ☒☒ Boot integrity verification (not implemented in v1.0)

- ☒ No secure element (like hardware wallets have)

Effectiveness: Medium (depends on user vigilance)

7.2 Comparison to Alternatives

vs. Hardware Wallets (Ledger/Trezor)

Aspect	Coldstar	Hardware Wallets
Cost	$5-10 (USB)	$50-200
Transparency	100% open-source	Partial (Trezor) to None (Ledger)
Physical Security	Standard USB	Secure element (Ledger only)

Supply Chain	Reproducible builds	Manufacturer trust
Attack Surface	~50MB OS	~1-2MB firmware
Portability	Requires PC	Standalone device
Setup Complexity	High (bootable USB)	Low (plug-and-play)

Verdict: Coldstar trades convenience for transparency and cost. Hardware wallets are more portable but require manufacturer trust.

vs. Paper Wallets

Aspect	Coldstar	Paper Wallets
Security	Offline signing	Keys printed on paper
Usability	CLI + scripts	Manual key entry
Backup	Multiple USB copies	Photocopy
Signing	Ed25519 (fast)	Requires import to hot wallet
Attack Surface	Alpine Linux	None (paper)

Verdict: Paper wallets are simpler but require importing keys to sign (defeats cold storage). Coldstar allows true offline signing.

SECTION 8

Performance & Scalability

8.1 Performance Metrics

Keypair Generation:

- Time: ~50-100ms
- CPU Usage: Minimal (single-threaded)
- Memory: <10MB

Transaction Signing:

- Time: ~5-10ms (Ed25519 signature)
- CPU Usage: Minimal
- Memory: <5MB

ISO Building:

- Time: ~2-5 minutes (depends on download speed)
- Disk Space: ~150MB temporary, 50MB final ISO
- Network: ~50MB download (Alpine rootfs)

Transaction Broadcasting:

- Time: ~1-2 seconds (RPC network latency)
- Success Rate: >99% on Devnet/Mainnet
- Retry Logic: Exponential backoff on failure

8.2 Scalability Considerations

CLI Tool:

- Single-user application (no concurrency needed)
- Stateless operations (no database)

- Minimal resource requirements

USB Device:

- Supports standard PC hardware (x86_64)

- RAM requirement: 512MB minimum

- Boot time: ~30 seconds on modern hardware

Network:

- Dependent on Solana RPC endpoint performance

- Public endpoints may rate-limit (60 req/min typical)

- Recommendation: Use private RPC for high volume

Conclusion

9.1 Summary

Coldstar v1.0 provides a production-ready, open-source cold wallet solution for Solana that prioritizes security through air-gap isolation. By leveraging industry-standard cryptography (Ed25519), minimal operating systems (Alpine Linux), and transparent Python code, it offers an accessible alternative to proprietary hardware wallets.

Key Achievements:

- ⬚ Complete air-gap separation between key storage and network access
- ⬚ Minimal attack surface (~50MB Alpine Linux)
- ⬚ Full code transparency (100% open-source)
- ⬚ Cost-effective ($5-10 USB drive)
- ⬚ Cross-platform CLI (Windows, macOS, Linux)

Trade-offs:

- ⬚⬚ Higher setup complexity vs. hardware wallets
- ⬚⬚ Requires user vigilance (transaction verification)
- ⬚⬚ Less portable (needs PC for signing)

9.2 Use Cases

Ideal For:

- Long-term HODLers prioritizing security over convenience
- Developers and power users comfortable with CLI tools
- Organizations requiring auditable security solutions
- Users seeking vendor-independent cold storage

- Educational purposes (understanding cold wallet internals)

Not Ideal For:

- Frequent traders (signing overhead too high)

- Non-technical users (complexity barrier)

- Mobile-only users (requires desktop PC)

9.3 Project Status

Current Version: 1.0.0 (Production-Ready)

Supported Features:

- ⊠ Ed25519 keypair generation

- ⊠ Offline transaction signing

- ⊠ Solana Devnet/Mainnet support

- ⊠ USB device detection (Windows/Linux)

- ⊠ Bootable ISO creation

- ⊠ Balance checking

- ⊠ SOL transfer transactions

- ⊠ Devnet airdrops

Known Limitations:

- ⊠ No SPL token support

- ⊠ No multi-signature support

- ⊠ No passphrase encryption (manual GPG required)

- ⊠ No boot integrity verification

- ⊠ Manual file transfer between systems

9.4 Contributing

How to Contribute:

- Report bugs: GitHub Issues

- Submit patches: Pull Requests

- Security vulnerabilities: security@chainlabs.io (responsible disclosure)

- Documentation improvements: Edit `README.md` or this whitepaper

Development Setup:

```
git clone https://github.com/ChainLabs-Technologies/coldstar.git
cd coldstar
python -m venv venv
source venv/bin/activate  # or `venv\Scripts\activate` on Windows
pip install -r local_requirements.txt
python main.py
```

9.5 Final Remarks

Cold storage is not about convenience—it's about security. Coldstar embraces this philosophy by prioritizing air-gap isolation and code transparency over user-friendliness. For users willing to invest time in proper setup and operational procedures, it offers a robust, verifiable, and cost-effective solution for securing Solana assets.

Your keys, your responsibility. Open source, open trust.

A. Technical Specifications

A.1 System Requirements

Development/Host System:

- OS: Windows 10+, macOS 11+, Linux (Ubuntu 20.04+)

- Python: 3.11 or higher

- RAM: 4GB minimum

- Disk Space: 500MB for CLI + temporary ISO build files

- Network: Required for RPC communication and Alpine download

Air-Gapped System (USB Boot):

- Architecture: x86_64 (Intel/AMD)

- RAM: 512MB minimum (1GB recommended)

- USB Port: USB 2.0 or higher

- BIOS: Legacy or UEFI boot support

- Display: Required for transaction verification

USB Drive:

- Capacity: 4GB minimum (8GB recommended for overhead)

- Speed: USB 2.0+ (faster for boot times)

- Format: Will be overwritten during flashing

A.2 Cryptographic Parameters

Parameter	Value	Standard
Signature Algorithm	Ed25519	RFC 8032

Curve	Curve25519 (Edwards form)	-
Hash Function	SHA-512	FIPS 180-4
Private Key Size	256 bits (32 bytes)	-
Public Key Size	256 bits (32 bytes)	-
Signature Size	512 bits (64 bytes)	-
Security Level	128-bit classical	-
Quantum Resistance	~64-bit	NIST estimate
Library	PyNaCl (libsodium)	Audited

A.3 Network Configuration

Solana Networks:

Network	RPC Endpoint	Purpose
Devnet	https://api.devnet.solana.com	Testing, airdrops
Testnet	https://api.testnet.solana.com	Validator testing
Mainnet-Beta	https://api.mainnet-beta.solana.com	Production

Transaction Costs:

- Base fee: 5,000 lamports (0.000005 SOL)
- Signature fee: 5,000 lamports per signature
- Priority fees: Optional (user-configurable)

Blockhash Validity:

- ~150 seconds (150 blocks @ 1 block/sec)
- Transactions must be broadcast within this window

A.4 File Formats

Keypair Storage (`keypair.json`):

```
[
  123, 45, 67, 89, ..., 234  // 64 bytes as decimal array
]
```

Public Key (`pubkey.txt`):

```
7gX8jK9pQvZjR3mN4sT6uV2wA1bC3dE4fG5hH6iJ7kK8
```

Base58-encoded, 43-44 characters

Unsigned Transaction:

Binary format (Solana `Transaction` serialized)

Signed Transaction:

Binary format (Solana `Transaction` with signatures)

B. Glossary

Term	Definition
Air-Gap	Physical isolation of a computer from networks
Alpine Linux	Minimal, security-focused Linux distribution
Base58	Binary-to-text encoding (Bitcoin-style)
Blockhash	Cryptographic hash of recent block (anti-replay)
CLI	Command-Line Interface
Cold Wallet	Cryptocurrency wallet with keys stored offline
Ed25519	Elliptic curve signature scheme (faster than ECDSA)
Evil Maid	Physical tampering attack while device unattended
Lamports	Smallest unit of SOL (1 SOL = 10^9 lamports)
NaCl	"Networking and Cryptography Library" (libsodium)
RPC	Remote Procedure Call (API for blockchain)
Signing	Creating digital signature with private key

C. Security Audit Checklist

For Independent Auditors:

- ☐ Verify SHA-256 checksum of Alpine Linux download

- ☐ Review `src/wallet.py` for key generation logic

- ☐ Inspect `src/transaction.py` for signing implementation

- ☐ Check file permissions on `keypair.json` (should be 0600)

- ☐ Verify network blacklist in ISO build process

- ☐ Test air-gap isolation (no network drivers loaded)

- ☐ Review entropy source (`/dev/urandom`)

- ☐ Check for hardcoded secrets or backdoors

- ☐ Verify Ed25519 implementation (PyNaCl)

- ☐ Test transaction signing correctness

- ☐ Review RPC communication for leaks

- ☐ Check for timing attack vulnerabilities

- ☐ Verify reproducible ISO build process

APPENDIX

D. References

1. EdDSA Specification: Bernstein, D. J., et al. "High-speed high-security signatures." RFC 8032, 2017.

2. Solana Documentation: https://docs.solana.com

3. Alpine Linux Security: https://alpinelinux.org/about/

4. NaCl Cryptography: https://nacl.cr.yp.to/

5. Python Solana SDK: https://github.com/michaelhly/solana-py

6. Solders (Rust): https://github.com/kevinheavey/solders

E. License

MIT License

```
Copyright (c) 2025 ChainLabs Technologies

Permission is hereby granted, free of charge, to any person obtaini
ng a copy
of this software and associated documentation files (the "Software"
), to deal
in the Software without restriction, including without limitation t
he rights
to use, copy, modify, merge, publish, distribute, sublicense, and/o
r sell
copies of the Software, and to permit persons to whom the Software
is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be incl
uded in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXP
RESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABI
LITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT S
HALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OT
HER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARI
SING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALI
NGS IN THE
SOFTWARE.
```

Contact & Support:

- GitHub: https://github.com/ChainLabs-Technologies/coldstar

- Issues: https://github.com/ChainLabs-Technologies/coldstar/issues

- Email: syrem@chainlabs.uno

Document Version: 1.0.0

Last Updated: December 24, 2025

Coldstar Version: 1.0.0