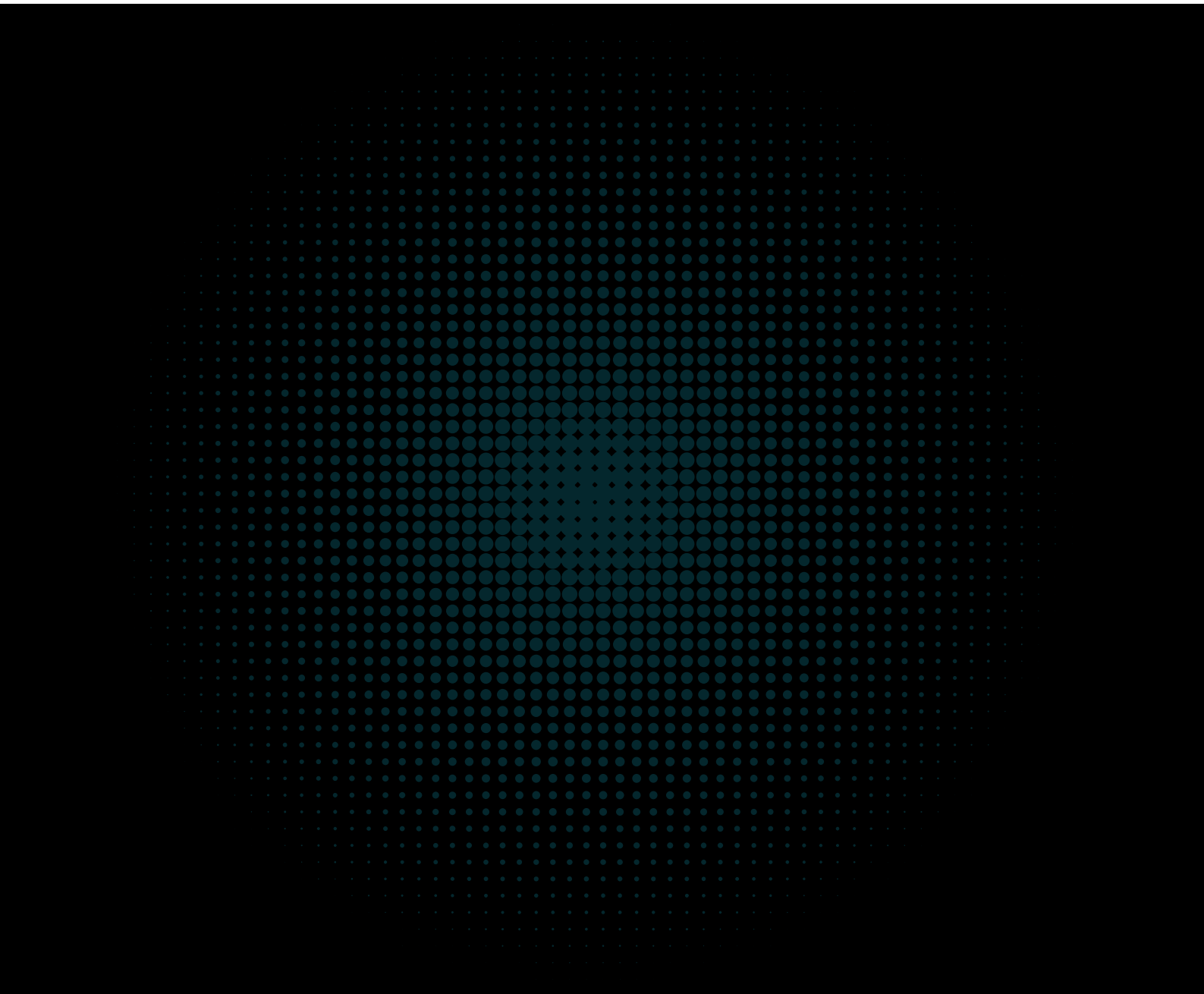
cmta.

CMTAT Confidential Specification

CMTAT Functional Specifications with Zama Confidential Blockchain Protocol and ERC-7984

CMTAT Confidential version: v1.0.0
First published: July 2026



CMTAT Confidential - Confidential Security Token

A confidential security token implementation combining CMTAT compliance features with the Zama Confidential Blockchain Protocol for private balances.

Table of Contents

Overview

CMTAT Confidential implements the ERC-7984 standard (Confidential Fungible Token) with CMTAT regulatory compliance modules. All token balances and transfer amounts are encrypted using Fully Homomorphic Encryption (FHE), ensuring transfer amount and balance privacy while maintaining regulatory compliance capabilities.

CMTAT is a security token framework by Capital Markets and Technology Association that includes various compliance features such as conditional transfer, account freeze, and token pause. The specification are blockchain agnostic with implementation available for several different blockchain ecosystem such as Ethereum, Solana, Tezos, and Aztec (which also enables confidential transactions).

CMTAT Confidential is built on top of OpenZeppelin Confidential Contracts, including its ERC-7984 implementation, and on top of the Solidity CMTAT implementation for compliance modules.

Version 1.0.0 of this project has undergone a security audit by OpenZeppelin, sponsored by Zama. The audit was performed on version v0.3.0, and the fixes were made and reviewed for version v1.0.0. The audit did not cover the underlying CMTAT library — see Security for the report and its scope.

What is FHE?

Fully Homomorphic Encryption (FHE) enables computing directly on encrypted data without ever decrypting it. The Zama Protocol uses FHE combined with Multi-Party Computation (MPC) for threshold decryption and Zero-Knowledge Proofs (ZKPoKs) for input validation, providing:

- **End-to-end encryption**: Transaction inputs and state remain encrypted - no one can see the data, not even node operators
- **Composability**: Confidential contracts can interact with other contracts and dApps
- **Programmable confidentiality**: Smart contracts define who can decrypt what through the Access Control List (ACL)

Key Features

- **Confidential Balances**: All balances are stored as `euint64` (encrypted unsigned 64-bit integers) - only authorized parties can decrypt
- **Confidential Transfers**: Transfer amounts are submitted as encrypted inputs with Zero-Knowledge Proofs of Knowledge (ZKPoKs)
- **Regulatory Compliance**: Pause, freeze, and forced transfer capabilities for compliance
- **Role-Based Access Control**: Granular permissions for minting, burning, pausing, and enforcement
- **Document Management**: Attach terms, documents, and metadata to tokens
- **ERC-7984 Standard**: Based on OpenZeppelin's confidential token implementation
- **ERC-7943 Compliance**: Standard errors (`ERC7943CannotSend`, `ERC7943CannotReceive`, `ERC7943CannotTransfer`) and view checks (`canSend`, `canReceive`, `canTransfer`) in `CMTATConfidentialWhitelist`

Architecture

```
CMTAT-Confidential
├── ERC7984 (OpenZeppelin Confidential Contracts)
│   ├── Encrypted balances (euint64)
│   ├── Confidential transfers
│   ├── Operator system
│   └── Disclosure mechanism
│
├── CMTATBaseGeneric (CMTAT Modules)
│   ├── PauseModule - Pause/unpause transfers
│   ├── EnforcementModule - Freeze/unfreeze addresses
│   ├── AccessControlModule - Role-based permissions
│   ├── DocumentEngineModule - ERC-1643 document management
│   └── ExtraInformationModule - Token metadata (tokenId, terms, info)
│
├── FHE Modules (custom extensions)
│   ├── ERC7984MintModule - Modular mint with authorization hook
│   ├── ERC7984BurnModule - Modular burn with authorization hook
│   ├── ERC7984EnforcementModule - Forced transfer and forced burn
│   ├── ERC7984BalanceViewModule - Per-account balance observers (holder + role
slots)
│   ├── ERC7984PublishTotalSupplyModule - Public total supply disclosure (all
variants)
│   ├── ERC7984TotalSupplyViewModule - Total supply observer list with auto ACL
re-grant (all full variants except Lite)
│   ├── ERC7984TokenAttributeModule - Post-deployment name/symbol updates (ERC-
3643 alignment)
│   └── ERC7984RuleEngineModule - RuleEngine transfer restrictions with public
value = 0
```

```
│
└── Zama Protocol Infrastructure (configured via ZamaEthereumConfig)
    ├── ACL - Access Control List for encrypted data permissions
    ├── FHEVMExecutor (Coprocessor) - Performs FHE computations
    ├── KMSVerifier - Verifies decryption proofs from Key Management System
    └── InputVerifier - Validates encrypted inputs and ZKPoKs
```

Module Reference

All FHE modules follow the same pattern: a role constant, a modifier, a virtual authorization hook overridden in `CMTATConfidentialBase`, and optional validation/hook functions. Every module has a corresponding interface in `contracts/interfaces/`.

Module	Role	Source	Availability	Purpose
`ERC7984MintModule`	`MINTER_ROLE`	`contracts/modules/`	all variants	Encrypted mint via ZKPoK input or existing handle
`ERC7984BurnModule`	`BURNER_ROLE`	`contracts/modules/`	all variants	Encrypted burn via ZKPoK input or existing handle
`ERC7984EnforcementModule`	`FORCED_OPS_ROLE`	`contracts/modules/`	all variants	Forced transfer and forced burn from frozen addresses
`ERC7984BalanceViewModule`	`OBSERVER_ROLE`	`contracts/modules/`	all variants	Per-account balance observers (holder slot + role slot); auto ACL re-grant on every `_update`
`ERC7984PublishTotalSupplyModule`	`SUPPLY_PUBLISHER_ROLE`	`contracts/modules/`	all variants	Mark the current total supply handle as publicly decryptable (one-shot, irrevocable per handle)
`ERC7984TotalSupplyViewModule`	`SUPPLY_OBSERVER_ROLE`	`contracts/modules/`	all except Lite	Registered observers automatically receive ACL access on the total supply handle after every mint/burn
`ERC7984TokenAttributeModule`	`TOKEN_ATTRIBUTE_ROLE`	`contracts/modules/`	all variants	Post-deployment `setName` / `setSymbol` — ERC-3643 alignment
`ERC7984RuleEngineModule`	`RULE_ENGINE_ROLE`	`contracts/modules/`	RuleEngine variant only	Plug in a CMTA `IRuleEngine` for transfer policy checks; passes `value = 0` because amounts are encrypted
`CMTATConfidentialVersionModule`	—	`contracts/modules/`	all variants	Pins `version()` to `1.0.0`, overriding CMTAT's own version module

CMTAT modules (from `lib/CMTAT/`) are inherited through `CMTATBaseGeneric` and always present in all variants:

CMTAT Module	Role	Purpose
`PauseModule`	`PAUSER_ROLE`	Pause / unpause all transfers
`EnforcementModule`	`ENFORCER_ROLE`	Freeze / unfreeze individual addresses
`AccessControlModule`	`DEFAULT_ADMIN_ROLE`	Role management and contract deactivation
`ValidationModule`	—	`canSend`, `canReceive`, `canTransfer` view checks; ERC-7943 errors
`ExtraInformationModule`	`EXTRA_INFORMATION_ROLE`	`setTokenId`, `setTerms`, `setInformation`
`DocumentERC1643Module`	`DOCUMENT_ROLE`	ERC-1643 document management (`setDocument`, `removeDocument`, `getDocument`)
`AllowlistModule`	`ALLOWLIST_ROLE`	On/off allowlist — `CMTATConfidentialWhitelist` only
`ValidationModuleRuleEngineInternal`	—	`ruleEngine()` storage + `RuleEngine` event — `CMTATConfidentialRuleEngine` only

How It Works

1. **Symbolic Execution**: When a contract calls an FHE operation, the host chain produces a pointer to the result and emits an event to notify the coprocessor network
2. **Coprocessor Computation**: The coprocessors perform the actual FHE computation off-chain
3. **Threshold Decryption**: Decryption requests go through the Key Management Service (KMS), which uses MPC to ensure no single party can access the private key

Summary

This section maps the CMTAT framework features to the CMTAT Confidential implementation, showing how standard functionalities are adapted for Fully Homomorphic Encryption.

CMTAT Framework Mapping

CMTAT Framework Mandatory Functionalities	CMTATConfidential Corresponding Features
Know total supply	`confidentialTotalSupply()` returns `euint64` (encrypted)
Know balance	`confidentialBalanceOf()` returns `euint64` (encrypted)
Transfer tokens	`confidentialTransfer()` with encrypted amount + ZKPoK

CMTAT Framework Mandatory Functionalities	CMTATConfidential Corresponding Features
Create tokens (mint)	`mint()` with encrypted amount + ZKPoK
Cancel tokens (burn)	`burn()` with encrypted amount + ZKPoK
Pause tokens	`pause()` - inherited from CMTAT
Unpause tokens	`unpause()` - inherited from CMTAT
Deactivate contract	`deactivateContract()` - inherited from CMTAT
Freeze	`setAddressFrozen(address, true)` - inherited from CMTAT
Unfreeze	`setAddressFrozen(address, false)` - inherited from CMTAT
Name attribute	`name()` — mutable post-deployment via `setName()` (`TOKEN_ATTRIBUTE_ROLE`)
Ticker symbol attribute	`symbol()` — mutable post-deployment via `setSymbol()` (`TOKEN_ATTRIBUTE_ROLE`)
Token ID attribute	`tokenId()` — mutable via `setTokenId()` (`EXTRA_INFORMATION_ROLE`)
Reference to legally required documentation	`terms()` — mutable via `setTerms()` (`EXTRA_INFORMATION_ROLE`)

Extended Features

Functionalities	CMTAT Confidential Features	Available
Forced Transfer	`forcedTransfer()` with encrypted amount	✔
Forced Burn	`forcedBurn()` with encrypted amount	✔
Operator System	`setOperator()` / `confidentialTransferFrom()`	✔
Public Disclosure	`requestDiscloseEncryptedAmount()` / `discloseEncryptedAmount()`	✔
On-chain snapshot	Not implemented	✘
Freeze partial tokens	Not implemented (all balances are encrypted)	✘
Integrated allowlisting	`CMTATConfidentialWhitelist` (ERC-7943 `canSend` / `canReceive` / `canTransfer`)	✔
RuleEngine / transfer hook	`CMTATConfidentialRuleEngine` (`value = 0` because amounts are encrypted)	✔

Functionalities	CMTAT Confidential Features	Available
Upgradability	Not implemented (standalone only)	✘

Implementation Details

Functionalities	CMTAT Confidential	Note
Mint while paused	✔	Minting is allowed when contract is paused (same as CMTAT)
Burn while paused	✔	Burning is allowed when contract is paused (same as CMTAT)
Self burn	✘	Only `BURNER_ROLE` can burn tokens
Standard burn on frozen address	✘	Use `forcedBurn()`
Forced burn from frozen address	✔	`forcedBurn()` with `FORCED_OPS_ROLE`
Burn via `forcedTransfer`	✘	`forcedTransfer` reverts if `to` is `address(0)` -- use `forcedBurn()`
Balance overflow protection	✔	Uses FHESafeMath: transfers 0 on overflow/underflow (privacy-preserving)

Key Differences from Standard CMTAT

Aspect	CMTAT (Standard)	CMTAT Confidential (Confidential)
Balance type	`uint256` (public)	`euint64` (encrypted)
Value range	Up to ~1.15×10^{77} (`uint256`)	Up to ~1.84×10^{19} (`uint64` max = 18,446,744,073,709,551,615)
Transfer amount	`uint256` (public)	`externalEuint64` + ZKPoK
Total supply	`uint256` (public)	`euint64` (encrypted)
Balance visibility	Anyone can read	Only ACL-authorized parties can decrypt
Transfer validation	Reverts on insufficient balance	Transfers 0 silently (privacy-preserving)
Allowance system	ERC20 `approve` / `allowance`	Operator system with time-limited access

Aspect	CMTAT (Standard)	CMTAT Confidential (Confidential)
Forced Burn	Through `forcedTransfer` or `forcedBurn` if implemented	Through `forcedBurn` since the function is implemented

> **Important:** The `euint64` type has a significantly smaller range than `uint256`. Decimals above 18 are rejected at construction (`CMTAT_DecimalsTooHigh`). See Choosing Decimals for the full supply impact table.

Decryption Requirements

To decrypt encrypted values (balances, amounts, total supply), the requesting party must:

1. Have ACL permission granted via `FHE.allow()` or `FHE.allowTransient()` (verifiable with `FHE.isAllowed()` or `FHE.isSenderAllowed()`)
2. Or the value must be marked publicly decryptable via `FHE.makePubliclyDecryptable()`
3. Request decryption through the Zama Relayer SDK (`@zama-fhe/relayer-sdk`)
4. Submit the decryption proof on-chain via `FHE.checkSignatures()` (reverts if the proof is invalid)

Deployment Variants

Four deployment-ready contracts are provided. They share the same abstract base (`CMTATConfidentialBase`) except for optional total supply visibility, optional RuleEngine transfer restrictions, and optional allowlist enforcement.

	`CMTATConfidential`	`CMTATConfidentialLite`	`CMTATConfidentialRuleEngine`	`CMTATConfidentialWhitelist`
Confidential balances & transfers	✔	✔	✔	✔
Mint / Burn / Forced ops	✔	✔	✔	✔
Pause / Freeze	✔	✔	✔	✔
Per-account balance observers	✔	✔	✔	✔
`publishTotalSupply` (public disclosure)	✔	✔	✔	✔
Total supply observer list (auto ACL)	✔	✘	✔	✔
RuleEngine transfer restriction	✘	✘	✔	✘
Allowlist enforcement	✘	✘	✘	✔
`canSend` / `canReceive` / `canTransfer` (partial ERC-7943)	✔	✔	✔	✔
ERC-7943 `0x3edbb4c4` (full compliance)	✘	✘	✘	✘
`SUPPLY_OBSERVER_ROLE`	✔	✘	✔	✔
`SUPPLY_PUBLISHER_ROLE`	✔	✔	✔	✔
`RULE_ENGINE_ROLE`	✘	✘	✔	✘
`ALLOWLIST_ROLE`	✘	✘	✘	✔
Contract size	~21.1 KB	~19.7 KB	~22.2 KB	~22.2 KB

Choose `CMTATConfidentialLite` when automatic per-observer ACL re-grant on every mint/burn is not required and you want to minimize deployment cost. `publishTotalSupply` (one-shot public disclosure) is available in all deployment variants.
Choose `CMTATConfidentialRuleEngine` when public transfer-policy rules such as whitelists, blacklists, jurisdiction checks, or other CMTA RuleEngine rules must restrict confidential transfers. Since amounts are encrypted, the token passes `0` as the RuleEngine `value` for validation and

transfer notifications.
Choose `CMTATConfidentialWhitelist` when a simple on/off allowlist is sufficient: both sender and recipient must be allowlisted when enforcement is enabled. Provides `canSend`, `canReceive`, `canTransfer` view checks (partial ERC-7943 — see note in Whitelist Variant section).

Installation

```
# Clone the repository
git clone --recursive https://github.com/your-repo/CMTAT-Confidential.git
cd CMTAT-Confidential

# Install dependencies
npm install

# Compile contracts
npm run compile

# Run tests
npm run test
```

Compiler & EVM Version

- Solidity pragmas use `^0.8.27` across the codebase to stay compatible with OpenZeppelin Confidential Contracts.
- Hardhat is configured to compile with `0.8.34`.
- The configured EVM version is `prague` (see `hardhat.config.ts`).

Versioning

The contract-level `version()` string is pinned to `1.0.0` via `CMTATConfidentialVersionModule`.

Security

Security Audit — OpenZeppelin

Version 1.0.0 of this project has undergone a security audit by OpenZeppelin, sponsored by Zama.

The audit was performed on version **v0.3.0** (commit `463087c`) between 2026-06-12 and 2026-06-24. All fixes were made and reviewed for version **v1.0.0**.

📄 **Final report:** `OpenZeppelin_Audit_Reportv1.0.0.pdf` (July 9, 2026) — remediation details in `OpenZeppelin-Remediation.md`.

8 findings — 0 critical, 0 high, **1 medium**, **2 low**, **5 notes & additional information**. All findings are resolved or accepted with documented rationale.

Audit scope

The audit covered **only the contracts in this repository** (`contracts/deployment/*.sol`, `contracts/modules/*.sol`, and `contracts/CMTATConfidentialBase.sol`). It did **not** cover the underlying dependencies, which were assumed to behave as specified:

- the CMTAT library (compliance modules) — `lib/CMTAT`
- the CMTA RuleEngine — `lib/RuleEngine`

- OpenZeppelin Confidential Contracts (ERC-7984) — `lib/openzeppelin-confidential-contracts`
- the Zama FHEVM protocol and `@fhevm/solidity` library

> ⚠ **Warning — the underlying CMTAT library was not audited.**
> At the time of the audit, this project pinned CMTAT `v3.3.0-rc1` (a release candidate), which had not itself undergone a security audit. The OpenZeppelin audit therefore provides **no assurance about the CMTAT compliance modules** (pause, freeze, allowlist, documents, access control) that CMTAT Confidential inherits.
>
> Before deploying this project in production, consider:
>
> - performing a security audit of the CMTAT library version you depend on;
> - upgrading `lib/CMTAT` to the latest released version, and re-running the test suite and static analysis afterwards;
> - applying the same scrutiny to `lib/RuleEngine` if you deploy the `CMTATConfidentialRuleEngine` variant.

ID	Severity	Finding	Disposition
M-01	Medium	Rule engine not applied to mint and burn, leaving issuance and redemption unscreened	Fixed — rule engine applied to mint/burn; forced operations intentionally still bypass it
L-01	Low	Sequential total-supply disclosures leak individual mint and burn amounts	Accepted as residual risk — mitigated by NatSpec and operator guidance (see Total Supply Visibility)
L-02	Low	`ERC7984TokenAttributeModule` seeds attributes via a skippable internal initializer	Fixed — replaced `_initTokenAttributes` with a constructor
N-01	Note	Missing docstrings	Fixed
N-02	Note	Incomplete docstrings	Fixed
N-03	Note	Floating pragma	Won't fix (by design) — `^0.8.27` is deliberate so library consumers can pick their own `0.8.x` compiler
N-04	Note	Prefix increment operator `++i` can save gas in loops	Fixed
N-05	Note	Misleading documentation	Fixed

> From the report's conclusion: *"the token's security depends as much on how confidentiality and compliance are wired together and operated as on the individual contracts."*

Automated Audit — Nethermind AuditAgent

Nethermind AuditAgent automated scan (March 18, 2026, commit `51f9d7aa`) reported **3 medium**, **2 low**, **2 info**, and **1 best practice** findings across 10 contracts (1 239 lines of code). Full rationale and fix details in `nethermind-audit-agent-report-feedback.md`.

> This report was generated entirely by AI and has not been manually reviewed by Nethermind's security team.

#	Severity	Finding	Disposition	Commit
1	Medium	Receiver reentrancy can bypass the `transferAndCall` rollback path	Disputed — upstream ERC-7984 design; NatSpec warning added	`36dbd3f`
2	Medium	Receiver reentrancy can steal tokens via `confidentialTransferAndCall`	Duplicate of #1 — resolved together	`36dbd3f`
3	Medium	Freezing `address(0)` gives `ENFORCER_ROLE` a global block over holder transfers	Documented — upstream fix proposed in CMTA/CMTAT#372; operator warning added	`1abe564`
4	Low	Unbounded supply-observer ACL refresh can cause OOG on mint/burn	Fixed — admin-controlled cap (`setMaxSupplyObservers`, default 10)	`12249c1`
5	Low	`forcedBurn` does not invoke `_afterBurn`, causing observer ACL staleness	Fixed — `_afterBurn` hook added to `ERC7984EnforcementModule`	`681ebde`
6	Info	`forcedBurn` does not refresh total-supply observer ACLs (full variant)	Duplicate of #5 — resolved together	`681ebde`
7	Info	Unbounded observer list can cause DoS on `mint` and `burn`	Duplicate of #4 — resolved together	`12249c1`
8	Best Practice	Duplicate observer removal via `setRoleObserver(account, address(0))`	Fixed — `setRoleObserver` now rejects `address(0)`	`a74314e`

Static Analysis — Aderyn (v1.0.0)

Aderyn static analysis (v1.0.0, Aderyn 0.6.5) reported **0 high** and **7 low** severity findings across 21 contracts (1 292 nSLOC). All findings are accepted or not applicable — unchanged in count and disposition from v0.3.0. Full rationale in `aderyn-report-feedback.md`, source report in `aderyn-report.md`. See also `doc/audit/AUDIT_OVERVIEW.md`.

Command used to generate the report (mocks excluded):

```
aderyn -x mocks --output doc/audit/v1.0.0/aderyn-report.md
```

ID	Finding	Instances	Disposition

ID	Finding	Instances	Disposition
L-1	Centralization Risk	14	Accepted — role-based access control is mandatory for a regulated security token
L-2	Unspecific Solidity Pragma (`^0.8.27`)	21	Accepted — lower bound required by OZ Confidential submodule; kept floating for library consumers (OZ finding N-03); toolchain compiles with `0.8.34`
L-3	PUSH0 Opcode	21	Not applicable — target is Ethereum mainnet, EVM version set to `prague`
L-4	Modifier Invoked Only Once	3	Accepted — consistent with the module authorization pattern across all modules
L-5	Empty Block	22	Accepted — modifier-only authorization hooks and intentional virtual extension points
L-6	Internal Function Used Only Once	1	Accepted — required by the OpenZeppelin `initializer` modifier pattern
L-7	Unchecked Return	8	Not applicable — `FHE.allow()` / `FHE.makePubliclyDecryptable()` return the same handle (fluent interface), not an error code

Static Analysis — Slither (v1.0.0)

Slither static analysis (v1.0.0, Slither 0.11.5, compiled via Foundry / solc 0.8.34) reported **0 high**, **8 medium**, **3 low**, and **7 informational** findings. None is exploitable — the Medium/Low results are the same FHE fluent-interface pattern Aderyn reports as L-7. Full rationale in `slither-report-feedback.md`, source report in `slither-report.md`.

Command used to generate the report (mocks excluded):

```
slither . --checklist --filter-paths "node_modules,lib,test,forge-std,mocks"
```

Detector	Severity	Instances	Disposition
unused-return	Medium	8	Not applicable — `FHE.allow` / `FHE.makePubliclyDecryptable` fluent API (return is the same handle)
reentrancy-events	Low	3	False positive — FHE coprocessor call followed only by an event; no exploitable state
dead-code	Informational	5	False positive — virtual hooks (`_validateMint`/`_validateBurn`/`_afterBurn`/`_validateForced*`) dispatched via inheritance override
naming-convention	Informational	1	Cosmetic — `_TOKEN_DECIMALS` immutable styled like a constant

Detector	Severity	Instances	Disposition
unindexed-event-address	Informational	1	Cosmetic — optional indexing on the rare `MaxSupplyObserversUpdated` admin event

Static Analysis — Aderyn (v0.3.0)

Aderyn static analysis (v0.3.0) reported **0 high** and **7 low** severity findings across 21 contracts (1 276 nSLOC). All findings are accepted or not applicable. Full rationale in `aderyn-report-feedback.md`.

ID	Finding	Instances	Disposition
L-1	Centralization Risk	14	Accepted — role-based access control is mandatory for a regulated security token
L-2	Unspecific Solidity Pragma (`^0.8.27`)	21	Accepted — lower bound required by OZ Confidential submodule; Hardhat compiles with `0.8.34`
L-3	PUSH0 Opcode	21	Not applicable — target is Ethereum mainnet, EVM version set to `prague` in `hardhat.config.ts`
L-4	Modifier Invoked Only Once	3	Accepted — consistent with the module authorization pattern across all modules
L-5	Empty Block	22	Accepted — modifier-only authorization hooks and intentional virtual extension points
L-6	Internal Function Used Only Once	1	Accepted — required by the OpenZeppelin `initializer` modifier pattern
L-7	Unchecked Return	8	Not applicable — `FHE.allow()` / `FHE.makePubliclyDecryptable()` return the same handle (fluent interface), not an error code

Static Analysis — Aderyn (v0.2.0)

Aderyn static analysis (v0.2.0) reported **0 high** and **8 low** severity findings across 12 contracts (663 nSLOC). All findings are accepted or not applicable for this codebase. Full rationale in `aderyn-report-feedback.md`, source report in `aderyn-report.md`.

Command used to generate the report:

```
aderyn --output aderyn-report.md
```

ID	Finding	Instances	Disposition

ID	Finding	Instances	Disposition
L-1	Centralization Risk	11	Accepted — role-based access control is mandatory for a regulated security token
L-2	Unspecific Solidity Pragma (`^0.8.27`)	12	Accepted — lower bound required by OZ Confidential submodule; Hardhat compiles with `0.8.34`
L-3	PUSH0 Opcode	12	Not applicable — target is Ethereum mainnet, EVM version set to `prague` in `hardhat.config.ts`
L-4	Modifier Invoked Only Once	2	Accepted — consistent with the module authorization pattern across all modules
L-5	Empty Block	18	Accepted — modifier-only authorization hooks and intentional virtual extension points
L-6	Internal Function Used Only Once	1	Accepted — required by the OpenZeppelin `initializer` modifier pattern
L-7	State Change Without Event	1	Not applicable — occurs in a mock contract (`ConfidentialReceiverMock`), not production code
L-8	Unchecked Return	12	Not applicable — `FHE.allow()` / `FHE.makePubliclyDecryptable()` return the same handle (fluent interface), not an error code

Static Analysis — Slither

We attempted to run Slither with:

```
slither . --checklist --filter-paths "openzeppelin-contracts|test|forge-std" > slither-report.md
```

But the run failed with:

```
ERROR:root:Error:
ERROR:root:The source code appears to be out of sync with the build artifacts on disk.
        This discrepancy can occur after recent modifications to node_modules/@fhevm/solidity/config/ZamaConfig.sol. To resolve this
        issue, consider executing the clean command of the build system (e.g. forge clean).
ERROR:root:Please report an issue to https://github.com/crytic/slither/issues
```

At this stage, no Slither findings are available for this release due to this tooling/build-artifact sync issue.

Design Limitations

`confidentialTransferAndCall` — silent refund failure on re-entrant receiver

`confidentialTransferAndCall` and `confidentialTransferFromAndCall` use an ERC-1363-style
callback pattern where the receiver is credited **before** its `onConfidentialTransferReceived` hook is called. If the callback returns `false`, the implementation attempts a compensating reverse transfer via `FHE.select(success, 0, sent)`.

This reverse transfer can silently produce **0** if a malicious or re-entrant receiver drains its encrypted balance inside the callback before returning `false`: because `FHESafeMath.tryDecrease`
operates on an encrypted value, it cannot revert on underflow — it saturates to 0 instead. The sender then loses the transferred amount permanently with no on-chain indication of failure.

This is a structural limitation of FHE arithmetic and an intentional trade-off in the upstream ERC-7984 library (see audit finding #1 and #2 in the table above).

**Only call `confidentialTransferAndCall` and `confidentialTransferFromAndCall` with trusted,
audited receiver contracts.**

Balance observers hold indivisible read + disclosure power

Granting a balance observer (via `setObserver` by the holder, or `setRoleObserver` by `OBSERVER_ROLE`) gives that
observer ACL access to the account's encrypted balance handle. In the FHEVM there is **no read-only ACL**: any
address allowed on a handle can, by calling the ACL system contract (`ACL.sol`) directly — with no way for this
contract to intercept it — not only read the value privately (`userDecrypt`) but also `allow()` it to any third
party or `allowForDecryption()` it to make the balance **publicly and irreversibly decryptable by anyone**. Read
access and disclosure access are the **same grant**; gating `requestDiscloseEncryptedAmount` at the token layer
would not prevent this and is intentionally not attempted.

Consequences:

- **Treat assigning any observer as granting full disclosure power** over the observed account's balance. Only
  make an observer of a party trusted with the plaintext *and* with the right to reveal or re-share it.
- **Observers must only read the value privately** (`userDecrypt` for their own key). They must never invoke any
  disclosure or re-grant primitive (`makePubliclyDecryptable` / `allowForDecryption` / `allow`) on a third
  party's handle — doing so leaks that account's balance permanently and cannot be undone.

- **Govern `OBSERVER_ROLE` tightly** (e.g. multisig/timelock): `setRoleObserver` can assign an observer to an
  account **without the account's consent**, so it can unilaterally place a party in a position to disclose that
  account's balance.

See finding **F-1** (`F-1.md`) for the full analysis.

Roles

Role	Description
`DEFAULT_ADMIN_ROLE`	Can grant/revoke all roles, deactivate contract, set total supply observer cap (`setMaxSupplyObservers`)
`MINTER_ROLE`	Can mint new tokens
`BURNER_ROLE`	Can burn tokens
`PAUSER_ROLE`	Can pause/unpause all transfers
`ENFORCER_ROLE`	Can freeze and unfreeze addresses — **must not freeze `address(0)`** (see warning below)
`FORCED_OPS_ROLE`	Can execute forced transfers and forced burns on frozen addresses
`OBSERVER_ROLE`	Can assign per-account balance observers via `setRoleObserver`
`SUPPLY_OBSERVER_ROLE`	Can manage total supply observers (`addTotalSupplyObserver`, `removeTotalSupplyObserver`)
`SUPPLY_PUBLISHER_ROLE`	Can call `publishTotalSupply`
`RULE_ENGINE_ROLE`	Can update the RuleEngine in `CMTATConfidentialRuleEngine`
`ALLOWLIST_ROLE`	Can manage the allowlist in `CMTATConfidentialWhitelist` (`setAddressAllowlist`, `enableAllowlist`)
`TOKEN_ATTRIBUTE_ROLE`	Can rename the token post-deployment (`setName`, `setSymbol`)
`EXTRA_INFORMATION_ROLE`	Can update token metadata (`setTokenId`, `setTerms`, `setInformation`)
`DOCUMENT_ROLE`	Can manage ERC-1643 documents (`setDocument`, `removeDocument`)

Events

All events emitted by the contract, organized by module. Events from FHE modules carry `euint64` handles for encrypted amounts — the plaintext values are not visible in logs.

FHE Modules

ERC7984MintModule

Event	Signature	Emitted when
`Mint`	`Mint(address indexed minter, address indexed to, euint64 encryptedAmount)`	`mint()` completes successfully

ERC7984BurnModule

Event	Signature	Emitted when
`Burn`	`Burn(address indexed burner, address indexed from, euint64 encryptedAmount)`	`burn()` completes successfully

ERC7984EnforcementModule

Event	Signature	Emitted when
`ForcedTransfer`	`ForcedTransfer(address indexed enforcer, address indexed from, address indexed to, euint64 encryptedAmount)`	`forcedTransfer()` completes successfully
`ForcedBurn`	`ForcedBurn(address indexed enforcer, address indexed from, euint64 encryptedAmount)`	`forcedBurn()` completes successfully

ERC7984BalanceViewModule

Event	Signature	Emitted when
`RoleObserverSet`	`RoleObserverSet(address indexed account, address indexed oldObserver, address indexed newObserver, address setBy)`	`setRoleObserver()` or `removeRoleObserver()` is called
`ERC7984ObserverAccessObserverSet`	`ERC7984ObserverAccessObserverSet(address account, address oldObserver, address newObserver)`	`setObserver()` is called by a holder to assign their personal observer (inherited from `ERC7984ObserverAccess`)

ERC7984PublishTotalSupplyModule

Event	Signature	Emitted when
`TotalSupplyPublished`	`TotalSupplyPublished(address indexed publishedBy)`	`publishTotalSupply()` marks the current total supply handle as publicly decryptable

ERC7984TotalSupplyViewModule — `CMTATConfidential`, `CMTATConfidentialRuleEngine`, `CMTATConfidentialWhitelist` only

Event	Signature	Emitted when
`TotalSupplyObserverAdded`	`TotalSupplyObserverAdded(address indexed observer, address indexed addedBy)`	`addTotalSupplyObserver()` registers a new observer
`TotalSupplyObserverRemoved`	`TotalSupplyObserverRemoved(address indexed observer, address indexed removedBy)`	`removeTotalSupplyObserver()` deregisters an observer
`MaxSupplyObserversUpdated`	`MaxSupplyObserversUpdated(uint256 oldMax, uint256 newMax, address updatedBy)`	`setMaxSupplyObservers()` changes the observer cap

ERC7984TokenAttributeModule — all variants

Event	Signature	Emitted when
`Name`	`Name(string indexed newNameIndexed, string newName)`	`setName()` updates the token name
`Symbol`	`Symbol(string indexed newSymbolIndexed, string newSymbol)`	`setSymbol()` updates the token symbol

ERC7984RuleEngineModule — `CMTATConfidentialRuleEngine` only

Event	Signature	Emitted when
`RuleEngine`	`RuleEngine(IRuleEngine indexed newRuleEngine)`	`setRuleEngine()` updates the active rule engine (inherited from `ValidationModuleRuleEngineInternal`)

CMTAT-Inherited Events

PauseModule (via OpenZeppelin `Pausable`)

Event	Signature	Emitted when
`Paused`	`Paused(address account)`	`pause()` is called
`Unpaused`	`Unpaused(address account)`	`unpause()` is called
`Deactivated`	`Deactivated(address indexed account)`	`deactivateContract()` permanently deactivates the contract

EnforcementModule

Event	Signature	Emitted when
`AddressFrozen`	`AddressFrozen(address indexed account, bool indexed isFrozen, address indexed enforcer, bytes data)`	`setAddressFrozen()` changes the freeze state of an address

AccessControlModule (via OpenZeppelin `AccessControl`)

Event	Signature	Emitted when
`RoleGranted`	`RoleGranted(bytes32 indexed role, address indexed account, address indexed sender)`	`grantRole()` assigns a role to an account
`RoleRevoked`	`RoleRevoked(bytes32 indexed role, address indexed account, address indexed sender)`	`revokeRole()` or `renounceRole()` removes a role
`RoleAdminChanged`	`RoleAdminChanged(bytes32 indexed role, bytes32 indexed previousAdminRole, bytes32 indexed newAdminRole)`	`setRoleAdmin()` changes the admin role for a role

AllowlistModule — `CMTATConfidentialWhitelist` only

Event	Signature	Emitted when
`AllowlistEnableStatus`	`AllowlistEnableStatus(address indexed operator, bool status)`	`enableAllowlist()` enables or disables allowlist enforcement
`AddressAddedToAllowlist`	`AddressAddedToAllowlist(address indexed account, bool indexed status, address indexed enforcer, bytes data)`	`setAddressAllowlist()` adds or removes an address from the allowlist

Contract Functions

Minting

Mint tokens to an address (requires `MINTER_ROLE`):

```
// With encrypted input and proof
function mint(
    address to,
    externalEuint64 encryptedAmount,
    bytes calldata inputProof
) public onlyRole(MINTER_ROLE) returns (euint64 transferred);

// With existing encrypted handle (caller must have ACL access)
function mint(
    address to,
    euint64 amount
) public onlyRole(MINTER_ROLE) returns (euint64 transferred);
```

Burning

Burn tokens from an address (requires `BURNER_ROLE`):

```
function burn(
    address from,
    externalEuint64 encryptedAmount,
    bytes calldata inputProof
) public onlyRole(BURNER_ROLE) returns (euint64 transferred);
```

Transfers

ERC-7984 exposes eight transfer function variants:

Function	Description
`confidentialTransfer(to, amount)`	Transfer using existing handle
`confidentialTransfer(to, amount, proof)`	Transfer with encrypted input
`confidentialTransferFrom(from, to, amount)`	Operator transfer using handle
`confidentialTransferFrom(from, to, amount, proof)`	Operator transfer with proof
`confidentialTransferAndCall(...)`	Transfer with ERC-1363 callback
`confidentialTransferFromAndCall(...)`	Operator transfer with callback

RuleEngine Variant

`CMTATConfidentialRuleEngine` adds CMTA RuleEngine checks to holder and operator transfers **as well as mint and burn**. It exposes:

```
function ruleEngine() public view returns (IRuleEngine);
function setRuleEngine(IRuleEngine newRuleEngine) public
onlyRole(RULE_ENGINE_ROLE);
function canTransfer(address from, address to, uint256 amount) public view
returns (bool);
function canTransferFrom(address spender, address from, address to, uint256
amount) public view returns (bool);
```

The public `amount` parameter is intentionally ignored. Confidential balances use encrypted `euint64` amounts, so RuleEngine calls receive `value = 0`:

- holder transfer validation: `ruleEngine.canTransfer(from, to, 0)`
- operator transfer validation: `ruleEngine.canTransferFrom(spender, from, to, 0)`
- holder transfer notification: `ruleEngine.transferred(from, to, 0)`
- operator transfer notification: `ruleEngine.transferred(spender, from, to, 0)`

- mint validation + notification: `ruleEngine.canTransferFrom(operator, address(0), to, 0)` / `ruleEngine.transferred(operator, address(0), to, 0)`
- burn validation + notification: `ruleEngine.canTransferFrom(operator, from, address(0), 0)` / `ruleEngine.transferred(operator, from, address(0), 0)`

Mint and burn are screened at the same chokepoint as standard CMTAT, so issuance to — or redemption from — a non-permitted address is rejected by the configured engine. Following CMTAT v3.3.0, the **operator** (`_msgSender()`, the `MINTER_ROLE` / `BURNER_ROLE` holder) is forwarded as the `spender` on mint/burn, exactly as CMTAT's `_mintOverride` / `_burnOverride` do; rules must exempt the spender on those legs (production `RuleWhitelist` does). **Forced operations (`forcedTransfer`, `forcedBurn`) intentionally bypass the RuleEngine** and remain governed only by the freeze precondition.

Example transfer:

```
import { fhevm } from 'hardhat';

// Create encrypted input
const encryptedInput = await fhevm
  .createEncryptedInput(tokenAddress, senderAddress)
  .add64(1000) // Amount to transfer
  .encrypt();

// Execute transfer
await token.connect(sender)['confidentialTransfer(address,bytes32,bytes)'](
  recipientAddress,
  encryptedInput.handles[0],
  encryptedInput.inputProof
);
```

Whitelist Variant

`CMTATConfidentialWhitelist` adds on/off allowlist enforcement to all holder and operator transfers. It partially implements ERC-7943 view functions but **does not claim full `IERC7943Fungible` compliance** (`0x3edbb4c4`): the mandatory enforcement functions (`forcedTransfer(uint256)`, `setFrozenTokens`, `getFrozenTokens`) and their associated events require plaintext amounts, which are incompatible with FHE encrypted balances. See the technical doc for the full breakdown.

```
// Enable or disable allowlist enforcement (ALLOWLIST_ROLE)
function enableAllowlist(bool enabled) public onlyRole(ALLOWLIST_ROLE);

// Add or remove an address from the allowlist (ALLOWLIST_ROLE)
function setAddressAllowlist(address account, bool allowlisted) public
onlyRole(ALLOWLIST_ROLE);

// Partial ERC-7943 view checks (return false when allowlist is enabled and
address is not allowlisted)
function canSend(address account) public view returns (bool);
function canReceive(address account) public view returns (bool);
function canTransfer(address from, address to, uint256 amount) public view
returns (bool);
```

When the allowlist is enabled, any transfer where either party is not allowlisted reverts with `ERC7943CannotTransfer(from, to, 0)` (amount is always `0` since the actual amount is encrypted). The contract also reverts `ERC7943CannotReceive(to)` on mint and `ERC7943CannotSend(from)` on burn when the respective party is not allowlisted.

When the allowlist is disabled, all these checks are bypassed and the contract behaves identically to `CMTATConfidential`.

> **Note:** `canSend`, `canReceive`, and `canTransfer` (with `amount` ignored) are available in **all four variants** via the inherited `ValidationModule`. The Whitelist variant adds the allowlist dimension to these checks.

Forced Transfer

Enforcers can move tokens from frozen addresses for regulatory compliance. Forced transfers can be performed even when the contract is deactivated.

```
function forcedTransfer(
    address from,         // Must be frozen
    address to,           // Must not be address(0)
    externalEuint64 encryptedAmount,
    bytes calldata inputProof
) public onlyRole(FORCED_OPS_ROLE) returns (euint64 transferred);
```

Requirements:

- The `from` address must be frozen
- The `to` address must not be `address(0)` -- use `forcedBurn()` for burning
- Can be performed even when the contract is deactivated

> **Note:** This is intentionally stricter than standard CMTAT, which allows `forcedTransfer` on any address (frozen or not). CMTAT Confidential requires the address to be frozen first, creating an explicit audit trail (freeze event followed by forced transfer).

Forced Burn

Enforcers can burn tokens directly from frozen addresses without the holder's consent. This is the dedicated burn equivalent of `forcedTransfer`. Forced burns can be performed even when the contract is deactivated.

```
function forcedBurn(
    address from,         // Must be frozen
    externalEuint64 encryptedAmount,
    bytes calldata inputProof
) public onlyRole(FORCED_OPS_ROLE) returns (euint64 burned);
```

Requirements:

- The `from` address must be frozen
- Can be performed even when the contract is deactivated

> **Note:** Same freeze requirement as `forcedTransfer` for consistency. The enforcer creates the encrypted input specifying how many tokens to burn.

> ⚠️ **Always decrypt the returned `transferred` / `burned` handle to confirm the amount actually moved.**
> Because balances are encrypted, an amount greater than the target's balance does **not** revert — FHE
> subtraction saturates to `0`, so the call **succeeds and emits `ForcedTransfer` / `ForcedBurn` while moving**
> `0`. A successful transaction is therefore **not** proof that a seizure took effect. The event carries the
> **actual encrypted amount moved** (`transferred` / `burned`), so enforcement tooling must decrypt that handle
> off-chain (`fhevm.userDecryptEuint` / `publicDecryptEuint`) and verify it equals the intended amount before
> treating a court-ordered seizure or redemption as complete. This is an inherent property of FHE arithmetic (it
> applies to all transfers/burns; it is called out here because for forced operations a silent `0` can be
> mistaken for a completed compliance action).

Total Supply Visibility

By default the total supply is encrypted and inaccessible to third parties. Two mechanisms are available to open read access, gated by `SUPPLY_OBSERVER_ROLE` (observer list) or `SUPPLY_PUBLISHER_ROLE` (public disclosure).

Option 1 — Authorized observers (automatic, stays current) — `CMTATConfidential`, `CMTATConfidentialRuleEngine`, `CMTATConfidentialWhitelist`

Register addresses that will automatically receive ACL access to the total supply handle after every mint or burn:

```
// Grant SUPPLY_OBSERVER_ROLE to the compliance manager
await token.grantRole(SUPPLY_OBSERVER_ROLE, complianceManager.address);

// Grant SUPPLY_PUBLISHER_ROLE to the compliance manager (separate permission)
await token.grantRole(SUPPLY_PUBLISHER_ROLE, complianceManager.address);

// Register a regulator as a total supply observer (capped by
maxSupplyObservers, default 10)
await
token.connect(complianceManager).addTotalSupplyObserver(regulatorAddress);

// Remove an observer (stops future grants; past ACL grants are irrevocable)
await
token.connect(complianceManager).removeTotalSupplyObserver(regulatorAddress);

// Inspect the current observer list and cap
const observers = await token.totalSupplyObservers();
const cap = await token.maxSupplyObservers(); // default 10

// Adjust the cap (DEFAULT_ADMIN_ROLE only; cannot go below current observer
count)
```

```
await token.connect(admin).setMaxSupplyObservers(20);
```

Once registered, the observer can decrypt off-chain using the standard user-decryption flow:

```
const handle = await token.confidentialTotalSupply();
const supply = await fhevm.userDecryptEuint(FhevmType.euint64, handle,
tokenAddress, observer);
```

Option 2 — Public disclosure (anyone, irrevocable per handle) — all variants

Mark the current total supply handle as publicly decryptable. Any off-chain party can then request decryption via the Zama Relayer SDK without ACL access. After the next mint or burn, the new handle will not be publicly decryptable — call again if needed.

```
await token.connect(complianceManager).publishTotalSupply();
```

> ⚠ **Disclosure warning — cross-publication delta inference (audit finding OZ-L-01).**
> A single call only reveals the *aggregate* total supply — never individual balances or transfer amounts. But **repeated** publication is a leak channel: an observer who reads two published values `V1` (before) and `V2` (after) some mints/burns can compute the net change `|V2 − V1|`. If exactly one mint or burn happens between two publications, its amount is fully revealed, defeating the confidentiality of that operation. This is inherent to disclosing an aggregate that moves by discrete confidential amounts and **cannot be fully fixed in code** (a `SUPPLY_PUBLISHER_ROLE` holder can always disclose). A `counter ≥ k` gate on `publishTotalSupply()` would give k-anonymity against outside observers, but it would trade away the transparent, on-demand total supply (the exact figure could not be published right after a single mint or burn), so it was deliberately not implemented in favour of the operational mitigations below:
>
> - Treat publishing as a deliberate governance action, not a high-frequency automated one. Prefer a multisig/timelock over `SUPPLY_PUBLISHER_ROLE`.
> - Aggregate many supply-changing operations between publications; never publish with a single mint/burn in between. Enforce a minimum operation-count or time window between disclosures.
> - If per-operation confidentiality of issuance/redemption is required, prefer **Option 1** (authorized observers) over public disclosure, and disclose on a coarse cadence.

Mechanism	Availability	Access scope	Stays current after mint/burn
`addTotalSupplyObserver`	All variants except `CMTATConfidentialLite`	Specific registered addresses	Yes — re-granted automatically via `_afterMint` / `_afterBurn` hooks
`publishTotalSupply`	All variants	`SUPPLY_PUBLISHER_ROLE`	No — must be called again after each mint/burn

> **Gas note:** In `CMTATConfidential`, every mint or burn triggers `_updateTotalSupplyObserversACL()`, which iterates over all registered total supply observers and calls `FHE.allow()` for each one. Additionally, `_update` runs a chain of balance observer ACL grants. Keep both observer lists small to control gas costs per operation.

Token Name and Symbol

Update the token name or symbol post-deployment (requires `TOKEN_ATTRIBUTE_ROLE`):

```
function setName(string calldata name_) public onlyRole(TOKEN_ATTRIBUTE_ROLE);
function setSymbol(string calldata symbol_) public
onlyRole(TOKEN_ATTRIBUTE_ROLE);
```

Emits `Name(string indexed, string)` and `Symbol(string indexed, string)` respectively — same event signatures as CMTAT's `ERC20BaseModule`.

Token Metadata

Update token metadata inherited from CMTAT (requires `EXTRA_INFORMATION_ROLE`):

```
function setTokenId(string calldata tokenId_) public
onlyRole(EXTRA_INFORMATION_ROLE);
function setTerms(IERC1643CMTAT.DocumentInfo calldata terms_) public
onlyRole(EXTRA_INFORMATION_ROLE);
function setInformation(string calldata information_) public
onlyRole(EXTRA_INFORMATION_ROLE);
```

Read back via the corresponding getters: `tokenId()`, `terms()`, `information()`.

Document Management

Manage ERC-1643 documents attached to the token (requires `DOCUMENT_ROLE`):

```
function setDocument(bytes32 name, string calldata uri, bytes32 documentHash)
public onlyRole(DOCUMENT_ROLE);
function removeDocument(bytes32 name) public onlyRole(DOCUMENT_ROLE);
function getDocument(bytes32 name) public view returns (Document memory);
function getAllDocuments() public view returns (bytes32[] memory);
```

Pause / Unpause

Pause all transfers (requires `PAUSER_ROLE`):

```
function pause() public onlyRole(PAUSER_ROLE);
function unpause() public onlyRole(PAUSER_ROLE);
```

Freeze / Unfreeze

Freeze specific addresses (requires `ENFORCER_ROLE`):

```
function setAddressFrozen(address account, bool freeze) public
onlyRole(ENFORCER_ROLE);
function isFrozen(address account) public view returns (bool);
```

> **Warning:** `ENFORCER_ROLE` holders must never freeze `address(0)`. The upstream CMTAT `EnforcementModule` does not guard against it. Direct holder transfers (`confidentialTransfer`, `confidentialTransferAndCall`) use `address(0)` as a synthetic spender in the freeze check, so freezing it would block **all** holder-initiated transfers, effectively acting as a global pause without holding `PAUSER_ROLE`. See `CMTAT` for a detailed analysis and the upstream fix proposal.

Deactivate Contract

Permanently deactivate the contract (requires `DEFAULT_ADMIN_ROLE`, contract must be paused):

```
function deactivateContract() public onlyRole(DEFAULT_ADMIN_ROLE);
```

Operator System

Operators can transfer tokens on behalf of holders using `confidentialTransferFrom`. Unlike ERC-20 allowances, operators have time-limited unlimited access.

```
// Set operator for 24 hours
const expirationTimestamp = Math.floor(Date.now() / 1000) + 86400;
await token.connect(holder).setOperator(operatorAddress, expirationTimestamp);

// Check operator status
const isOp = await token.isOperator(holderAddress, operatorAddress);
```

Warning: Setting an operator grants them access to transfer ALL your tokens during the approval period.

Decrypting Balances

Only the balance holder can decrypt their balance:

```
import { FhevmType } from '@fhevm/hardhat-plugin';

// Get encrypted balance handle
const balanceHandle = await token.confidentialBalanceOf(holderAddress);

// Decrypt (only works for the holder)
const balance = await fhevm.userDecryptEuint(
  FhevmType.euint64,
  balanceHandle,
  tokenAddress,
  holder // Signer must be the balance owner
);
```

Public Disclosure

Holders can publicly disclose amounts:

```
// Request disclosure (makes handle publicly decryptable)
function requestDiscloseEncryptedAmount(euint64 encryptedAmount) public;

// Finalize with decryption proof
function discloseEncryptedAmount(
    euint64 encryptedAmount,
    uint64 cleartextAmount,
    bytes calldata decryptionProof
) public;
```

Deployment

```
import { ethers } from 'hardhat';

const extraInfoAttributes = {
  tokenId: 'TOKEN-001',
  terms: {
    name: 'Terms Document',
    uri: 'https://example.com/terms',
    documentHash: ethers.ZeroHash,
  },
  information: 'Security token for XYZ',
};

const token = await ethers.deployContract('CMTATConfidential', [
  'My Token',           // name
  'MTK',                // symbol
  'https://example.com/metadata', // contractURI
  6,                    // decimals (choose per token, e.g. 0, 6, 8)
  adminAddress,         // admin with DEFAULT_ADMIN_ROLE
  extraInfoAttributes,  // token metadata
]);

// Grant roles
await token.grantRole(MINTER_ROLE, minterAddress);
await token.grantRole(BURNER_ROLE, burnerAddress);
await token.grantRole(PAUSER_ROLE, pauserAddress);
await token.grantRole(ENFORCER_ROLE, enforcerAddress);     // freeze/unfreeze
addresses
await token.grantRole(FORCED_OPS_ROLE, enforcerAddress);   // forced transfer /
forced burn
```

Choosing Decimals

Decimals are configurable at deployment for both `CMTATConfidential` and `CMTATConfidentialLite`. The constructor argument order is: `name, symbol, contractURI, decimals, admin, extraInfoAttributes`.

> **Warning:** ERC-7984 balances use `euint64` (max `18,446,744,073,709,551,615` raw units). The maximum human-readable supply is `uint64 max / 10^decimals`:

Decimals	Max supply

Decimals	Max supply
0	~18.4 × 10^{18} tokens
6	~18.4 trillion tokens *(recommended default)*
8	~184 billion tokens
18	~18 tokens

Values above **18 are rejected** at construction (`CMTAT_DecimalsTooHigh`), because even a single token (`1 × 10^decimals` raw units) would overflow `uint64`. Values of 9 or more are technically valid but severely constrain the maximum supply — verify that `expectedMaxSupply × 10^decimals ≤ 18,446,744,073,709,551,615` before deploying.

Note: FHE overflow is silent — a mint or transfer that exceeds `uint64` max will transfer `0` without reverting.

Dependencies

Package	Version
`@fhevm/solidity`	0.11.1
`@fhevm/hardhat-plugin`	0.4.2
`@zama-fhe/relayer-sdk`	0.4.1
`@openzeppelin/contracts`	5.6.1
`@openzeppelin/contracts-upgradeable`	5.6.1
Submodule	
OpenZeppelin Confidential Contracts	v0.5.1
CMTAT	v3.3.0-rc1
RuleEngine	v3.0.0-rc4

Project Structure

```
CMTAT-Confidential/
├── contracts/
│   ├── CMTATConfidentialBase.sol                    # Abstract base (all
shared logic)
│   ├── deployment/
│   │   ├── CMTATConfidential.sol                    # Full variant (+ total
supply visibility)
│   │   ├── CMTATConfidentialLite.sol                # Lite variant (smaller,
no total supply module)
│   │   ├── CMTATConfidentialRuleEngine.sol          # Full variant +
RuleEngine transfer restrictions
│   │   └── CMTATConfidentialWhitelist.sol           # Full variant + on/off
allowlist enforcement
```

```
│   └── modules/
│       ├── ERC7984MintModule.sol                  # Mint with authorization
hook
│       ├── ERC7984BurnModule.sol                  # Burn with authorization
hook
│       ├── ERC7984EnforcementModule.sol           # Forced transfer and forced
burn
│       ├── ERC7984BalanceViewModule.sol           # Per-account balance
observers
│       ├── ERC7984PublishTotalSupplyModule.sol    # Public total supply
disclosure
│       ├── ERC7984TokenAttributeModule.sol        # Post-deployment
name/symbol (ERC-3643)
│       ├── ERC7984TotalSupplyViewModule.sol       # Total supply observer list
(auto ACL)
│       ├── ERC7984RuleEngineModule.sol            # RuleEngine storage,
checks, and notifications
│       └── CMTATConfidentialVersionModule.sol     # CMTAT Confidential version
override (1.0.0)
├── lib/
│   ├── CMTAT/                                 # CMTAT submodule (compliance
modules)
│   └── RuleEngine/                            # CMTA RuleEngine submodule
├── openzeppelin-confidential-contracts/       # OZ submodule (ERC7984)
├── doc/
│   ├── audit/                                 # Aderyn static analysis reports
│   ├── ERCSpecification/                      # Referenced ERC specs (ERC-7943,
etc.)
│   ├── specification/                         # Project specification
│   └── technical/                             # Per-variant and per-module
technical documentation
├── test/
│   ├── CMTATConfidential.test.ts                          # Full variant core
tests
│   ├── CMTATConfidentialLite.test.ts                      # Lite variant core
tests (shared suite)
│   ├── CMTATConfidentialRuleEngine.test.ts                # RuleEngine
variant tests
│   ├── CMTATConfidentialWhitelist.test.ts                 # Whitelist variant
tests
│   ├── ERC7984BalanceViewModule.test.ts           # Balance observer module
tests
│   ├── CMTATBaseFeatures.test.ts                  # CMTAT metadata:
name/symbol, terms, documents
│   ├── ERC7984EnforcementModule.test.ts           # Forced transfer/burn
module tests
│   ├── ERC7984PublishTotalSupplyModule.test.ts    # Public disclosure module
tests
│   ├── ERC7984TotalSupplyViewModule.test.ts       # Total supply observer
module tests
│   └── helpers/
│       ├── deploy.ts                          # Shared deploy helper + role
constants
│       └── core-tests.ts                      # Shared Mocha test suite
└── hardhat.config.ts
```

FAQ

1. As an issuer, can I burn tokens from a token holder without their consent?

Answer: Yes, as an issuer with the `FORCED_OPS_ROLE`, you can burn tokens from any holder without their consent using the `forcedBurn()` function.

How it works:

1. First freeze the holder's address using `setAddressFrozen(holderAddress, true)` (requires `ENFORCER_ROLE`)
2. Use the `forcedBurn()` function to burn tokens directly from the frozen address (requires `FORCED_OPS_ROLE`)
3. This function can be performed even when the contract is deactivated
4. Only accounts with `FORCED_OPS_ROLE` can execute forced burns

Use cases for regulatory compliance:

- Court orders requiring asset seizure
- Sanctions compliance
- Error correction (e.g., tokens sent to wrong address)

Code example:

```
// Step 1: Freeze the holder's address
await token.connect(enforcer).setAddressFrozen(holderAddress, true);

// Step 2: Create encrypted input for the amount to burn
const encryptedInput = await fhevm
    .createEncryptedInput(tokenAddress, enforcerAddress)
    .add64(amountToBurn)
    .encrypt();

// Step 3: Force burn tokens from the frozen address
await token.connect(enforcer)['forcedBurn(address,bytes32,bytes)'](
    holderAddress,                  // from (must be frozen)
    encryptedInput.handles[0],      // encrypted amount
    encryptedInput.inputProof       // ZKPoK proof
);
```

Note: The regular `burn()` function requires `BURNER_ROLE` and will fail if the target address is frozen. For frozen addresses, use `forcedBurn()` instead (requires `FORCED_OPS_ROLE`). The `from` address must be frozen before calling `forcedBurn()`. Note that `forcedTransfer()` reverts if `to` is `address(0)` -- use `forcedBurn()` for burning.

> **Design choice:** Standard CMTAT allows `forcedTransfer` and `forcedBurn` on any address (frozen or not). CMTAT Confidential intentionally requires the address to be frozen first, creating an explicit audit trail (freeze event followed by forced burn/transfer).

2. As a token holder, how do I transfer my tokens to another address?

Answer: Transfers in CMTAT Confidential use encrypted inputs to preserve confidentiality. Here's the complete process:

Step 1: Create an encrypted input

Encrypted inputs are data values submitted in ciphertext form, accompanied by **Zero-Knowledge Proofs of Knowledge (ZKPoKs)** to ensure validity without revealing the plaintext.

```
const encryptedInput = await fhevm
    .createEncryptedInput(tokenContractAddress, yourAddress)
    .add64(amount)  // Amount to transfer (will be encrypted)
    .encrypt();
```

Step 2: Call the transfer function

```
await token.confidentialTransfer(
    recipientAddress,
    encryptedInput.handles[0],  // externalEuint64 handle
    encryptedInput.inputProof   // ZKPoK proof
);
```

How validation works on-chain

1. **Input verification**: The `FHE.fromExternal()` function validates the ciphertext and ZKPoK
2. **Type conversion**: Converts `externalEuint64` into `euint64` for contract operations
3. **Balance check**: If balance is insufficient, transfer executes but transfers 0 (FHE doesn't reveal balance)

Alternative: Using an operator

You can authorize an operator to transfer on your behalf using time-limited approval:

```
const expirationTimestamp = Math.round(Date.now() / 1000) + 60 * 60 * 24; // 24
hours
await token.connect(holder).setOperator(operatorAddress, expirationTimestamp);

// Operator can now call confidentialTransferFrom
await token.connect(operator).confidentialTransferFrom(
    holderAddress,
    recipientAddress,
    encryptedAmount,
    inputProof
);
```

Important: Setting an operator allows them to transfer **all your tokens**. Carefully vet operators before approval.

Transfer requirements

- Your address must not be frozen
- The recipient address must not be frozen
- The contract must not be paused or deactivated

---

3. Can I deploy CMTAT Confidential contracts on Ethereum mainnet?

Answer: Yes - the Zama Protocol mainnet on Ethereum is now live.

How does it work?

Ethereum mainnet does not natively support FHE operations. The Zama Protocol uses a **coprocessor architecture** where the host chain (Ethereum) performs symbolic execution, and the actual FHE computations are performed off-chain by coprocessors. The infrastructure includes:

- **ACL (Access Control List)**: Manages permissions for encrypted data
- **FHEVMExecutor (Coprocessor)**: Performs encrypted computations off-chain
- **KMSVerifier**: Verifies decryption proofs from the Key Management System
- **InputVerifier**: Validates encrypted inputs and ZKPoKs

Where can you deploy?

Network	Chain ID	Status
Local development	31337	Supported (mock coprocessor via hardhat plugin)
Ethereum Sepolia	11155111	Supported (Zama testnet infrastructure)
Ethereum Mainnet	1	**Live**

Requirements for deployment

1. **Inherit from `ZamaEthereumConfig`**: This automatically configures coprocessor addresses:

```
import {ZamaEthereumConfig} from "@fhevm/solidity/config/ZamaConfig.sol";

contract MyToken is ERC7984, ZamaEthereumConfig {
    // Constructor automatically calls FHE.setCoprocessor()
}
```

2. **Target network must have Zama infrastructure**: The coprocessor contracts must be deployed and operational
3. **Users need compatible tools**: Client applications must use the Relayer SDK to create encrypted inputs and request decryptions

---

4. Is it possible to also provide privacy for addresses?

Answer: Not in this implementation. CMTAT Confidential does **not** provide encrypted addresses.
This project keeps Ethereum addresses public and only encrypts amounts/balances (`euint64`).

Encrypted addresses are technically possible in fhEVM using the `eaddress` type.

How it works

The fhEVM library provides the `eaddress` type which encrypts Ethereum addresses. This enables:

- Hiding sender and recipient addresses in transfers
- Omnibus account patterns where multiple users share a single on-chain address

Implementation: Omnibus pattern

The OpenZeppelin Confidential Contracts library includes `ERC7984Omnibus` extension:

1. Uses a single omnibus address visible on-chain
2. Maintains encrypted mappings of actual user addresses to balances
3. Transfers happen between encrypted addresses within the omnibus

```
function confidentialTransferFromOmnibus(
    address omnibusFrom,
    address omnibusTo,
    externalEaddress externalSender,      // encrypted sender
    externalEaddress externalRecipient,   // encrypted recipient
    externalEuint64 externalAmount,
    bytes calldata inputProof
) public virtual returns (euint64);
```

Trade-offs

Benefit	Cost
Address privacy	Higher gas costs for encrypted address operations
Regulatory compliance (omnibus accounts)	More complex user experience
Institutional custody patterns	Requires trusted omnibus operators

Regulatory considerations

Hiding participant identities may conflict with AML/KYC requirements in some jurisdictions. Consider your compliance obligations before implementing address privacy.

---

5. Is the total supply public or private information?

Answer: The total supply is **private** (encrypted) by default in ERC7984.

Technical details

- The `_totalSupply` variable is stored as `euint64` (encrypted unsigned 64-bit integer)
- The `confidentialTotalSupply()` function returns an encrypted handle, not a plaintext value
- Access is controlled by the ACL (Access Control List)

```
// Returns an encrypted handle (euint64), not the actual value
function confidentialTotalSupply() public view returns (euint64);
```

How to grant access to the total supply

CMTAT Confidential provides two built-in mechanisms:

Option A — Authorized observers (stays current automatically) — `ERC7984TotalSupplyViewModule`, `CMTATConfidential` only

Register specific addresses that automatically receive ACL access after every mint or burn:

```
token.addTotalSupplyObserver(regulatorAddress);  // re-granted on every
mint/burn
token.removeTotalSupplyObserver(regulatorAddress); // stops future grants
```

Once registered, the observer decrypts using standard user-decryption — see Decrypting Balances.

Option B — Public disclosure — `ERC7984PublishTotalSupplyModule`, available on both `CMTATConfidential` and `CMTATConfidentialLite`

Call `publishTotalSupply()` (requires `SUPPLY_PUBLISHER_ROLE`) to mark the current handle as publicly decryptable. Must be called again after each mint or burn since the handle changes. This will revert if total supply has never been initialized (no mint/burn yet).

```
token.publishTotalSupply();
```

> **Note:** `publishTotalSupply()` reverts until the total supply handle is initialized (i.e., at least one mint or burn has occurred).
>
> ⚠ **Cross-publication delta inference (audit finding OZ-L-01):** publishing repeatedly lets observers subtract consecutive values (`|V2 - V1|`) to recover the net minted/burned amount between disclosures — fully revealing a mint/burn amount if only one occurs in between. See the disclosure warning under Total Supply Visibility → Option 2 for operational mitigations.

Internally this calls `FHE.makePubliclyDecryptable()`, which triggers the following **asynchronous three-step process**:

Public decryption — step by step

Public decryption splits work between on-chain and off-chain:

Step 1: On-chain - Mark as publicly decryptable

The contract sets the ciphertext handle's status as publicly decryptable, **globally and permanently** authorizing any entity to request its off-chain cleartext value.

```
FHE.makePubliclyDecryptable(confidentialTotalSupply());
```

Step 2: Off-chain - Request decryption from KMS

Any off-chain client can submit the ciphertext handle to the Zama Relayer's Key Management System (KMS) using the Relayer SDK (`@zama-fhe/relayer-sdk`).

```
const result = await fhevmInstance.publicDecrypt([totalSupplyHandle]);
// Returns:
// - clearValues: mapping of handles to decrypted values
// - abiEncodedClearValues: ABI-encoded byte string of all cleartext values
// - decryptionProof: cryptographic proof from the KMS
```

Step 3: On-chain - Verify and use the decrypted value

The caller submits the cleartext and decryption proof back to a contract function. The contract calls `FHE.checkSignatures`, which reverts if the proof is invalid.

```
FHE.checkSignatures(handlesList, abiEncodedCleartexts, decryptionProof);
// Now you can use the verified cleartext value
```

Important: The decryption proof is cryptographically bound to the specific order of handles passed in the input array.

ACL permissions

To access encrypted values, accounts need proper ACL permissions:

Function	Purpose
`FHE.allow(handle, address)`	Permanent access for specific address
`FHE.allowThis(handle)`	Shorthand for `FHE.allow(handle, address(this))` - allow current contract
`FHE.allowTransient(handle, address)`	Temporary access (current transaction only)
`FHE.makePubliclyDecryptable(handle)`	Allow anyone to decrypt off-chain
`FHE.isAllowed(handle, address)`	Check if an address has access to a ciphertext
`FHE.isSenderAllowed(handle)`	Check if `msg.sender` has access to a ciphertext

Why keep total supply private?

- Prevents market manipulation based on supply information
- Protects issuer's business information
- Consistent with the privacy-first design of confidential tokens

Note: If you need public total supply, implement a function that goes through the full decryption process and emits the result as an event. Consider the privacy implications carefully.

---

6. As an issuer, can I get the non-encrypted balance of a token holder?

Answer: Yes, but only through the asynchronous decryption process -- there is no direct way to "read" a plaintext balance.

How to decrypt a holder's balance

The issuer must have ACL permission on the holder's balance ciphertext. By default, only the balance holder and the contract itself have access. To allow the issuer to decrypt:

Option 1: Grant the issuer ACL access (on-chain)

The contract can grant ACL permission to the issuer's address on the holder's balance handle. This would need to be built into the contract logic (e.g., a function that grants the issuer access to a specific balance):

```
// Inside the contract, an admin function could grant access
function grantBalanceAccess(address holder, address viewer) public
onlyRole(DEFAULT_ADMIN_ROLE) {
    FHE.allow(confidentialBalanceOf(holder), viewer);
}
```

Once the issuer has ACL access, they can decrypt the balance off-chain:

```
const balanceHandle = await token.confidentialBalanceOf(holderAddress);
const balance = await fhevm.userDecryptEuint(
    FhevmType.euint64,
    balanceHandle,
    tokenAddress,
    issuer // Signer with ACL access
);
```

Option 2: Public decryption

The holder (or a contract function) can mark their balance as publicly decryptable, then anyone can request decryption through the three-step process (see FAQ #5).

Do I need a viewing key?

Zama's FHEVM does **not** use a "viewing key" model like some other privacy systems (e.g., Zcash). Instead, access is controlled through the **ACL (Access Control List)**:

Concept	Zama FHE Approach

Concept	Zama FHE Approach
Viewing key	Not applicable -- uses ACL permissions instead
Grant read access	`FHE.allow(handle, address)` grants permanent access to a specific ciphertext
Temporary access	`FHE.allowTransient(handle, address)` grants access for the current transaction only
Public access	`FHE.makePubliclyDecryptable(handle)` allows anyone to decrypt

The ACL model is more flexible than viewing keys: you can grant access per-ciphertext, per-address, and with different durations (permanent, transient, or public).

Can third-parties read balances?

Yes, if they are granted ACL permission. The contract logic decides who gets access.

The handle staleness problem

Every FHE arithmetic operation (mint, transfer, burn) produces a *new* ciphertext handle for the affected balance. A third party who was granted ACL access to an old handle loses the ability to read the balance the moment that handle is replaced. Access must therefore be re-granted after every update — it cannot be set once and forgotten.

CMTAT Confidential solves this with the `ERC7984ObserverAccess` extension. After every `_update`, the contract automatically calls `FHE.allow()` on the new balance handle for each registered observer, keeping their ACL access current without any manual intervention.

Two observer slots per holder

`ERC7984BalanceViewModule` (built on `ERC7984ObserverAccess`) provides two independent observer slots per address:

Slot	Set by	Typical use
Holder observer (`setObserver`)	The holder themselves	Personal wallet app, portfolio dashboard
Role observer (`setRoleObserver`, requires `OBSERVER_ROLE`)	The issuer / compliance team	Regulator, auditor, compliance tool

Both slots receive `FHE.allow()` on every balance update automatically.

Observer ACL scope: balance *and* transfer amount

On every `_update`, observers are granted ACL access to **two** ciphertext handles:

1. The account's new **balance** handle — so the observer can read the current balance at any time
2. The **transferred amount** handle — so the observer can reconstruct individual transaction amounts

This is intentional design for regulatory compliance: a compliance observer needs transfer-level granularity, not just balance snapshots. An observer set via `setRoleObserver` should therefore be treated as having access to individual transaction amounts for the account they are observing.

Decrypting as an observer

Once access is granted the observer can decrypt off-chain through the standard user-decryption flow:

```
// Read the current handle
const handle = await token.confidentialBalanceOf(holderAddress);

// Decrypt — observer must have ACL access on that handle
const balance = await fhevm.userDecryptEuint(
    FhevmType.euint64,
    handle,
    tokenAddress,
    observer // signer with ACL access
);
```

ACL access is permanent and cannot be revoked

`FHE.allow()` is a one-way operation — once an observer is granted access to a handle, that access cannot be removed. Removing an observer with `setObserver` / `setRoleObserver` only stops *future* grants: the observer retains read access to all handles they were allowed on before removal.

Common patterns

Party	How access is granted
The holder themselves	Automatically by the contract on every transfer/mint
Regulatory observer	Issuer calls `setRoleObserver(holder, regulatorAddress)`
Personal observer	Holder calls `setObserver(holderAddress, walletAppAddress)`
Auditor (temporary)	Admin calls `FHE.allowTransient()` during the audit transaction

---

7. Can the issuer compute the inputProof for operations if the issuer is not the token holder?

Answer: Yes. The `inputProof` (Zero-Knowledge Proof of Knowledge) is generated by whoever **creates the encrypted input**, not by the token holder.

How encrypted inputs work

When any party calls a function that requires an encrypted amount (mint, burn, forcedBurn, forcedTransfer), they:

1. **Choose the plaintext amount** they want to encrypt
2. **Generate the encrypted input** using the FHEVM client library
3. **The library produces** a ciphertext handle + a ZKPoK proof

The ZKPoK proves that the caller **knows** the plaintext value inside the ciphertext, without revealing it. It does not prove anything about token ownership or balances.

Example: Issuer burns tokens from a holder

```
// The enforcer (issuer) creates the encrypted input themselves
const encryptedInput = await fhevm
    .createEncryptedInput(tokenAddress, enforcerAddress) // enforcer's address,
NOT holder's
    .add64(amountToBurn)
    .encrypt();

// The enforcer calls forcedBurn with their own proof
await token.connect(enforcer)['forcedBurn(address,bytes32,bytes)'](
    holderAddress,                  // target to burn from
    encryptedInput.handles[0],      // enforcer's encrypted amount
    encryptedInput.inputProof       // enforcer's proof
);
```

Key points

Question	Answer
Who generates the inputProof?	The caller of the function (e.g., the enforcer/issuer)
Does the holder need to participate?	No -- forced operations don't require holder involvement
Is the proof tied to the holder's balance?	No -- it proves knowledge of the encrypted input value, not the balance
Can the issuer specify any amount?	Yes, but if the amount exceeds the holder's balance, FHE transfers/burns 0 silently

The `inputProof` is tied to the **caller's address** and the **contract address** (passed to `createEncryptedInput`), not to the token holder. This is what enables administrative operations like `forcedBurn` and `forcedTransfer` without the holder's participation.

Glossary

Term	Definition
FHE (Fully Homomorphic Encryption)	Cryptographic scheme that enables arbitrary computations directly on ciphertext without ever decrypting it. The result, once decrypted, is identical to what would have been produced on the plaintext. It is the core primitive behind confidential balances and transfers in CMTAT Confidential.
euint64	Encrypted unsigned 64-bit integer — the on-chain type used to store confidential balances and transfer amounts. It is a pointer to a ciphertext managed by the Zama coprocessor network, not a raw encrypted value. Maximum representable value is ~18.4×10^{18}.

Term	Definition
externalEuint64	The user-facing form of an encrypted 64-bit integer: a ciphertext handle produced by the client library and submitted alongside a ZKPoK. Converted to `euint64` on-chain via `FHE.fromExternal()` after the proof is verified.
ZKPoK (Zero-Knowledge Proof of Knowledge)	A cryptographic proof that the submitter knows the plaintext inside an encrypted input, without revealing that plaintext. Required for every encrypted input to prevent malleability and replay attacks. Verified on-chain by the InputVerifier contract.
Ciphertext Handle	A 32-byte pointer returned by every FHE operation (e.g., `euint64`). It references the actual ciphertext stored and computed by the coprocessor network, not the ciphertext itself. Arithmetic on handles triggers off-chain coprocessor computation and produces new handles.
ACL (Access Control List)	On-chain permission registry that tracks which addresses may decrypt a given ciphertext handle. Access is granted with `FHE.allow()` (permanent) or `FHE.allowTransient()` (current transaction only). Without ACL access, a party cannot request decryption of a handle.
Coprocessor (FHEVMExecutor)	Off-chain network of nodes that performs the actual FHE computations. When a Solidity contract calls an FHE operation, the host chain emits an event; coprocessors pick it up, compute the result, and make the new ciphertext available.
KMS (Key Management System)	Zama's key management infrastructure that holds the FHE private key in a distributed manner using MPC. Decryption requests are sent to the KMS, which returns a decrypted value together with a cryptographic proof that can be verified on-chain.
MPC (Multi-Party Computation)	Threshold cryptography protocol used by the KMS so that no single node ever holds the complete FHE private key. Decryption only succeeds when a quorum of KMS nodes cooperates, preventing any single point of compromise.
Symbolic Execution	The on-chain execution model for FHE operations: the EVM does not perform the actual FHE computation — it only records an intent (emits an event with a new handle). The coprocessor network executes the computation off-chain asynchronously.
ERC-7984	The Confidential Fungible Token standard (drafted by OpenZeppelin). It defines the interface for tokens with encrypted balances and transfer amounts, including the operator system, disclosure mechanism, and ACL-based access patterns used by CMTAT Confidential.
Operator	An address authorized by a token holder to call `confidentialTransferFrom` on their behalf. Unlike ERC-20 allowances, operators are granted time-limited *unlimited* access — they may transfer any amount until the approval timestamp expires.

Term	Definition
Observer	An ACL-authorized third party (e.g., a regulator or auditor) granted read access to one or more encrypted balance handles. Implemented via `ERC7984ObserverAccess`: the contract grants `FHE.allow()` on balances affected by each transfer, giving the observer a continuously updated view.

References

- CMTAT - Capital Markets and Technology Association Token Standard
- Openzeppelin
  - OpenZeppelin Confidential Contracts
  - ERC-7984 Specification
- Zama
  - Zama Protocol Litepaper
  - Zama FHEVM Documentation
  - Zama FHE Types
  - Encrypted Inputs
  - Access Control List (ACL)
  - Decryption
- Part of this project was carried out with the help of Claude Code and Codex

License