**7. Developer Guide: Extending Mango with Custom Transforms**

Mango’s cryptographic engine (CryptoLib.cs) ships with ~40 production-grade transforms. These form the building blocks of every encryption sequence. However, developers are encouraged to extend Mango by creating new transforms tailored to specific goals — whether for research, experimentation, or domain-specific optimization.

Once registered, custom transforms immediately become available throughout Mango’s ecosystem:

* ✅ Workbench interface
* ✅ CLI commands like add transform
* ✅ Automated discovery engines (run munge, optimize sequence)
* ✅ Production use via Encrypt() and Decrypt() APIs

**7.1 Adding a Transform to the Core Engine**

To add a new transform, you will:

1. Write the transform function in CryptoLib.cs.
2. Register it in the transform registry with a unique ID and name.
3. Optionally create an inverse function (if the transform isn’t self-reversing).

**7.2 Involutory (Self-Reversing) Transform Example**

Involutory transforms are their own inverse and simplify registration and reversibility.

private void NibbleInterleaverTx(byte[] input, byte coin)

{

for (int i = 0; i < input.Length; i++)

{

if (i % 2 == 1)

{

byte high = (byte)((input[i] & 0xF0) >> 4);

byte low = (byte)(input[i] & 0x0F);

input[i] = (byte)((low << 4) | high);

}

}

}

**Registry entry:**

{ 39, new TransformInfo

{

Name = "NibbleInterleaverTx",

Id = 39,

InverseId = 39, // Self-inverting

Implementation = NibbleInterleaverTx,

}

}

**7.3 Forward & Inverse Transform Pair**

If your transform is **not self-inverting**, define an explicit forward and inverse.

private void MaskedCascadeSubFwdFbTx(byte[] input, byte coin)

{

var prng = new TOM\_Random(cryptoLib: this, seed: coin);

for (int i = 0; i < input.Length; i++)

{

byte randomMask = prng.NextMask();

byte transformed = (byte)(input[i] ^ randomMask);

transformed = (byte)Tables.SBox[CBox[transformed]];

input[i] = transformed;

}

}

private void MaskedCascadeSubInvFbTx(byte[] input, byte coin)

{

var prng = new TOM\_Random(cryptoLib: this, seed: coin);

for (int i = 0; i < input.Length; i++)

{

byte transformed = Tables.InverseSBox[input[i]];

transformed = InverseCBox[transformed];

byte randomMask = prng.NextMask();

input[i] = (byte)(transformed ^ randomMask);

}

}

**Registry entries:**

{ 35, new TransformInfo

{

Name = "MaskedCascadeSubFwdFbTx",

Id = 35,

InverseId = 36,

Implementation = MaskedCascadeSubFwdFbTx,

}

},

{ 36, new TransformInfo

{

Name = "MaskedCascadeSubInvFbTx",

Id = 36,

InverseId = 35,

Implementation = MaskedCascadeSubInvFbTx,

}

}

**7.4 Understanding Coins and CBox**

Modern Mango does **not use an IV**. Instead:

* **Coin Table**: A 256-byte deterministic shuffle table derived from the **user’s password and salt**. It is generated during session startup using either PBKDF2 or a fast SHA256-based fallback. Each transform receives a positionally indexed coin from this table, introducing session-specific but deterministic entropy. This allows for transform-level variation without requiring block-level feedback modes like CBC.
* **CBox / InverseCBox**: Substitution tables generated from a hash of the **input data itself**. This creates a per-message entanglement layer that disrupts fixed patterns and aligns transform behavior with the content being encrypted. CBox is used in nonlinear mappings such as SBox[CBox[x]], ensuring the same transform behaves differently for different inputs — enhancing security through deterministic chaos.

**🧠 Note:**  
The CBox plays a pivotal role not only during encryption but also during **sequence discovery**. Because CBox is generated from the input data itself, it introduces deterministic variability into transform behavior. As a result, when Munge searches for high-performing sequences, it is indirectly optimizing for sequences that align well with the structural patterns expressed through the CBox. In this way, CBox acts as a hidden shaping force — guiding the discovery of different “best” sequences for different types of data.

**7.5 Transform Safety & Reversibility**

Every transform must:

* Explicitly declare its InverseId in the registry.
* Respect session state (CBox, CoinTable, PRNG).
* Be vetted for correctness and invertibility.

This ensures that any sequence composed of registered transforms is **reversible**, provided the original key, salt, and structure are known.

**7.6 Integration in the Workbench**

Once registered, your new transform:

* Appears in the numbered menu
* Can be referenced by name in CLI commands (add transform YourTx)
* Can be discovered by run munge, optimize sequence, etc.
* Is validated automatically by Workbench test pipelines (e.g., run regression tests)

**7.7 Real-World Usage via CryptoLib**

All transforms — custom or stock — are available to CryptoLib.

Example:

var crypto = new CryptoLib("my password", new CryptoLibOptions(salt));

var profile = new InputProfile("Example", new (byte, byte)[]

{

(35, 1),

(36, 1)

}, globalRounds: 6);

byte[] encrypted = crypto.Encrypt(profile.Sequence, profile.GlobalRounds, input);

byte[] decrypted = crypto.Decrypt(encrypted);

Your custom transform is just another building block — and will be evaluated, scored, and reversed like any other.

**7.8 Closing Notes**

* The **CryptoLib** engine is modular — it only cares about transform IDs and round counts.
* The **Workbench** provides a rich testbed for iterating and verifying your additions.
* All production sequences (InputProfiles) were discovered using the Workbench’s automation features (Munge, BTR).
* To build trustworthy transforms, always validate **reversibility**, **non-linearity**, and **statelessness**.