# 🛠️ Developer Guide: Extending Mango with Custom Transforms

The Mango cryptographic system ships with ~40 production-ready transforms. However, developers are encouraged to extend the system by creating new transforms tailored to specific security goals, input characteristics, or research interests.  
  
Custom transforms integrate seamlessly once registered. They immediately become available to:

* The Workbench menu
* Commands like add transform
* Sequence evaluation systems (e.g., run munge, run best fit sequence)

## 🔁 Self-Reversing (Involutory) Transform Example

Involutory transforms are their own inverse. This reduces registration complexity and makes reversibility automatic.  
  
Example: NibbleInterleaverTx

[MethodImpl(MethodImplOptions.AggressiveInlining)]  
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,  
 BenchmarkTimeMs = 0.0  
 }  
}

## 🔄 Forward & Inverse Transform Pair

Some transforms are not self-reversing and require explicitly defined inverse logic. Mango handles this seamlessly through paired TransformInfo entries.  
  
These often use the coin model to entangle the transform with session- and input-specific state.

Forward Transform: MaskedCascadeSubFwdFbTx

[MethodImpl(MethodImplOptions.AggressiveInlining)]  
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 transformedByte = (byte)(input[i] ^ randomMask);  
 transformedByte = (byte)Tables.SBox[CBox[transformedByte]];  
 input[i] = transformedByte;  
 }  
}

Inverse Transform: MaskedCascadeSubInvFbTx

[MethodImpl(MethodImplOptions.AggressiveInlining)]  
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 transformedByte = (byte)Tables.InverseSBox[input[i]];  
 transformedByte = InverseCBox[transformedByte];  
 byte randomMask = prng.NextMask();  
 input[i] = (byte)(transformedByte ^ randomMask);  
 }  
}

## 🔗 Registry Entries

{ 35, new TransformInfo  
 {  
 Name = "MaskedCascadeSubFwdFbTx",  
 Id = 35,  
 InverseId = 36,  
 Implementation = MaskedCascadeSubFwdFbTx,  
 BenchmarkTimeMs = 0.0  
 }  
},  
{ 36, new TransformInfo  
 {  
 Name = "MaskedCascadeSubInvFbTx",  
 Id = 36,  
 InverseId = 35,  
 Implementation = MaskedCascadeSubInvFbTx,  
 BenchmarkTimeMs = 0.0  
 }  
}

## 🧠 Coin Theory & Dynamic Entanglement

Mango employs a dual model for dynamic, input-dependent behavior:  
  
• Coins: A 256-byte table of deterministic selectors, derived from hashing the input and IV. Each transform accesses its own indexed coin value, adding controlled entropy and variation.  
• Pseudo-Random Mutation: Transforms like MaskedCascadeSubFwdFbTx utilize deterministic, key-seeded randomness via TOM\_Random. This achieves the \*spirit\* of feedback — dynamic behavior that varies with the session — without requiring block-to-block chaining (as in CBC).

## 🔐 Input Entanglement Techniques

CBox: Mango generates a session-dependent substitution and masking table from the user’s password or seed. Used in constructs like:  
  
transformed = (byte)Tables.SBox[CBox[input[i]]];  
  
This provides nonlinear, key-tied behavior throughout many transforms, ensuring input/output correlation is disrupted in a deterministic but secure manner.

## 🔁 Sequence Reversibility & Transform Safety

Each transform:  
• Declares its InverseId in the registry  
• Respects per-input and per-session state  
• Is vetted for reversibility (or paired with its inverse)  
  
This guarantees that all sequences run through Mango are decryptable if the correct key, structure, and parameters are used.

## 📦 InputProfiles and Portability

An InputProfile is a pre-built configuration describing how to transform a specific kind of data. It includes the transform sequence, per-transform rounds (TR), and global rounds (GR).  
  
The core CryptoLib engine is agnostic to profiles — it simply consumes the provided configuration.

Creating a Custom InputProfile:  
1. Add a new `InputType` enum entry.  
2. Update `TestInputGenerator` in `Utilities.cs` to load sample or canned input.  
3. Run `run munge`, `optimize sequence`, or `run best fit sequence` to discover the best sequence for your custom InputType.  
4. Save the result as a new InputProfile.  
5. Pass that InputProfile to `Encrypt()` .

Example InputProfile:  
the format is transform id, transform rounds. Finally, the entire sequence is run global rounds times (6 in this case)  
  
{ "Combined", new InputProfile("Combined", new (byte, byte)[]   
{  
 (8, 3), // ButterflyTx, 3 rounds  
 (10, 1), // SubBytesXorMaskInvTx, 1 round  
 (31, 1), // ButterflyWithRotationFwdTx, 1 round  
 (9, 1), // SubBytesXorMaskFwdTx, 1 round  
 (31, 1) // ButterflyWithRotationFwdTx, 1 round  
}, 6) // then the entire sequence is run 6 times  
}

The sample program MangoAC demonstrates how InputProfiles are used.  
To replace one of Mango’s canned InputProfiles with your own, you would just call Encrypt() with your custom sequence and global rounds.

internal class MangoAC

{

static void Main(string[] args)

{

// 🔐 Step 1: Create your cryptographic engine

var crypto = new CryptoLib("my password");

// 📦 Step 2: Load or define your input data

byte[] input = Enumerable.Range(0, 256).Select(i => (byte)i).ToArray();

// 🔍 Step 3: Profile the input (detect type, best sequence + rounds)

InputProfile profile = InputProfiler.GetInputProfile(input);

// 🔒 Step 4: Encrypt using adaptive configuration

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

// 🔓 Step 5: Decrypt (CryptoLib pulls everything it needs from the header)

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

// ✅ Step 6: Verify

bool match = input.SequenceEqual(decrypted);

Console.WriteLine(match ? "✅ Decryption successful!" : "❌ Decryption failed.");

}

}