Privacy-Preserving Cross-Platform User Identification

## Technical Appendix

Generated: June 08, 2025

# Technical Implementation Details

## 1. Privacy Protection Module Implementation

The privacy protection module implements multiple privacy-preserving techniques:

class PrivacyProtector:  
 def \_\_init\_\_(self, config):  
 self.epsilon = config.get('dp\_epsilon', 1.0)  
 self.delta = config.get('dp\_delta', 1e-5)  
 self.k\_anonymity = config.get('k\_anonymity', 5)  
 self.l\_diversity = config.get('l\_diversity', 3)  
   
 def add\_differential\_privacy\_noise(self, data, sensitivity=1.0):  
 """Add Laplace noise for differential privacy."""  
 scale = sensitivity / self.epsilon  
 noise = np.random.laplace(0, scale, data.shape)  
 return data + noise  
   
 def apply\_k\_anonymity(self, df, quasi\_identifiers):  
 """Apply k-anonymity by generalizing quasi-identifiers."""  
 grouped = df.groupby(quasi\_identifiers)  
 small\_groups = grouped.filter(lambda x: len(x) < self.k\_anonymity)  
   
 if len(small\_groups) > 0:  
 df = self.\_generalize\_small\_groups(df, quasi\_identifiers, small\_groups)  
   
 return df  
   
 def secure\_multiparty\_computation(self, embeddings1, embeddings2):  
 """Compute similarity using additive secret sharing."""  
 # Split embeddings into random shares  
 shares1\_a = np.random.random(embeddings1.shape)  
 shares1\_b = embeddings1 - shares1\_a  
   
 shares2\_a = np.random.random(embeddings2.shape)  
 shares2\_b = embeddings2 - shares2\_a  
   
 # Compute similarity on shares  
 similarity\_a = np.dot(shares1\_a, shares2\_a.T)  
 similarity\_b = np.dot(shares1\_b, shares2\_b.T)  
   
 return similarity\_a + similarity\_b

## 2. Multi-Modal Feature Extraction

The system extracts features from four different modalities:

class MultiModalFeatureExtractor:  
 def \_\_init\_\_(self, config):  
 self.semantic\_embedder = SemanticEmbedder(config)  
 self.network\_embedder = NetworkEmbedder(config)  
 self.temporal\_embedder = TemporalEmbedder(config)  
 self.profile\_embedder = ProfileEmbedder(config)  
   
 def extract\_features(self, user\_data):  
 """Extract multi-modal features for a user."""  
 features = {}  
   
 # Semantic features from text content  
 if 'text\_data' in user\_data:  
 features['semantic'] = self.semantic\_embedder.fit\_transform(  
 user\_data['text\_data']  
 )  
   
 # Network features from connections  
 if 'network\_data' in user\_data:  
 features['network'] = self.network\_embedder.fit\_transform(  
 user\_data['network\_data'], user\_data['platform']  
 )  
   
 # Temporal features from activity patterns  
 if 'temporal\_data' in user\_data:  
 features['temporal'] = self.temporal\_embedder.fit\_transform(  
 user\_data['temporal\_data'], user\_data['platform'],  
 'timestamp', 'user\_id'  
 )  
   
 # Profile features from user attributes  
 if 'profile\_data' in user\_data:  
 features['profile'] = self.profile\_embedder.fit\_transform(  
 user\_data['profile\_data']  
 )  
   
 return features

## 3. Advanced Fusion Mechanisms

class CrossModalAttention(nn.Module):  
 def \_\_init\_\_(self, d\_model, num\_heads=16):  
 super().\_\_init\_\_()  
 self.d\_model = d\_model  
 self.num\_heads = num\_heads  
 self.head\_dim = d\_model // num\_heads  
   
 self.q\_linear = nn.Linear(d\_model, d\_model)  
 self.k\_linear = nn.Linear(d\_model, d\_model)  
 self.v\_linear = nn.Linear(d\_model, d\_model)  
 self.out\_linear = nn.Linear(d\_model, d\_model)  
   
 def forward(self, query, key, value, mask=None):  
 batch\_size = query.size(0)  
   
 # Linear transformations and split into heads  
 Q = self.q\_linear(query).view(batch\_size, -1, self.num\_heads, self.head\_dim)  
 K = self.k\_linear(key).view(batch\_size, -1, self.num\_heads, self.head\_dim)  
 V = self.v\_linear(value).view(batch\_size, -1, self.num\_heads, self.head\_dim)  
   
 # Transpose for attention computation  
 Q = Q.transpose(1, 2)  
 K = K.transpose(1, 2)  
 V = V.transpose(1, 2)  
   
 # Scaled dot-product attention  
 scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.head\_dim)  
   
 if mask is not None:  
 scores = scores.masked\_fill(mask == 0, -1e9)  
   
 attention\_weights = F.softmax(scores, dim=-1)  
 attended\_values = torch.matmul(attention\_weights, V)  
   
 # Concatenate heads and apply output linear layer  
 attended\_values = attended\_values.transpose(1, 2).contiguous().view(  
 batch\_size, -1, self.d\_model  
 )  
   
 return self.out\_linear(attended\_values)

# Configuration Examples

## 1. Basic Configuration

Standard configuration for development and testing:

{  
 "privacy": {  
 "dp\_epsilon": 1.0,  
 "dp\_delta": 1e-5,  
 "k\_anonymity": 5,  
 "l\_diversity": 3,  
 "enable\_smpc": true  
 },  
 "embeddings": {  
 "semantic\_embedding\_dim": 384,  
 "network\_embedding\_dim": 256,  
 "temporal\_embedding\_dim": 128,  
 "profile\_embedding\_dim": 128  
 },  
 "fusion": {  
 "method": "cross\_modal\_attention",  
 "hidden\_dim": 256,  
 "num\_heads": 16,  
 "dropout": 0.1  
 },  
 "ensemble": {  
 "gsmua": {  
 "hidden\_dim": 256,  
 "num\_heads": 8,  
 "dropout": 0.1  
 },  
 "frui\_p": {  
 "iterations": 5,  
 "damping": 0.85,  
 "weighted\_propagation": true  
 },  
 "lightgbm": {  
 "num\_estimators": 500,  
 "learning\_rate": 0.05,  
 "max\_depth": 6  
 },  
 "cosine": {  
 "threshold": 0.7,  
 "normalize\_scores": true  
 }  
 },  
 "performance": {  
 "batch\_size": 32,  
 "num\_workers": 4,  
 "gpu\_acceleration": true,  
 "cache\_embeddings": true  
 }  
}

## 2. High Privacy Configuration

Configuration optimized for maximum privacy protection:

{  
 "privacy": {  
 "dp\_epsilon": 0.1,  
 "dp\_delta": 1e-6,  
 "k\_anonymity": 10,  
 "l\_diversity": 5,  
 "enable\_smpc": true,  
 "noise\_multiplier": 2.0,  
 "clip\_norm": 1.0  
 },  
 "embeddings": {  
 "add\_noise\_to\_embeddings": true,  
 "embedding\_noise\_scale": 0.1,  
 "use\_private\_aggregation": true  
 },  
 "ensemble": {  
 "privacy\_aware\_combination": true,  
 "secure\_aggregation": true,  
 "differential\_privacy\_per\_matcher": true  
 },  
 "compliance": {  
 "gdpr\_mode": true,  
 "ccpa\_mode": true,  
 "audit\_logging": true,  
 "consent\_required": true,  
 "data\_minimization": true  
 }  
}

## 3. Production Configuration

Configuration for production deployment with scalability:

{  
 "scalability": {  
 "distributed\_processing": true,  
 "num\_workers": 16,  
 "batch\_size": 128,  
 "gpu\_acceleration": true,  
 "memory\_optimization": true  
 },  
 "monitoring": {  
 "enable\_metrics": true,  
 "log\_level": "INFO",  
 "performance\_tracking": true,  
 "privacy\_budget\_monitoring": true  
 },  
 "deployment": {  
 "containerized": true,  
 "kubernetes\_ready": true,  
 "health\_checks": true,  
 "auto\_scaling": true  
 },  
 "security": {  
 "encryption\_at\_rest": true,  
 "encryption\_in\_transit": true,  
 "secure\_key\_management": true,  
 "access\_control": true  
 }  
}

# API Documentation

## 1. Main System API

from src.models.cross\_platform\_identifier import CrossPlatformUserIdentifier  
  
# Initialize the system  
identifier = CrossPlatformUserIdentifier(config\_path="config.json")  
  
# Generate synthetic data for testing  
identifier.generate\_synthetic\_data(  
 num\_users=1000,  
 overlap\_ratio=0.7,  
 output\_dir="data/synthetic"  
)  
  
# Load real data  
data = identifier.load\_data({  
 'linkedin': 'data/linkedin/',  
 'instagram': 'data/instagram/',  
 'ground\_truth': 'data/ground\_truth.csv'  
})  
  
# Preprocess data  
processed\_data = identifier.preprocess\_data(data)  
  
# Extract features  
features = identifier.extract\_features(processed\_data)  
  
# Perform matching  
matches = identifier.match\_users(  
 features['linkedin'],  
 features['instagram'],  
 privacy\_level='medium'  
)  
  
# Evaluate results  
if 'ground\_truth' in data:  
 metrics = identifier.evaluate\_matches(matches, data['ground\_truth'])  
 print(f"F1-Score: {metrics['f1\_score']:.3f}")  
 print(f"Precision: {metrics['precision']:.3f}")  
 print(f"Recall: {metrics['recall']:.3f}")  
  
# Generate privacy report  
privacy\_report = identifier.generate\_privacy\_report()  
print(f"Privacy Budget Used: {privacy\_report['epsilon\_used']}")  
print(f"Compliance Status: {privacy\_report['compliance\_status']}")

## 2. Privacy Protection API

from src.utils.privacy import PrivacyProtector  
  
# Initialize privacy protector  
protector = PrivacyProtector({  
 'epsilon': 1.0,  
 'delta': 1e-5,  
 'k\_anonymity': 5  
})  
  
# Record user consent  
consent\_id = protector.record\_consent(  
 user\_id="user123",  
 purpose="user\_identification",  
 granted=True,  
 data\_types=["profile", "posts", "network"]  
)  
  
# Check consent before processing  
if protector.check\_consent("user123", "user\_identification"):  
 # Apply differential privacy to embeddings  
 private\_embeddings = protector.anonymize\_embeddings(embeddings)  
   
 # Apply k-anonymity to profile data  
 anonymous\_profiles = protector.apply\_k\_anonymity(  
 profiles\_df,   
 quasi\_identifiers=['age\_group', 'location', 'occupation']  
 )  
   
 # Secure similarity computation  
 similarity\_scores = protector.secure\_multiparty\_computation(  
 embeddings1, embeddings2  
 )  
  
# Generate compliance report  
report = protector.generate\_privacy\_report()