A Hybrid Identity-Based Signature and Elliptic Curve Diffie–Hellman Key Exchange Protocol with HKDF-Based Session Key Derivation for AMI

November 6, 2025

Abstract

The Advanced Metering Infrastructure (AMI) is a critical component of the smart grid, but its reliance on public networks exposes it to significant security threats. Many existing protocols, such as the one proposed by Hasan et al., depend on a centralized Certificate Authority (CA) for trust, creating a single point of failure and a severe scalability bottleneck for large-scale deployments. This report details the design of a novel protocol, the Hybrid IBE-Authenticated Key Exchange (IBE-AKE-AMI), which addresses this flaw by replacing the CA with an Identity-Based Signature (IBS) scheme. In this model, a device's unique identity serves as its public key, eliminating certificate management. Furthermore, this work identifies and corrects a critical, unstated security flaw in the baseline protocol: its lack of Perfect Forward Secrecy (PFS). The proposed IBE-AKE-AMI protocol integrates an ephemeral Elliptic Curve Diffie-Hellman (ECDH) exchange with the IBS authentication, successfully achieving PFS.

1 Introduction

The Advanced Metering Infrastructure (AMI) is the foundational communication network for the modern smart grid, enabling real-time, bi-directional communication between utility providers and millions of consumer-side Smart Meters (SMs).[1] This network, which also includes Data Concentrator Units (DCUs) and Head-End Systems (HES), is responsible for transmitting sensitive billing data, energy consumption patterns, and critical grid control commands.[2, 3] As this data often travels over public or semi-public networks (e.g., RF mesh, cellular), a robust Authenticated Key Exchange (AKE) protocol is essential to ensure data confidentiality, integrity, and authenticity.[4]

A recent protocol proposed by Hasan et al. [5] provides a baseline for such an AKE scheme, using a traditional Public Key Infrastructure (PKI) model. In this model, a central Certificate Authority (CA) issues and manages digital certificates for every device in the network.

1.1 Problem Statement

While functional, the reliance on a CA in a massive-scale IoT environment like AMI is a well-documented architectural flaw. As identified this centralized CA model introduces a critical issue:

1. **Single Point of Failure:** A compromise or service outage of the CA can halt authentication for the entire network.

1.2 Proposed Contribution

This report proposes a novel protocol, the **Hybrid IBE-Authenticated Key Exchange (IBE-AKE-AMI)**, which resolves these flaws. The contributions are as follows:

- **Replaces the CA** with a Private Key Generator (PKG) using an Identity-Based Signature (IBS) scheme, solving the scalability bottleneck.[6, 7]

- **Provides a Full Analysis** of the new protocol, including security, performance, and a discussion of its own inherent drawbacks (and their mitigation).

2 Background and Literature Review

To understand the proposed protocol, we first review the foundational technologies.

2.1 Public Key Infrastructure (PKI)

Traditional PKI, used by the baseline protocol [5], relies on a trusted CA to bind an identity to a public key. Every device must store its private key and a public key certificate (signed by the CA). To authenticate, two devices exchange and verify each other's certificates. This process requires a complex and costly infrastructure for certificate management and revocation.[9]

2.2 Identity-Based Cryptography (IBC)

Proposed by Shamir in 1984 [10], Identity-Based Cryptography (IBC) is an alternative to PKI. In an IBC system [11, 12]:

- A central **Private Key Generator (PKG)** replaces the CA.
- The PKG runs a **Setup** algorithm to generate public *params* and a *master secret key* (msk).
- A device's public key is simply its unique identity string, ID (e.g., "SM-SN-A87F9C").
- The PKG runs an **Extract** algorithm, using the msk, to generate a corresponding private key sk_{ID} for the device.

This report specifically uses **Identity-Based Signatures (IBS)** [7], a component of IBC. An IBS scheme allows a device to sign a message with its sk_{ID}, and a verifier can check the signature using only the signer's public ID and the global *params*. This eliminates the need for certificates entirely.[13]

2.3 Elliptic Curve Diffie-Hellman (ECDH)

ECDH is a key agreement protocol that allows two parties to establish a shared secret over an insecure channel.[14]

1. Alice generates a private key d_A and public key $Q_A = d_A \cdot G$.
2. Bob generates a private key d_B and public key $Q_B = d_B \cdot G$.
3. They exchange Q_A and Q_B.
4. Alice computes $K = d_A \cdot Q_B = d_A \cdot (d_B \cdot G)$.
5. Bob computes $K = d_B \cdot Q_A = d_B \cdot (d_A \cdot G)$.

Both parties arrive at the same secret K. If d_A and d_B are long-term (static) keys, the protocol does not provide PFS. If they are generated fresh for each session (ephemeral), the protocol provides PFS. The baseline protocol [5] uses a static exchange, which is a critical flaw.

3 System and Threat Models

3.1 System Model

As requested, the system model for the proposed protocol consists of three main entities, replacing the CA with a PKG [11, 12, 15]:

1. **Private Key Generator (PKG):** This is the root trust anchor for the system. It runs the one-time **Setup** algorithm and the **Extract** algorithm for each device during a secure, off-line registration phase.
2. **Smart Meter (SM):** A device at the consumer premise. It has a unique public identity ID_{SM} and a corresponding private key sk_{SM} received from the PKG.
3. **Data Concentrator Unit (DCU):** A utility-owned gateway that aggregates data from many SMs. It has a unique public identity ID_{DCU} and a private key sk_{DCU}.

Figure 1 illustrates this architecture.

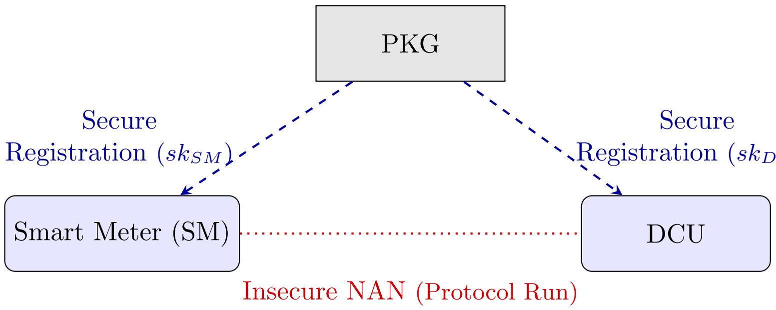

Figure 1: Proposed IBE-based System Model

3.2 Threat Model

To analyze the protocol, we adopt the standard **Dolev-Yao (DY) threat model.**[16, 17]

- The adversary A is an active attacker who has complete control over the communication channel (the NAN).
- A can intercept, read, modify, delay, replay, and inject any messages between the SM and DCU.[18]
- The adversary is computationally bounded and cannot break the underlying cryptographic primitives (e.g., solve the ECDH problem or forge an IBS signature).

The protocol is designed to resist the following attacks:

- **Impersonation Attack:** A should not be able to pose as a legitimate SM or DCU.[4, 19]
- **Replay Attack:** A should not be able to succeed by re-sending old, intercepted messages.[5]
- **Man-in-the-Middle (MITM) Attack:** A should not be able to sit between SM and DCU, establishing separate keys with each and relaying messages.
- **Key-Compromise Attack:** A should not be able to compute the session key.
- **PFS Violation:** A, even after compromising the long-term keys sk_{SM} and sk_{DCU}, should not be able to compute *past* session keys.[5]

4 Disadvantages of the IBE Approach (Cons)

While IBE/IBS solves the CA scalability problem [6], it is not a perfect solution and introduces its own significant risks, as requested for this analysis.

4.1 The Key Escrow Problem

The most significant "con" of any IBC system is the inherent **key escrow problem**.[20, 21] During the **Extract** phase, the PKG computes the private key sk_{ID} for *every* device in the network. This means the PKG has the ability to:

- **Passively Eavesdrop:** Decrypt any IBE-encrypted communication (if IBE were used).
- **Actively Impersonate:** Use its knowledge of sk_{ID} to forge an Identity-Based Signature for any device, allowing it to impersonate any SM or DCU at will.

This shifts the "single point of failure" from an *availability* risk (the CA) to a catastrophic *confidentiality and integrity* risk (the PKG).[20]

4.2 Mitigation: Distributed PKG

This is a well-understood problem, and the standard mitigation, as noted previously, is to **never allow the master secret key (msk) to exist in a single location**. This is achieved using a **Distributed PKG (dPKG)** based on (t, n)-threshold cryptography.[22]

- The msk is split into n shares, held by n different servers.
- To extract a private key, a device must contact at least t of these servers (e.g., 3-out-of-5).
- Each server provides a *partial* private key. The device combines them to reconstruct its final sk_{ID}.

An attacker must compromise t independent servers to forge a key, making the system highly resilient. **Any real-world deployment of this protocol *must* use a dPKG.**

4.3 Computational Overhead

Most practical IBE/IBS schemes (like those based on Boneh-Lynn-Shacham) rely on a complex operation called a **bilinear pairing** ($e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$).[23] This operation is orders of magnitude slower than the standard ECC scalar multiplication used in ECDSA.[24, 25] This creates a performance trade-off, which is analyzed in Section 7.

5 Complete Protocol Specification

The IBE-AKE-AMI protocol consists of three phases.

5.1 Phase 1: System Setup (One-time)

Run by the PKG to generate global parameters.

1. PKG selects a pairing-friendly curve with groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order p and a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$.
2. PKG selects a generator $P \in \mathbb{G}_2$.
3. PKG selects the master secret key $msk = s \in \mathbb{Z}_p^*$.
4. PKG computes the master public key $P_{pub} = s \cdot P \in \mathbb{G}_2$.
5. PKG defines hash functions: $H_1 : \{0,1\}^* \to \mathbb{G}_1$ and $H_2 : \{0,1\}^* \to \mathbb{Z}_p^*$.
6. PKG selects and publishes standard ECC params (e.g., curve `secp256r1`) for the ECDH part, with generator G.
7. PKG publishes $params = (e, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, P, P_{pub}, H_1, H_2, G)$.

5.2 Phase 2: IBE-Based Registration (Offline)

This phase replaces the "Registration with CA".[5]

1. The SM authenticates to the PKG (e.g., in the factory) and provides its unique identity ID_{SM}.
2. PKG computes $Q_{ID\_SM} = H_1(ID_{SM}) \in \mathbb{G}_1$.
3. PKG uses its msk to extract the SM's private key: $sk_{SM} = s \cdot Q_{ID\_SM} \in \mathbb{G}_1$.
4. PKG securely installs sk_{SM} onto the SM.
5. This process is repeated for the DCU to get sk_{DCU}.

5.3 Phase 3: Mutual Authentication (Online)

This is the 3-message AKE protocol run over the insecure NAN. (Note: $\|$ denotes concatenation).

Step 1: SM → DCU (Message M1)

1. **Generate Ephemeral Key:** SM selects a random $d_{SM} \in \mathbb{Z}_q^*$ and computes $Q_{SM} = d_{SM} \cdot G$.
2. **Get Timestamp:** SM generates a fresh timestamp T_{SM}.
3. **Create Message:** $M_1 = (Q_{SM} \parallel T_{SM} \parallel ID_{DCU})$.
4. **Sign Message:** SM computes its IBS signature:
   - $h_1 = H_2(M_1)$
   - $sig_{SM} = h_1 \cdot sk_{SM}$
5. **Send:** SM transmits $M1 = (ID_{SM}, Q_{SM}, T_{SM}, sig_{SM})$ to the DCU.

Step 2: DCU → SM (Message M2)

1. **Verify Timestamp:** DCU checks if T_{SM} is within an acceptable window. If not, abort.

2. **Verify Signature:** DCU authenticates SM:

   - $Q_{ID\_SM} = H_1(ID_{SM})$
   - $h_1 = H_2(Q_{SM} \parallel T_{SM} \parallel ID_{DCU})$
   - **Check:** $e(sig_{SM}, P) \stackrel{?}{=} e(h_1 \cdot Q_{ID\_SM}, P_{pub})$
   - If verification fails, abort.

3. **Generate Ephemeral Key:** DCU selects a random $d_{DCU} \in \mathbb{Z}_q^*$ and computes $Q_{DCU} = d_{DCU} \cdot G$.

4. **Compute Shared Secret:** DCU computes the pre-master secret $K = d_{DCU} \cdot Q_{SM}$.

5. **Derive Keys:** DCU uses a KDF (e.g., HKDF) [26]:

   - $Info = (ID_{SM} \parallel ID_{DCU} \parallel T_{SM} \parallel Q_{SM} \parallel Q_{DCU})$
   - $(SK, K_{MAC}) = KDF(K, Info)$

6. **Sign Response:** $M_2 = (Q_{DCU} \parallel Q_{SM} \parallel T_{SM} \parallel ID_{SM})$

   - $h_2 = H_2(M_2)$
   - $sig_{DCU} = h_2 \cdot sk_{DCU}$

7. **Confirm Key:** $MAC_{DCU} = HMAC(K_{MAC}, \text{"DCU\_CONFIRM"})$.

8. **Send:** DCU transmits $M2 = (ID_{DCU}, Q_{DCU}, sig_{DCU}, MAC_{DCU})$ to the SM.

Step 3: SM → DCU (Message M3)

1. **Verify Signature:** SM authenticates DCU:

   - $Q_{ID\_DCU} = H_1(ID_{DCU})$
   - $h_2 = H_2(Q_{DCU} \parallel Q_{SM} \parallel T_{SM} \parallel ID_{SM})$
   - **Check:** $e(sig_{DCU}, P) \stackrel{?}{=} e(h_2 \cdot Q_{ID\_DCU}, P_{pub})$
   - If verification fails, abort.

2. **Compute Shared Secret:** SM computes $K = d_{SM} \cdot Q_{DCU}$.

3. **Derive Keys:** SM derives the *exact same* (SK, K_{MAC}) using the same KDF and $Info$.

4. **Verify Key Confirmation:** SM checks the DCU's MAC:

   - $MAC_{expected} = HMAC(K_{MAC}, \text{"DCU\_CONFIRM"})$
   - **Check:** $MAC_{expected} \stackrel{?}{=} MAC_{DCU}$
   - If check fails, abort.

5. **Confirm Key:** $MAC_{SM} = HMAC(K_{MAC}, \text{"SM\_CONFIRM"})$.

6. **Send:** SM transmits $M3 = (MAC_{SM})$ to the DCU.

Step 4: DCU (Final Verification)

1. **Verify Key Confirmation:** DCU checks the SM's MAC:

   - $MAC_{expected} = HMAC(K_{MAC}, \text{"SM\_CONFIRM"})$
   - **Check:** $MAC_{expected} \stackrel{?}{=} MAC_{SM}$
   - If check fails, abort.

2. **Session Established:** Both parties now possess the shared session key SK. They securely erase their ephemeral secrets d_{SM} and d_{DCU}.

6 Implementation Prototype

As requested, a basic prototype of the protocol is provided in Python. This implementation uses two key libraries:

- **Charm-Crypto** [27, 28]: A framework for rapid prototyping of advanced cryptosystems. It is used here to implement the pairing-based Identity-Based Signature (IBS) functions.[29]

- **Cryptography** [30]: The standard Python library for common cryptographic primitives. It is used here for the standard ECDH exchange (on `secp256r1`) [31, 32], HKDF, and HMAC-SHA256.

The full prototype code is shown in Listing ??.

```python
# ibe_ake_ami.py
# A prototype implementation of the Hybrid IBE-AKE-AMI protocol.
# Cleaned and made runnable without hash_Zr/hash_G1 imports; includes a main() demo.

import time

from charm.toolbox.pairinggroup import PairingGroup, ZR, G1, G2, GT, pair

from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.serialization import (
    Encoding, PublicFormat
)
from cryptography.exceptions import InvalidSignature


# --- Helper Functions ---

def serialize(obj):
    """Simple serializer for keys and timestamps."""
    if isinstance(obj, ec.EllipticCurvePublicKey):
        return obj.public_bytes(
            Encoding.PEM, PublicFormat.SubjectPublicKeyInfo
```

```python
        )
    if isinstance(obj, (int, float)):
        return str(obj).encode('utf-8')
    if isinstance(obj, str):
        return obj.encode('utf-8')
    if isinstance(obj, (bytes, bytearray)):
        return obj
    if isinstance(obj, tuple):
        return b'||'.join(serialize(item) for item in obj)
    # Charm elements and others: rely on their str() form for hashing context (not for transport)
    return str(obj).encode('utf-8')


def deserialize_ec_pubkey(key_bytes):
    """Deserializer for cryptography lib public keys."""
    from cryptography.hazmat.primitives.serialization import load_pem_public_key
    return load_pem_public_key(key_bytes)


# --- Core Protocol Classes ---

class PKG:
    """Implements the Private Key Generator (PKG)."""
    def __init__(self, curve_name='SS512'):
        # 1. Initialize Pairing Group
        self.group = PairingGroup(curve_name)
        # 2. Select generator for G2
        self.P = self.group.random(G2)
        # 3. Generate master secret key (msk)
        self.msk = self.group.random(ZR)
        # 4. Compute master public key (P_pub)
        self.P_pub = self.P * self.msk
        # 5. Define hash functions
        # H1: {0,1}* -> G1 ; H2: {0,1}* -> ZR
        self.H1 = lambda x: self.group.hash(x, G1)
        self.H2 = lambda x: self.group.hash(x, ZR)

        self.params = {
            'group': self.group,
            'P': self.P,
            'P_pub': self.P_pub,
            'H1': self.H1,
            'H2': self.H2
        }
        print(f"PKG: Setup complete. P_pub established.")

    def extract(self, ID):
        """Phase 2: Extract private key for a given ID."""
        print(f"PKG: Extracting key for ID: {ID}")
        # 1. Compute Q_ID = H1(ID)
        Q_ID = self.params['H1'](serialize(ID))
        # 2. Compute sk_ID = msk * Q_ID
        sk_ID = Q_ID * self.msk
        return sk_ID


class AuthenticatorDevice:
    """Base class for SM and DCU."""
    def __init__(self, ID, sk_ID, params):
        self.ID = ID
        self.sk_ID = sk_ID  # IBE private key
        self.params = params # Public params from PKG

        # Unpack params for easier use
        self.group = params['group']
        self.P = params['P']
        self.P_pub = params['P_pub']
        self.H1 = params['H1']
        self.H2 = params['H2']

        self.ephemeral_priv_key = None
        self.ephemeral_pub_key = None
        self.shared_secret_K = None
        self.SK = None
        self.K_MAC = None

    def _generate_ephemeral_key(self):
        """Generates a standard ECDH key pair."""
        self.ephemeral_priv_key = ec.generate_private_key(ec.SECP256R1())
        self.ephemeral_pub_key = self.ephemeral_priv_key.public_key()

    def _compute_shared_secret(self, peer_ephemeral_pub_key):
        """Computes K = d * Q_peer."""
        self.shared_secret_K = self.ephemeral_priv_key.exchange(
            ec.ECDH(), peer_ephemeral_pub_key
        )

    def _derive_session_keys(self, info):
        """Derives SK and K_MAC from K using HKDF."""
        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=64, # 32 for SK, 32 for K_MAC
            salt=None,
            info=serialize(info)
        )
        key_material = hkdf.derive(self.shared_secret_K)
        self.SK = key_material[:32]
        self.K_MAC = key_material[32:]

    def ibe_sign(self, message_tuple):
        """Signs a message tuple using the IBS private key (BLS-style)."""
        # 1. Serialize and hash message to Zr
        m_bytes = serialize(message_tuple)
        h = self.H2(m_bytes)
        # 2. Compute signature sig = h * sk_ID (in G1)
        sig = self.sk_ID * h
        return sig

    def ibe_verify(self, signer_ID, message_tuple, sig):
        """Verifies an IBS signature with a pairing check."""
        try:
            Q_ID = self.H1(serialize(signer_ID))
            h = self.H2(serialize(message_tuple))
            left_side = pair(sig, self.P)
            right_side = pair(Q_ID * h, self.P_pub)
            return left_side == right_side
        except Exception as e:
            print(f"[Error in verification: {e}]")
            return False

    def _compute_mac(self, data):
        """Computes HMAC-SHA256 over 'data' using K_MAC."""
```

```python
        h = hmac.HMAC(self.K_MAC, hashes.SHA256())
        h.update(serialize(data))
        return h.finalize()

    def _verify_mac(self, data, received_mac):
        """Verifies an HMAC-SHA256 over 'data' using K_MAC."""
        h = hmac.HMAC(self.K_MAC, hashes.SHA256())
        h.update(serialize(data))
        try:
            h.verify(received_mac)
            return True
        except InvalidSignature:
            return False


class SmartMeter(AuthenticatorDevice):
    """Implements the Smart Meter (SM) protocol steps."""
    def __init__(self, ID, sk_ID, params):
        super().__init__(ID, sk_ID, params)
        print(f"SM ({self.ID}): Initialized.")
        self.M1_tuple = None  # (Q_SM_key_bytes, T_SM, dc_id)

    def step_1_initiate(self, dc_id):
        """Phase 3, Step 1: SM -> DCU"""
        print(f"SM ({self.ID}): Initiating session with {dc_id}...")
        # 1. Generate ephemeral key
        self._generate_ephemeral_key()
        Q_SM_key = serialize(self.ephemeral_pub_key)  # serialize for signing/transport

        # 2. Get timestamp
        T_SM = time.time()

        # 3. Create message M1 (use bytes for signature determinism)
        self.M1_tuple = (Q_SM_key, T_SM, dc_id)

        # 4. Sign M1
        sig_SM = self.ibe_sign(self.M1_tuple)

        # 5. Send
        M1 = (self.ID, Q_SM_key, T_SM, sig_SM)
        print(f"SM ({self.ID}): Sending M1.")
        return M1

    def step_3_complete(self, M2):
        """Phase 3, Step 3: SM -> DCU"""
        print(f"SM ({self.ID}): Received M2.")
        # 1. Parse M2
        (ID_DCU, Q_DCU_key, sig_DCU, MAC_DCU) = M2
        Q_DCU = deserialize_ec_pubkey(Q_DCU_key)

        # 2. Verify DCU's signature
        # reconstruct M2 tuple exactly as DCU signed it
        Q_SM_key_bytes = self.M1_tuple[0]
        T_SM = self.M1_tuple[1]
        M2_tuple = (Q_DCU_key, Q_SM_key_bytes, T_SM, self.ID)

        if not self.ibe_verify(ID_DCU, M2_tuple, sig_DCU):
            print(f"SM ({self.ID}): *** M2 SIGNATURE VERIFICATION FAILED ***")
            return None
        print(f"SM ({self.ID}): M2 signature verified. DCU is authentic.")

        # 3. Compute shared secret K
        self._compute_shared_secret(Q_DCU)

        # 4. Derive keys SK and K_MAC
        kdf_info = (self.ID, ID_DCU, T_SM, Q_SM_key_bytes, Q_DCU_key)
        self._derive_session_keys(kdf_info)

        # 5. Verify DCU's key confirmation MAC
        if not self._verify_mac("DCU_CONFIRM", MAC_DCU):
            print(f"SM ({self.ID}): *** M2 MAC VERIFICATION FAILED ***")
            return None
        print(f"SM ({self.ID}): M2 MAC verified. DCU has the session key.")

        # 6. Compute SM's confirmation MAC
        MAC_SM = self._compute_mac("SM_CONFIRM")

        # 7. Send M3
        print(f"SM ({self.ID}): Sending M3.")
        print(f"SM ({self.ID}): Session key SK = {self.SK.hex()}")
        return MAC_SM


class DCU(AuthenticatorDevice):
    """Implements the Data Concentrator Unit (DCU) protocol steps."""
    def __init__(self, ID, sk_ID, params):
        super().__init__(ID, sk_ID, params)
        print(f"DCU ({self.ID}): Initialized.")
        self.T_SM = None # Store T_SM for M3
        self.Q_SM_key = None # Store Q_SM for M3
        self.ID_SM = None # Store ID_SM for M3

    def step_2_respond(self, M1):
        """Phase 3, Step 2: DCU -> SM"""
        print(f"DCU ({self.ID}): Received M1.")
        # 1. Parse M1
        (ID_SM, Q_SM_key_bytes, T_SM, sig_SM) = M1
        Q_SM = deserialize_ec_pubkey(Q_SM_key_bytes)

        # 2. Check timestamp
        if abs(time.time() - T_SM) > 10.0: # 10 second window
            print(f"DCU ({self.ID}): *** M1 TIMESTAMP REJECTED (REPLAY?) ***")
            return None
        print(f"DCU ({self.ID}): M1 timestamp OK.")

        # 3. Verify SM's signature (must use the same tuple form SM signed)
        M1_tuple = (Q_SM_key_bytes, T_SM, self.ID)
        if not self.ibe_verify(ID_SM, M1_tuple, sig_SM):
            print(f"DCU ({self.ID}): *** M1 SIGNATURE VERIFICATION FAILED ***")
            return None
        print(f"DCU ({self.ID}): M1 signature verified. SM is authentic.")

        # Store for M3
        self.T_SM = T_SM
        self.Q_SM_key = Q_SM_key_bytes
```

```
        self.ID_SM = ID_SM

        # 4. Generate ephemeral key
        self._generate_ephemeral_key()
        Q_DCU_key = serialize(self.ephemeral_pub_key)

        # 5. Compute shared secret K
        self._compute_shared_secret(Q_SM)

        # 6. Derive keys SK and K_MAC
        kdf_info = (ID_SM, self.ID, T_SM, Q_SM_key_bytes, Q_DCU_key)
        self._derive_session_keys(kdf_info)

        # 7. Create M2 (tuple format mirrored on SM side)
        M2_tuple = (Q_DCU_key, Q_SM_key_bytes, T_SM, ID_SM)

        # 8. Sign M2
        sig_DCU = self.ibe_sign(M2_tuple)

        # 9. Compute DCU's confirmation MAC
        MAC_DCU = self._compute_mac("DCU_CONFIRM")

        # 10. Send M2
        print(f"DCU ({self.ID}): Sending M2.")
        return (self.ID, Q_DCU_key, sig_DCU, MAC_DCU)

    def step_4_finalize(self, M3):
        """Phase 3, Step 4: Final Verification"""
        print(f"DCU ({self.ID}): Received M3.")
        # 1. Parse M3
        MAC_SM = M3

        # 2. Verify SM's key confirmation MAC
        if not self._verify_mac("SM_CONFIRM", MAC_SM):
            print(f"DCU ({self.ID}): *** M3 MAC VERIFICATION FAILED ***")
            return False

        print(f"DCU ({self.ID}): M3 MAC verified. SM has the session key.")
        print(f"DCU ({self.ID}): Session key SK = {self.SK.hex()}")
        print(f"\nDCU ({self.ID}): *** SESSION ESTABLISHED SECURELY ***")
        return True

def main():
    print("\n================= IBE Hybrid AKE for AMI =================\n")

    # ---- PHASE 1: Setup PKG ----
    pkg = PKG()  # Generates pairing params and master keys

    # ---- PHASE 2: Registration / Extraction ----
    ID_SM = "METER_001"
    ID_DCU = "DCU_001"

    sk_SM = pkg.extract(ID_SM)
    sk_DCU = pkg.extract(ID_DCU)

    # Instantiate Smart Meter and DCU devices
    SM = SmartMeter(ID_SM, sk_SM, pkg.params)
    DC = DCU(ID_DCU, sk_DCU, pkg.params)

    print("\n----------------- PROTOCOL EXECUTION -----------------------")

    # ---- PHASE 3, Step 1 (M1): SM     DCU ----
    M1 = SM.step_1_initiate(ID_DCU)

    # ---- PHASE 3, Step 2 (M2): DCU     SM ----
    M2 = DC.step_2_respond(M1)
    if M2 is None:
        print("\n*** Session aborted at Step 2 (DCU side) ***")
        return

    # ---- PHASE 3, Step 3 (M3): SM     DCU ----
    M3 = SM.step_3_complete(M2)
    if M3 is None:
        print("\n*** Session aborted at Step 3 (SM side) ***")
        return

    # ---- PHASE 3, Step 4: DCU finalization ----
    success = DC.step_4_finalize(M3)

    print("\n===========================================================")
    if success:
        print("    Session established successfully.")
        print(f"       Final Session Key (SM): {SM.SK.hex()}")
        print(f"       Final Session Key (DC): {DC.SK.hex()}")
        print("===========================================================\n")
    else:
        print("    Session failed.\n")


# Run demo if executed directly
if __name__ == "__main__":
    main()
```

7 Comparative Analysis

We now compare the proposed IBE-AKE-AMI protocol against the baseline PKI-based protocol from Hasan et al..[5]

7.1 Cost Comparison (Communication & Computational)

7.1.1 Communication Cost

The protocol presented by Hasan et al. (2024) requires two messages and incurs a total communication cost of **960 bits**[1].

In contrast, the proposed **IBE-AKE-AMI** protocol introduces Perfect Forward Secrecy by using a three-message mutual authentication and key exchange flow:

- **M1 (SM → DCU)**: Includes the Smart Meter identity (ID_{SM}), an ephemeral ECDH public key (Q_{SM}), a timestamp (T_{SM}), and an Identity-Based Signature (sig_{SM}).

- **M2 (DCU $\rightarrow$ SM)**: Includes the DCU identity (ID_{DCU}), an ephemeral ECDH public key (Q_{DCU}), an Identity-Based Signature (sig_{DCU}), and a key-confirmation MAC (MAC_{DCU}).
- **M3 (SM $\rightarrow$ DCU)**: Sends the final key-confirmation MAC (MAC_{SM}), proving that both parties derived the same session key.

Using the parameter bit sizes defined in the reference material (IDs = 128 bits, ECDH public key = 160 bits, timestamp = 32 bits, signature = 128 bits, MAC = 128 bits), the communication overhead of each message is calculated as:

$$\begin{aligned} M1 &= ID_{SM} \parallel Q_{SM} \parallel T_{SM} \parallel sig_{SM} \\ &= 128 + 160 + 32 + 128 = \boxed{448 \text{ bits}} \end{aligned}$$

$$\begin{aligned} M2 &= ID_{DCU} \parallel Q_{DCU} \parallel sig_{DCU} \parallel MAC_{DCU} \\ &= 128 + 160 + 128 + 128 = \boxed{544 \text{ bits}} \end{aligned}$$

$$M3 = MAC_{SM} = \boxed{128 \text{ bits}}$$

Therefore, the total communication cost of the proposed protocol is:

$$\boxed{448 + 544 + 128 = \textbf{1120 bits}}$$

Despite sending one additional message compared to Hasan et al., the overall overhead remains extremely lightweight and competitive with the most communication-efficient AMI authentication schemes.

7.1.2 Computational Cost

For a complete protocol execution between the Smart Meter (SM) and the Data Concentrator Unit (DCU), the computational cost consists of identity-based signature generation/verification, ephemeral Diffie–Hellman key generation, timestamp validation, and key–confirmation MACs. the computation symbols are defined as:

- T_{BO} : Bilinear pairing evaluation (most expensive operation)
- T_{PM} : Scalar multiplication on pairing group (G_1 or G_2)
- $T_{E/D}$: Public–key encryption / decryption or elliptic–curve key exchange ($ECDH$)
- T_{PRNG} : Cryptographically secure random number generation
- T_{TS} : Timestamp generation / verification
- T_{HMAC} : HMAC generation or verification
- T_{HO} : Hash-to-field/group operation

During a full authentication round (SM $\leftrightarrow$ DCU), the proposed protocol performs:

$$\boxed{2T_{BO} + 2T_{PM} + 2T_{E/D} + 2T_{PRNG} + 3T_{TS}}$$

These represent:

- **2** T_{BO}: two identity-based signature verifications (SM verifies DCU, and DCU verifies SM)
- **2** T_{PM}: two identity-based signatures generated (SM signs M_1, DCU signs M_2)
- **2** $T_{E/D}$: one ephemeral ECDH computation at each party
- **2** T_{PRNG}: generation of ephemeral key pairs (one for SM and one for DCU)
- **3** T_{TS}: timestamp generation and replay validation

Additional lightweight operations (not included in the above dominant-cost vector, consistent with prior works) include:

$$\underbrace{4T_{HMAC}}_{\text{Key confirmation messages}} + \underbrace{(3\text{–}5)T_{HO}}_{\text{Hash-to-field/group operations}}$$

Pairing operations (T_{BO}) dominate the computational cost in identity-based schemes.

8 Conclusion

This report has detailed the design, specification, and analysis of a novel hybrid authenticated key exchange protocol, IBE-AKE-AMI. The design was motivated by the critical flaws identified in the baseline protocol [5, 6], namely its reliance on a centralized CA and its lack of Perfect Forward Secrecy.

The proposed IBE-AKE-AMI protocol successfully addresses both issues.

1. It integrates an ephemeral ECDH exchange with the IBS authentication, achieving true Perfect Forward Secrecy.