The Design and Formal Analysis of the Hierarchal Key Distribution System

John G. Underhill

Quantum Resistant Cryptographic Solutions Corporation

Abstract. The Hierarchical Key Distribution System (HKDS) is a symmetric key establishment mechanism that derives all operational keys from two server held root secrets and per device identifiers using the SHAKE and KMAC functions. HKDS provides token based key diversification followed by a per transaction cache of one time encryption and authentication keys. In this paper we present a provable security analysis of HKDS in a standard game based framework. We formalize confidentiality, message authenticity, token authenticity, forward secrecy under erasure, and post compromise security through dedicated security experiments and adversarial oracles that model both client corruption and server assisted key evolution.

Our main results show that HKDS achieves IND CPA confidentiality and UF CMA message authenticity, and that token forgery reduces to the PRF security of KMAC under the embedded device key. We further prove that the protocol provides forward secrecy and post compromise security when erasure is correctly implemented and the server token key remains secret. All bounds are derived under the assumption that SHAKE and KMAC behave as pseudo-random functions on their absorbed domains and that master keys are generated uniformly at random. These results formalize and justify the security claims made in the HKDS design and specification.

1 Introduction

Distributed symmetric key systems play a critical role in payment networks, point of sale devices, and other embedded environments where large numbers of terminals must communicate securely with a central authority. The dominant construction used in commercial financial systems for more than three decades has been the Derived Unique Key Per Transaction (DUKPT) mechanism, which produces a sequence of transaction specific keys from an embedded device key and a server side base derivation key. Although DUKPT has enabled the deployment of large scale symmetric key systems, it provides limited security properties by modern standards and its design does not extend naturally to higher security levels or post compromise recovery.

The Hierarchical Key Distribution System (HKDS) is a symmetric key establishment protocol designed to address these limitations. HKDS replaces the one way key chain structure of DUKPT with a two tier hierarchy based on two independent server held secret roots. Each client device contains a unique embedded device key derived from a base derivation key and device specific identifiers, and obtains an ephemeral token from the server that is authenticated and encrypted under independent inputs to SHAKE and KMAC. The client combines the token with its embedded device key to derive a cache of one time encryption and authentication keys that are consumed sequentially. This structure provides efficient key evolution, high performance, and cryptographic separation between key generation domains.

This paper provides the first formal provable security analysis of HKDS. We develop a complete game based model for the protocol, including adversarial oracles for token generation, message encryption, corruption of client state, and challenge queries for key indistinguishability. We formalize the security goals of HKDS, including confidentiality of encrypted messages, unforgeability of authenticated ciphertexts, authenticity of server generated tokens, forward secrecy under an explicit erasure model, and post compromise security based on the recovery provided by fresh token exchange. Using these definitions we construct reductions that bound the advantage of any adversary in terms of the pseudo-randomness of SHAKE and KMAC on their absorbed domains. Our analysis also considers partial compromise of the server side master keys, showing how the independence of the two root secrets provides a degree of defense in depth.

This work complements the normative HKDS specification and the implementation analysis that describes the engineering and performance characteristics of the protocol. Whereas those documents emphasize operational detail and implementation correctness, the present paper establishes that the security properties of HKDS can be derived from well defined assumptions under a standard formal model. The remainder of this paper is organized as follows. Section 2 gives an engineering level description of HKDS that consolidates the operational details required for the formal analysis. Section 3 introduces notation and assumptions. Section 4 defines the adversarial model and the security experiments. Section 5 proves structural properties of HKDS, including domain separation and deterministic reconstruction. Section 6 presents the provable security results. Section 7 provides concrete security bounds. Section 8 discusses implementation considerations. We conclude in Section 9.

2 Engineering Description of the HKDS Protocol

2.1 High Level Overview

The HKDS protocol is a symmetric key distribution system designed for devices that communicate with a centralized transaction processing server. Each client device is provisioned with a unique embedded device key and a device identity. The server maintains a master derivation key that contains two independent secret roots. By combining

server generated entropy with device specific information, HKDS produces a deterministic sequence of one time encryption and authentication keys that can be reconstructed by the server without maintaining per device state.
An HKDS interaction consists of three major components. First, the client requests an authenticated and encrypted token from the server. Second, the client decrypts and authenticates this token to derive an ephemeral secret that, together with its embedded device key, seeds the pseudo-random function that generates a cache of single use transaction keys. Finally, the client uses these keys to encrypt and authenticate messages that are sent to the server. The server reproduces the same transaction keys deterministically to verify and decrypt incoming ciphertexts. All key derivation is based on SHAKE and KMAC in their standard forms.

2.2 State, Keys, and Identifiers

Master Derivation Key. The server holds a master derivation key

$$\mathsf{MDK} = (\mathsf{BDK}, \mathsf{STK}, \mathsf{KID}),$$

where the base derivation key BDK and the secret token key STK are independent uniformly random byte strings of length $n \in \{16, 32, 64\}$ depending on the desired security level. The key identifier KID is a fixed 4 byte string used to select the appropriate master key when processing client requests.

Device Identity and Embedded Device Key. Each client possesses a device identity DID consisting of protocol flags, a manufacturer identifier, and a unique device identifier. The embedded device key

$$\mathsf{EDK} = \mathsf{SHAKE}(\mathsf{DID} \parallel \mathsf{BDK})$$

is computed at provisioning time and stored in a secure module on the device.

Transaction Counter and Key Serial Number. The client maintains a 32 bit transaction counter KTC that increments once for every transaction key consumed. The key serial number $\mathsf{KSN} = \mathsf{DID} \parallel \mathsf{KTC}$ is included with each client request and is used by the server, together with the master key identifier KID, to reconstruct the client state.

Customization Strings. Two customization strings are used to bind token and message authentication operations to a given device and epoch:

$$\mathsf{CTOK} = \mathsf{TRC} \parallel \mathsf{FN} \parallel \mathsf{DID}, \qquad \mathsf{TMS} = \mathsf{KSN} \parallel \mathsf{MACName},$$

where TRC is the token request counter derived from KTC and the cache size, FN is the implementation formal name, and $\mathsf{MACName}$ identifies the KMAC variant in use.

Token and Encrypted Token. The server generates a per epoch token

$$\mathsf{TOK} = \mathsf{SHAKE}(\mathsf{CTOK} \parallel \mathsf{STK}),$$

and encrypts it using a one time pad derived from the embedded device key:

$$\mathsf{Kpad} = \mathsf{SHAKE}(\mathsf{CTOK} \parallel \mathsf{EDK}), \qquad \mathsf{ETOK} = \mathsf{TOK} \oplus \mathsf{Kpad}.$$

Transaction Key Cache. After authenticating and decrypting ETOK, the client derives its transaction key cache:

$$\mathsf{TKC} = \mathsf{SHAKE}(\mathsf{TOK} \parallel \mathsf{EDK}),$$

returning blocks of pseudo-random output from which individual encryption and authentication keys are extracted sequentially.

2.3 Token Request and Response Flow

The token exchange begins when the client constructs and transmits a token request packet containing its KSN. The server performs the following steps using the master key identified by the KID field:

1. Recompute the embedded device key EDK from (DID, BDK).
2. Compute the customization string CTOK from the token request counter, formal name, and device identity.
3. Derive the ephemeral token TOK from (CTOK, STK).
4. Derive the token pad Kpad from (CTOK, EDK).
5. Form the encrypted token $\mathsf{ETOK} = \mathsf{TOK} \oplus \mathsf{Kpad}$.
6. Compute the token authentication tag

$$\mathsf{TAG}_t = \mathsf{KMAC}_{\mathsf{EDK}}(\mathsf{ETOK}; \mathsf{custom} = \mathsf{TMS}),$$

and append it to ETOK.

Upon receiving the packet, the client:

1. Computes $\mathsf{TAG}_t^* = \mathsf{KMAC}_{\mathsf{EDK}}(\mathsf{ETOK}; \mathsf{custom} = \mathsf{TMS})$.
2. Verifies $\mathsf{TAG}_t^* = \mathsf{TAG}_t$; if verification fails, it rejects the token.
3. Reconstructs CTOK and recomputes Kpad.
4. Decrypts the token as $\mathsf{TOK} = \mathsf{ETOK} \oplus \mathsf{Kpad}$.

All intermediate token values, including Kpad and the decrypted TOK, are erased after use.

2.4 Transaction Key Cache Generation

Once the token is authenticated and decrypted, the client initializes the transaction key cache by invoking SHAKE on the concatenation (TOK ‖ EDK). The resulting pseudo-random stream is partitioned into 16 byte blocks, each of which forms a single use transaction key:

$$\mathsf{TKC} = (K_1, K_2, \ldots, K_m).$$

The number of keys m depends on the security level and a cache multiplier. The client consumes each key exactly once. After a key is used, it is overwritten in memory. When the cache is exhausted, the client increments its token request counter, constructs a new token request, and repeats the token exchange.

2.5 Message Protection

HKDS uses an encrypt then authenticate construction based on the sequential use of transaction keys. To encrypt and authenticate a message M, the client proceeds as follows:

1. Extract the next encryption key K_{enc} from the cache and increment KTC.
2. Form the ciphertext

$$C = M \oplus K_{\mathsf{enc}}.$$

3. Extract the next authentication key K_{mac} from the cache.

4. Compute the authentication tag

$$\mathrm{TAG} = \mathrm{KMAC}_{K_{\mathrm{mac}}}\big(C;\ \mathrm{custom} = A\big),$$

   where A is optional associated data supplied as the KMAC customization string.

5. Send $(C, \mathsf{TAG}, \mathsf{KSN})$ to the server.

Both K_{enc} and K_{mac} are erased immediately after use.

2.6 Server Side Reconstruction

Upon receiving a ciphertext and key serial number from the client, the server deterministically reconstructs the corresponding transaction keys without maintaining per device state. Using KSN, it performs:

1. Identify the correct master key using KID.
2. Recompute $\mathsf{EDK} = \mathsf{SHAKE}(\mathsf{DID} \parallel \mathsf{BDK})$.
3. Reconstruct CTOK from the token request counter, formal name, and device identity.
4. Derive $\mathsf{TOK} = \mathsf{SHAKE}(\mathsf{CTOK} \parallel \mathsf{STK})$.
5. Recompute $\mathsf{TKC} = \mathsf{SHAKE}(\mathsf{TOK} \parallel \mathsf{EDK})$ until reaching the index corresponding to KTC.
6. Extract K_{enc} and K_{mac} and verify:

$$\mathsf{TAG}^* = \mathsf{KMAC}_{K_{\mathsf{mac}}}(C \parallel A).$$

7. If verification succeeds, decrypt $M = C \oplus K_{\mathsf{enc}}$.

Because all operations are deterministic functions of $(\mathsf{MDK}, \mathsf{KSN}, \mathsf{DID})$, the server does not store any long term per device state, and the client and server remain synchronized as long as counters evolve monotonically.

2.7 Reference Pseudo-code Definitions

This section presents reference pseudo-code for the core HKDS client side and server side procedures. Each algorithm is a direct transcription of the reference C implementation and preserves the same control flow, buffer layouts, and constant usage. Variable names and buffer sizes match the code so that each step can be traced back line by line.

2.7.1 Client Side Procedures

The client maintains a per device state that includes the key serial number `ksn`, the embedded device key `edk`, the current transaction key cache `tkc`, a cache index and emptiness flag, and the PRF rate. Initialization copies the static configuration, derives `edk` from the master derivation key, and sets counters from the `ksn`.

Algorithm 1 HKDS_CLIENT_INITIALIZE_STATE

Require: Client state `state`, master key `mdk`, key serial number `ksn`

```
state.ksn ← ksn
state.cache_empty ← true
state.index ← 0
state.rate ← HKDS_PRF_RATE
state.mdk ← mdk
// Generate embedded device key EDK = SHAKE(DID ‖ BDK)
did ← first HKDS_DID_SIZE bytes of ksn
dkey ← zero buffer of size HKDS_BDK_SIZE + HKDS_DID_SIZE
copy did into dkey[0..HKDS_DID_SIZE-1]
copy mdk.bdk into dkey[HKDS_DID_SIZE..]
if then HKDS_SHAKE_128 is defined
    state.edk ← SHAKE128(dkey, HKDS_EDK_SIZE)
else if then HKDS_SHAKE_256 is defined
    state.edk ← SHAKE256(dkey, HKDS_EDK_SIZE)
else
    state.edk ← SHAKE512(dkey, HKDS_EDK_SIZE)
end if
```

The client derives individual transaction keys for each message from the cached key material. The cache index is computed from the counter portion of the `ksn`, and each generated key is erased after use. When the final cache entry is consumed, the `cache_empty` flag is set.

Algorithm 2 HKDS_CLIENT_GENERATE_TRANSACTION_KEY

Require: Client state `state`, output buffer `tkey` of length `tkeylen`

```
// Compute cache index from counter in KSN modulo cache size
count ← BE8TO32(state.ksn[HKDS_DID_SIZE..])
index ← count mod HKDS_CACHE_SIZE
// Copy selected cache entry into tkey
copy state.tkc[index][0..tkeylen-1] to tkey[0..tkeylen-1]
// Erase cache entry after use
set state.tkc[index] to all zeros
// Increment big endian counter in KSN
INC_BE32 at state.ksn[HKDS_DID_SIZE..]
if then index = HKDS_CACHE_SIZE −1
    state.cache_empty ← true
end if
```

Token decryption reconstructs the same PRF input that the server used for token encryption, regenerates the keystream that hides the token, and verifies the attached token MAC. On success, the recovered token is written to the client state.

Algorithm 3 HKDS_CLIENT_DECRYPT_TOKEN

Require: Client state `state`, received encrypted token `etok`
Ensure: Returns `true` if the token and MAC verify, else `false`

```
res ← false
ctok ← zero buffer of size HKDS_CTOK_SIZE
tms ← zero buffer of size HKDS_TMS_SIZE
tmpk ← zero buffer of size HKDS_CTOK_SIZE + HKDS_EDK_SIZE
tok ← zero buffer of size HKDS_STK_SIZE
// Rebuild CTOK from client state
HKDS_CLIENT_GET_CTOK(state, ctok)
// Copy ctok and edk into PRF key
copy ctok into tmpk[0..HKDS_CTOK_SIZE-1]
copy state.edk into tmpk[HKDS_CTOK_SIZE..]
// Regenerate keystream used to encrypt the token
if thenHKDS_SHAKE_128 is defined
    tok ← SHAKE128(tmpk, HKDS_STK_SIZE)
else if thenHKDS_SHAKE_256 is defined
    tok ← SHAKE256(tmpk, HKDS_STK_SIZE)
else
    tok ← SHAKE512(tmpk, HKDS_STK_SIZE)
end if
// Form token MAC string TMS from KSN and MAC name
HKDS_CLIENT_GET_TMS(state.ksn, tms)
// Verify token MAC under EDK
code ← zero buffer of size HKDS_TAG_SIZE
if thenHKDS_SHAKE_128 is defined
    code ← KMAC128(etok[0..HKDS_STK_SIZE-1], state.edk, tms)
else if thenHKDS_SHAKE_256 is defined
    code ← KMAC256(etok[0..HKDS_STK_SIZE-1], state.edk, tms)
else
    code ← KMAC512(etok[0..HKDS_STK_SIZE-1], state.edk, tms)
end if
if thenVERIFY_EQUAL(code, etok[HKDS_STK_SIZE..], HKDS_TAG_SIZE) = 0
    // MAC is valid, decrypt the token
    for i ← 0 to HKDS_STK_SIZE −1 do
        state.tok[i] ← etok[i] ⊕ tok[i]
    end for
    res ← true
end if
return res
```

Once a valid token is installed, the client expands it into a transaction key cache using SHAKE seeded with the concatenation of `tok` and `edk`. The resulting stream is sliced into fixed size message keys.

Algorithm 4 HKDS_CLIENT_GENERATE_CACHE

Require: Client state `state`

1: `skey` ← zero buffer of size `HKDS_CACHE_SIZE` × `HKDS_MESSAGE_SIZE`
2: `tmpk` ← zero buffer of size `HKDS_STK_SIZE + HKDS_EDK_SIZE`
3: // Copy token and EDK into PRF key
4: copy `state.tok` into `tmpk[0..HKDS_STK_SIZE-1]`
5: copy `state.edk` into `tmpk[HKDS_STK_SIZE..]`
6: // Expand to cache worth of key material
7: **if** **then**HKDS_SHAKE_128 is defined
8: `skey` ← SHAKE128(`tmpk`, `sizeof(skey)`)
9: **else if** **then**HKDS_SHAKE_256 is defined
10: `skey` ← SHAKE256(`tmpk`, `sizeof(skey)`)
11: **else**
12: `skey` ← SHAKE512(`tmpk`, `sizeof(skey)`)
13: **end if**
14: // Fill cache entries with consecutive message keys
15: **for** $i \leftarrow 0$ to `HKDS_CACHE_SIZE` -1 **do**
16: copy `skey[i × HKDS_MESSAGE_SIZE .. (i+1) × HKDS_MESSAGE_SIZE −1]` into `state.tkc[i]`
17: **end for**
18: `state.cache_empty` ← `false`

Authenticated encryption on the client side consumes one cache entry for the encryption key and a second cache entry for the MAC key. The function returns `false` if either key cannot be obtained.

Algorithm 5 HKDS_CLIENT_ENCRYPT_AUTHENTICATE_MESSAGE

Require: Client state `state`, plaintext `input` of size `HKDS_MESSAGE_SIZE`
Require: Output buffer `ciphertext` of size `HKDS_MESSAGE_SIZE + HKDS_TAG_SIZE`
Require: Associated data buffer `data` and length `datalen`
Ensure: Returns `true` on success, `false` if cache is empty

```
res ← false
// Derive encryption and MAC key material from cache
if then state.cache_empty = false
    dkey ← zero buffer of size 2 × HKDS_MESSAGE_SIZE
    HKDS_CLIENT_GENERATE_TRANSACTION_KEY(state, dkey, HKDS_MESSAGE_SIZE)
    // Encrypt using the first message-key block
    for i ← 0 to HKDS_MESSAGE_SIZE −1 do
        ciphertext[i] ← input[i] ⊕ dkey[i]
    end for
    res ← true
end if
if then state.cache_empty = false
    // Obtain MAC key from a second cache entry
    HKDS_CLIENT_GENERATE_TRANSACTION_KEY(state, dkey, HKDS_MESSAGE_SIZE)
    // Compute tag over ciphertext and associated data
    if then HKDS_SHAKE_128 is defined
        KMAC128(
     output ← ciphertext + HKDS_MESSAGE_SIZE,
     tag length ← HKDS_TAG_SIZE,
     message ← ciphertext, message length ← HKDS_MESSAGE_SIZE,
     key ← dkey, key length ← HKDS_MESSAGE_SIZE,
     customization ← data, customization length ← datalen)
    else if then HKDS_SHAKE_256 is defined
        same call using KMAC256
    else
        same call using KMAC512
    end if
else
    res ← false
end if
return res
```

2.7.2 Server Side Procedures

The server maintains a master derivation key `mdk` containing the base derivation key `bdk`, the server token key `stk`, and the key identifier `kid`. For each device, it holds a per client state with the `ksn`, a local copy of the master key pointer, and counters. Initialization derives these values directly from the inputs.

Algorithm 6 HKDS_SERVER_GENERATE_MDK

Require: RNG callback `rng_generate`, master key struct `mdk`, key identifier `kid`

```
tmpr ← buffer of size HKDS_BDK_SIZE + HKDS_STK_SIZE
// Fill combined buffer with randomness
rng_generate(tmpr, sizeof(tmpr))
copy tmpr[0..HKDS_BDK_SIZE-1] into mdk.bdk
copy tmpr[HKDS_BDK_SIZE..] into mdk.stk
copy kid into mdk.kid
```

Algorithm 7 HKDS_SERVER_INITIALIZE_STATE

Require: Server state `state`, master key `mdk`, key serial number `ksn`

```
copy ksn into state.ksn
state.mdk ← mdk
state.count ← BE8TO32(ksn[HKDS_DID_SIZE..])
state.rate ← HKDS_PRF_RATE
```

The server derives per device embedded keys `edk` from the base derivation key and device identifier, and constructs tokens by hashing the custom token string together with the server token key `stk`.

Algorithm 8 HKDS_SERVER_GENERATE_EDK

Require: Base derivation key `bdk`, device identifier `did`
Ensure: Embedded device key `edk`

```
dkey ← zero buffer of size HKDS_BDK_SIZE + HKDS_DID_SIZE
copy did into dkey[0..HKDS_DID_SIZE-1]
copy bdk into dkey[HKDS_DID_SIZE..]
if then HKDS_SHAKE_128 is defined
    edk ← SHAKE128(dkey, HKDS_EDK_SIZE)
else if then HKDS_SHAKE_256 is defined
    edk ← SHAKE256(dkey, HKDS_EDK_SIZE)
else
    edk ← SHAKE512(dkey, HKDS_EDK_SIZE)
end if
```

Algorithm 9 HKDS_SERVER_GET_CTOK and HKDS_SERVER_GET_TMS

Require: Server state `state`, key serial number `ksn`
Ensure: Custom token string `ctok`, token MAC string `tms`

```
// Custom token string CTOK
tkc ← BE8TO32(state.ksn[HKDS_DID_SIZE..])/HKDS_CACHE_SIZE
BE32TO8(ctok[0..HKDS_TKC_SIZE-1], tkc)
copy hkds_formal_name into ctok at offset HKDS_TKC_SIZE
copy state.ksn[0..HKDS_DID_SIZE-1] into ctok[HKDS_TKC_SIZE+HKDS_NAME_SIZE..]
// Token MAC string TMS
copy ksn into tms[0..HKDS_KSN_SIZE-1]
copy hkds_mac_name into tms[HKDS_KSN_SIZE..]
```

The server side transaction key generator mirrors the client side cache expansion, regenerating the same per message keys deterministically from the base data.

Algorithm 10 HKDS_SERVER_GENERATE_TRANSACTION_KEY

Require: Server state `state`, output buffer `tkey` of length `tkeylen`

```
ctok ← zero buffer of size HKDS_CTOK_SIZE
did ← zero buffer of size HKDS_DID_SIZE
edk ← zero buffer of size HKDS_EDK_SIZE
skey ← zero buffer of size HKDS_CACHE_SIZE × HKDS_MESSAGE_SIZE
tok ← zero buffer of size HKDS_STK_SIZE
tmpk ← zero buffer of size HKDS_STK_SIZE + HKDS_EDK_SIZE
// Compute cache index from KSN
index ← BE8TO32(state.ksn[HKDS_DID_SIZE..]) mod HKDS_CACHE_SIZE
copy state.ksn[0..HKDS_DID_SIZE-1] into did
// Regenerate EDK, CTOK, and TOK
HKDS_SERVER_GENERATE_EDK(state.mdk.bdk, did, edk)
HKDS_SERVER_GET_CTOK(state, ctok)
HKDS_SERVER_GENERATE_TOKEN(state.mdk.stk, ctok, tok)
// Seed SHAKE with token and embedded key
copy tok into tmpk[0..HKDS_STK_SIZE-1]
copy edk into tmpk[HKDS_STK_SIZE..]
if  then HKDS_SHAKE_128 is defined
    skey ← SHAKE128(tmpk, sizeof(skey))
else if  then HKDS_SHAKE_256 is defined
    skey ← SHAKE256(tmpk, sizeof(skey))
else
    skey ← SHAKE512(tmpk, sizeof(skey))
end if
// Select the same cache entry as the client
offset ← index × HKDS_MESSAGE_SIZE
copy skey[offset..offset + tkeylen - 1] into tkey
```

Token encryption on the server side reconstructs EDK and CTOK, generates the token TOK from STK and CTOK, encrypts it under a SHAKE derived keystream, and appends a KMAC tag under EDK.

Algorithm 11 HKDS_SERVER_ENCRYPT_TOKEN

Require: Server state `state`, output buffer `etok` of size `HKDS_STK_SIZE + HKDS_TAG_SIZE`

```
ctok ← zero buffer of size HKDS_CTOK_SIZE
did ← zero buffer of size HKDS_DID_SIZE
edk ← zero buffer of size HKDS_EDK_SIZE
tms ← zero buffer of size HKDS_TMS_SIZE
tmpk ← zero buffer of size HKDS_CTOK_SIZE + HKDS_EDK_SIZE
tok ← zero buffer of size HKDS_STK_SIZE
copy state.ksn[0..HKDS_DID_SIZE-1] into did
HKDS_SERVER_GENERATE_EDK(state.mdk.bdk, did, edk)
HKDS_SERVER_GET_CTOK(state, ctok)
HKDS_SERVER_GENERATE_TOKEN(state.mdk.stk, ctok, tok)
copy ctok into tmpk[0..HKDS_CTOK_SIZE-1]
copy edk into tmpk[HKDS_CTOK_SIZE..]
if then HKDS_SHAKE_128 is defined
    etok[0..HKDS_STK_SIZE-1] ← SHAKE128(tmpk, HKDS_STK_SIZE)
else if then HKDS_SHAKE_256 is defined
    etok[0..HKDS_STK_SIZE-1] ← SHAKE256(tmpk, HKDS_STK_SIZE)
else
    etok[0..HKDS_STK_SIZE-1] ← SHAKE512(tmpk, HKDS_STK_SIZE)
end if
// Encrypt token by xor with keystream
for i ← 0 to HKDS_STK_SIZE −1 do
    etok[i] ← etok[i] ⊕ tok[i]
end for
HKDS_SERVER_GET_TMS(state.ksn, tms)
if then HKDS_SHAKE_128 is defined
    KMAC128(etok + HKDS_STK_SIZE, HKDS_TAG_SIZE,
     message ← etok, length ← HKDS_STK_SIZE,
     key ← edk, key length ← HKDS_EDK_SIZE,
     customization ← tms, length ← HKDS_TMS_SIZE)
else if then HKDS_SHAKE_256 is defined
    same call using KMAC256
else
    same call using KMAC512
end if
```

Finally, server side decryption regenerates the same transaction key pair as the client, verifies the MAC before decryption, and only then recovers the plaintext.

Algorithm 12 HKDS_SERVER_DECRYPT_VERIFY_MESSAGE

Require: Server state `state`, ciphertext block `ciphertext`
Require: Output plaintext buffer `plaintext`, associated data `data`, length `datalen`
Ensure: Returns `true` if verification and decryption succeed, else `false`

```
res ← false
code ← zero buffer of size HKDS_TAG_SIZE
dkey ← zero buffer of size 2 × HKDS_MESSAGE_SIZE
// Regenerate transaction key material
HKDS_SERVER_GENERATE_TRANSACTION_KEY(state, dkey, sizeof(dkey))
// Compute MAC over received ciphertext using second half of dkey
if thenHKDS_SHAKE_128 is defined
    code ← KMAC128(ciphertext, HKDS_MESSAGE_SIZE, dkey + HKDS_MESSAGE_SIZE, HKDS_MESSAGE_SIZE, data, datalen)
else if thenHKDS_SHAKE_256 is defined
    code ← KMAC256(ciphertext, HKDS_MESSAGE_SIZE, dkey + HKDS_MESSAGE_SIZE, HKDS_MESSAGE_SIZE, data, datalen)
else
    code ← KMAC512(ciphertext, HKDS_MESSAGE_SIZE, dkey + HKDS_MESSAGE_SIZE, HKDS_MESSAGE_SIZE, data, datalen)
end if
if thenVERIFY_EQUAL(code, ciphertext + HKDS_MESSAGE_SIZE, HKDS_TAG_SIZE) = 0
    // Only on successful MAC check, decrypt the message
    for i ← 0 to HKDS_MESSAGE_SIZE −1 do
        plaintext[i] ← ciphertext[i] ⊕ dkey[i]
    end for
    res ← true
end if
return res
```

3 Preliminaries and Assumptions

3.1 Notation and Conventions

We write $x \parallel y$ for the concatenation of byte strings x and y, and $x \oplus y$ for bitwise exclusive OR on equal length strings. For a string X, we write $|X|$ for its length in bytes. All bit and byte strings are interpreted as elements of $\{0,1\}^*$ unless stated otherwise. A function F that outputs an unbounded stream of pseudo-random bytes is assumed to be truncated to the number of bytes required by the construction in which it is used.

We summarize the principal symbols used throughout this paper:

- BDK: Base derivation key held by the server, of length $n \in \{16, 32, 64\}$ bytes.
- STK: Secret token key held by the server, independent of BDK and of the same length.
- KID: Four byte key identifier selecting the appropriate master derivation key.
- DID: Device identity, consisting of protocol flags, a manufacturer identifier, and a unique device identifier.
- EDK: Embedded device key, defined as SHAKE(DID $\parallel$ BDK).
- KTC: Thirty two bit transaction counter incremented per key use.

- KSN:
- KSN: Key serial number, DID ‖ KTC.
- TRC: Token request counter derived from the cache size and KTC.
- FN: Implementation formal name identifying the HKDS variant.
- CTOK: Token customization string, TRC ‖ FN ‖ DID.
- TMS: Token MAC customization string, KSN ‖ MACName.
- TOK: Ephemeral token, SHAKE(CTOK ‖ STK).
- Kpad: Token pad, SHAKE(CTOK ‖ EDK).
- ETOK: Encrypted token, TOK $\oplus$ Kpad.
- TKC: Transaction key cache derived as SHAKE(TOK ‖ EDK).
- $K_{\mathsf{enc}}, K_{\mathsf{mac}}$: Consecutive single use keys drawn from TKC.

The message size for HKDS protected payloads is fixed at 16 bytes, and authentication tags are 16 bytes. These parameters match the normative HKDS specification.

3.2 HKDS Parameter Sizes

The HKDS protocol uses a number of fixed-width identifiers, keys, and customization strings. The parameter size table summarizes these core parameters as defined in the HKDS specification and implementation headers. All subsequent pseudo-code and formal definitions rely on these symbolic constants.

Table 1: HKDS Parameter and Identifier Sizes

Parameter	Size (bytes)
Device Identifier (DID)	`96 bits`
Key Serial Number (KSN)	`128 bits`
Base Derivation Key (BDK)	`128/256/512 bits`
Server Token Key (STK)	`128/256/512 bits`
Embedded Device Key (EDK)	`128/256/512 bits`
Token (ETOK)	`256/384/640 bits`
Token MAC Tag	`128/256/512 bits`
CTOK Customization String	`184 bits`
TMS Customization String	`184 bits`
Message Block Size	`128 bits`
Transaction Cache Entries	`42/34/18`
PRF Rate	`168/136/72`

These definitions ensure consistent interpretation of HKDS messages and state across both the client and server roles. The security model and provable bounds derived later in the paper assume these constants match the implementation exactly.

3.3 Cryptographic Primitives

HKDS relies exclusively on the SHAKE and KMAC functions defined in the SHA-3 standard. We summarize their usage and security assumptions below.

SHAKE. We treat $\mathsf{SHAKE}(\cdot)$ as a pseudo-random function on the absorbed input. For any fixed input string X, the function

$$Y \leftarrow \mathsf{SHAKE}(X)$$

produces an arbitrarily long sequence of bytes that is computationally indistinguishable from a uniformly random string of the same length. In our analysis we use an explicit PRF security definition. For an adversary $\mathcal{B}$,

$$\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{SHAKE}}(\mathcal{B}) = \left|\Pr[\mathcal{B}^{\mathsf{SHAKE}(\cdot)} = 1] - \Pr[\mathcal{B}^{R(\cdot)} = 1]\right|,$$

where R is a function that on each input returns a uniformly random output of matching length.

KMAC. We model $\mathsf{KMAC}_K(M; \mathsf{custom})$ as a keyed pseudo-random function on the message input and customization string. For adversary $\mathcal{B}$,

$$\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{KMAC}}(\mathcal{B}) = \left|\Pr[\mathcal{B}^{\mathsf{KMAC}_K(\cdot)} = 1] - \Pr[\mathcal{B}^{R(\cdot)} = 1]\right|,$$

with K drawn uniformly at random. HKDS uses KMAC with key sizes matching the underlying security parameter and with explicit customization strings to enforce domain separation.

Assumptions. All reductions in this paper rely on the following assumptions:

1. SHAKE behaves as a pseudo-random function on its absorbed input.
2. KMAC behaves as a pseudo-random function under its key.
3. The absorbed input domains used by HKDS are disjoint and uniquely decodable.

3.4 System and Implementation Assumptions

The security of HKDS depends on several operational assumptions that are outside the scope of the cryptographic model but essential for correct deployment.

Erasure Model. HKDS requires that the client erase intermediate key material after use. In particular, the decrypted token TOK, the pad Kpad, and consumed elements of TKC must be overwritten immediately. Our forward secrecy and post compromise security definitions assume that these erasures occur as specified.

Random Number Generation. The master derivation key $(\mathsf{BDK}, \mathsf{STK})$ must be generated using a cryptographically secure random number generator. All security bounds rely on these keys being independent uniformly random values.

Constant Time Implementations. All invocations of SHAKE and KMAC must be implemented in a manner that prevents side channel leakage of absorbed inputs or key material. The security model excludes adversaries who exploit timing, power, electromagnetic, or microarchitectural side channels. HKDS is secure only to the extent that these implementations are constant time and do not leak internal state.

Protocol Scope. This analysis applies to HKDS operating in authenticated mode with full round Keccak permutations. Optional reduced round or unauthenticated operation modes described in the HKDS specification fall outside the scope of this formal analysis.

4 Security Model for HKDS

4.1 Adversarial Capabilities

We consider an adversary $\mathcal{A}$ that has full control over the communication channel between clients and the server. The adversary may intercept, modify, drop, reorder, and inject packets. It may request tokens, observe encrypted tokens, and obtain ciphertexts and authentication tags produced by honest devices. The adversary may also cause decryption attempts at the server side and observe acceptance or rejection outcomes.
The model allows corruption of client devices. When a client is corrupted, the adversary obtains all current readable client state, including the embedded device key EDK, any cached token material that has not been erased, and any unconsumed transaction keys. Server compromise is not modeled explicitly except in the dedicated partial compromise analysis. The server held roots BDK and STK are assumed secret unless stated otherwise. The adversary is computationally bounded and interacts with the protocol through the oracle interface described below. All security notions are defined through these oracle interactions.

4.2 Game Based Definitions

We model HKDS through two execution experiments. In the real experiment, all keys and outputs follow the protocol specification. In the random experiment, test messages are replaced with random strings of matching length.
There is a single global system state containing the master derivation key and a set of client records. Each client record stores the device identity, the transaction counter, and any state that persists across token exchanges. The per client state evolves deterministically based on oracle calls. Each oracle precisely specifies how the system state is updated.

Real Experiment. In $\mathsf{Exp}^{\mathsf{REAL}}_{\mathsf{HKDS}}(\mathcal{A})$, the adversary interacts with the oracles defined below. The experiment outputs 1 if the adversary correctly guesses the hidden bit in the confidentiality game and 0 otherwise.

Random Experiment. In $\mathsf{Exp}^{\mathsf{RAND}}_{\mathsf{HKDS}}(\mathcal{A})$, the experiment behaves identically except that challenge outputs are replaced with random strings. Again the experiment outputs 1 if the adversary guesses the hidden bit.

Adversarial Advantage. For security notion P, the advantage of $\mathcal{A}$ is

$$\mathsf{Adv}^{P}_{\mathsf{HKDS}}(\mathcal{A}) = \left|\Pr[\mathsf{Exp}^{\mathsf{REAL}}_{\mathsf{HKDS}}(\mathcal{A}) = 1] - \Pr[\mathsf{Exp}^{\mathsf{RAND}}_{\mathsf{HKDS}}(\mathcal{A}) = 1]\right|.$$

4.3 Oracles

All oracle definitions include both their functional behavior and the corresponding state transitions.

NewEpoch(Client). This oracle models the token request and response sequence. On input a client identifier C, the oracle increments the token request counter, constructs CTOK and TMS, derives TOK and Kpad, forms ETOK, computes the token tag

$$\mathsf{TAG}_t = \mathsf{KMAC}_{\mathsf{EDK}}(\mathsf{ETOK}; \mathsf{custom} = \mathsf{TMS}),$$

and returns $(\mathsf{ETOK}, \mathsf{TAG}_t, \mathsf{KSN})$ to the adversary. The internal state is updated to reflect the new epoch.

Encrypt(Client, M, A). Encrypt(Client, M, A). This oracle models honest client encryption and authentication. It derives the next two transaction keys from the client's cache, updates the transaction counter, computes:

$$C = M \oplus K_{\text{enc}}, \qquad \text{TAG} = \text{KMAC}_{K_{\text{mac}}}\big(C;\, \text{custom} = A\big),$$

and returns $(C, \text{TAG}, \mathsf{KSN})$ to the adversary.

VerifyDecrypt(Client, Ctx). This oracle models the server side processing of ciphertexts supplied by the adversary. It reconstructs the appropriate transaction keys from KSN, verifies the tag, and if verification succeeds outputs the corresponding plaintext. If verification fails it outputs a rejection symbol. Only the accept or reject behavior is revealed to the adversary.

CorruptClient(Client). This oracle models corruption of a client device. It returns the current readable state of the device, including EDK and any unconsumed transaction keys. It does not return erased keys or previously consumed cache entries. After corruption, the adversary controls all future behavior of that client.

Test(Client, M0, M1). This oracle is used for confidentiality. It selects a hidden bit b and encrypts M_b under the next transaction key. In the real experiment it returns $(C, \mathsf{TAG}, \mathsf{KSN})$ computed honestly. In the random experiment it returns a pair of uniformly random strings of matching length. The adversary may not invoke Test on a corrupted client or on a client that has previously appeared in CorruptClient.

Forward Secrecy Oracles. For the forward secrecy experiment, we introduce an oracle that allows corruption at time t and a challenge on keys from the earlier epoch. These oracles operate under the erasure model and are defined in the forward secrecy subsection.

Post Compromise Recovery Oracles. For predictive resistance, we introduce oracles that allow corruption of a client prior to a new epoch and test confidentiality of keys derived after token renewal.

4.4 Security Notions

4.4.1 Message Confidentiality (IND CPA)

HKDS provides confidentiality if no polynomial time adversary can distinguish encryptions of two chosen messages under the transaction keys derived from the token. The adversary may use all oracles except CorruptClient on the challenged client. The IND CPA advantage is defined as

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{HKDS}}(\mathcal{A}) = \Big|\Pr[\mathsf{Exp}^{\mathsf{REAL}}_{\mathsf{HKDS}}(\mathcal{A}) = 1] - \Pr[\mathsf{Exp}^{\mathsf{RAND}}_{\mathsf{HKDS}}(\mathcal{A}) = 1]\Big|.$$

4.4.2 Message Authenticity (UF CMA)

HKDS provides message authenticity if no adversary using the Encrypt oracle as a signing oracle can produce a ciphertext and tag that the server accepts but that was not produced by an honest client invocation. The UF CMA advantage is the probability that VerifyDecrypt accepts a forgery.

4.4.3 Token Authenticity

Token authenticity ensures that no adversary can produce $(\mathsf{ETOK}, \mathsf{TAG}_t)$ for which the client verifies the KMAC tag while ETOK was not generated by an honest server. The advantage of any such adversary is defined as the probability of producing a pair that passes verification.

4.4.4 Forward Secrecy

Forward secrecy holds if compromise of the client after a token epoch does not reveal any information about transaction keys from earlier epochs. The forward secrecy experiment allows the adversary to:

1. obtain a challenge on a transaction key from epoch $t - 1$
2. corrupt the client at time t

and requires that the adversary be unable to distinguish the earlier key from random. All definitions assume that the client erases all expired token and cache material prior to corruption.

4.4.5 Predictive Resistance and Post Compromise Security

Predictive resistance, also known as post compromise security, captures the ability of HKDS to recover security after client compromise. The adversary may corrupt the client at time t and learn all current state. After the client receives a fresh token in epoch $t + 1$, the adversary is challenged on keys derived from the new token. HKDS provides PCS if the adversary cannot distinguish these keys from random in the random experiment.

5 Domain Separation and Structural Properties

5.1 Absorbed Domains for SHAKE and KMAC

HKDS uses SHAKE and KMAC on a small number of well defined input domains. Each domain corresponds to a distinct stage in the key hierarchy. We summarize these domains below.

SHAKE domains.

- Device key derivation:
$$\mathsf{EDK} = \mathsf{SHAKE}(\mathsf{DID} \parallel \mathsf{BDK}).$$
- Token derivation:
$$\mathsf{TOK} = \mathsf{SHAKE}(\mathsf{CTOK} \parallel \mathsf{STK}).$$
- Token pad derivation:
$$\mathsf{Kpad} = \mathsf{SHAKE}(\mathsf{CTOK} \parallel \mathsf{EDK}).$$
- Transaction key cache derivation:
$$\mathsf{TKC} = \mathsf{SHAKE}(\mathsf{TOK} \parallel \mathsf{EDK}).$$

The values DID, BDK, STK, CTOK, and TOK have fixed and distinct lengths determined by the HKDS specification. In particular, DID and CTOK include protocol flags and identifiers that do not appear in any other absorbed input.

KMAC domains. KMAC is used in two ways.

- Token authenticity:

$$\mathsf{TAG}_t = \mathsf{KMAC}_{\mathsf{EDK}}(\mathsf{ETOK}; \mathsf{custom} = \mathsf{TMS}).$$

- *Message authenticity:*

$$\mathrm{TAG} = \mathrm{KMAC}_{K_{\mathrm{mac}}}\big(C;\ \mathrm{custom} = A\big),$$

  where C is the ciphertext block and A is the associated data.

The keys for these invocations are independent outputs of SHAKE on distinct absorbed domains. The customization string for token authentication, TMS, contains the key serial number and KMAC variant name, while the message MAC uses the associated data A as its customization string. This enforces explicit domain separation between token and message tags.

Lemma 1 (Domain Separation)**.** *Let $\mathcal{B}$ be any adversary that interacts with the HKDS construction and treats SHAKE or KMAC as a black box. Suppose $\mathcal{B}$ can exploit a collision or cross domain confusion between any two of the absorbed inputs listed above. Then one can construct a distinguisher $\mathcal{D}$ that achieves non negligible PRF advantage against SHAKE or KMAC. Equivalently, under the PRF assumptions on SHAKE and KMAC, the HKDS input domains are computationally separated.*

Proof. We consider the SHAKE and KMAC domains separately.
For SHAKE, the absorbed inputs are of the form

$$X_1 = \mathsf{DID} \parallel \mathsf{BDK}, X_2 = \mathsf{CTOK} \parallel \mathsf{STK}, X_3 = \mathsf{CTOK} \parallel \mathsf{EDK}, X_4 = \mathsf{TOK} \parallel \mathsf{EDK}.$$

The strings DID, CTOK, BDK, STK, and EDK have fixed lengths and are constructed from disjoint sets of fields. In particular, DID and CTOK contain protocol flags and type information that never appear inside keys, while BDK, STK, and EDK are uniformly random key strings with lengths chosen according to the security level. This implies that there is an injective map from the tuples

$$(\mathsf{DID}, \mathsf{BDK}),\ (\mathsf{CTOK}, \mathsf{STK}),\ (\mathsf{CTOK}, \mathsf{EDK}),\ (\mathsf{TOK}, \mathsf{EDK})$$

to the set of absorbed inputs $\{X_1, X_2, X_3, X_4\}$. In particular, no two pairs of semantic values can produce the same concatenated input except with probability negligible in the key lengths. Hence the SHAKE domains are uniquely decodable and structurally disjoint. Suppose now that an adversary $\mathcal{B}$ can exploit a relationship between outputs of SHAKE on different HKDS domains with non negligible advantage. We build a distinguisher $\mathcal{D}$ that is given oracle access to either a real SHAKE instance or a uniformly random function. $\mathcal{D}$ simulates the HKDS environment for $\mathcal{B}$ by implementing all SHAKE calls through its oracle. If $\mathcal{B}$ succeeds in distinguishing the HKDS derived values from the ideal behavior required by the protocol, then $\mathcal{D}$ uses this event to output a guess that its oracle is real rather than random. The probability gap between these two cases is exactly the advantage claimed by $\mathcal{B}$. Therefore any non negligible cross domain advantage contradicts the PRF security of SHAKE.
For KMAC, the absorbed domains are determined by the key and customization string. Token tags use the key EDK with customization TMS, while message tags use independent keys K_{mac} with empty customization. The keys EDK and K_{mac} originate from SHAKE on different absorbed inputs, so under the PRF assumption on SHAKE they are computationally independent. The customization strings are also disjoint, since TMS always contains a key serial number and MAC name, while the message MAC customization is empty.

If an adversary $\mathcal{B}$ could exploit a structural relationship between these KMAC domains, we would construct a PRF distinguisher $\mathcal{D}$ for KMAC. The distinguisher samples either a real KMAC instance under a random key or a random function, chooses one of the HKDS domains to embed in its oracle, and simulates the remaining domains using independent random keys. Any cross domain advantage of $\mathcal{B}$ translates into an ability of $\mathcal{D}$ to distinguish KMAC from random.
In both SHAKE and KMAC cases, we conclude that, under the PRF assumptions on the underlying primitives, the input domains used by HKDS are computationally separated. No efficient adversary can transfer a distinguishing advantage between these domains without breaking the assumed security of SHAKE or KMAC. □

5.2 Determinism and Recoverability

The HKDS client and server compute all keys as deterministic functions of their shared inputs. We record this as a simple structural property.

Lemma 2 (Deterministic Recoverability)**.** *Let* $(\mathsf{BDK}, \mathsf{STK}, \mathsf{KID})$ *be a fixed master derivation key and let* DID *and* KSN *be fixed device identity and key serial number values. Then the tuple*

$$(\mathsf{EDK}, \mathsf{TOK}, \mathsf{TKC})$$

is uniquely determined as a deterministic function of $(\mathsf{MDK}, \mathsf{DID}, \mathsf{KSN})$*. In particular, any honest client and honest server that hold the same* $(\mathsf{MDK}, \mathsf{DID}, \mathsf{KSN})$ *will compute identical transaction keys at each index of the cache.*

Proof. The values DID and MDK determine the embedded device key

$$\mathsf{EDK} = \mathsf{SHAKE}(\mathsf{DID} \parallel \mathsf{BDK})$$

deterministically. The key serial number KSN and device identity DID determine the token request counter and the customization string CTOK. Together with the fixed secret token key STK, this yields a unique token

$$\mathsf{TOK} = \mathsf{SHAKE}(\mathsf{CTOK} \parallel \mathsf{STK}).$$

Finally, $(\mathsf{TOK}, \mathsf{EDK})$ determine the transaction key cache

$$\mathsf{TKC} = \mathsf{SHAKE}(\mathsf{TOK} \parallel \mathsf{EDK}),$$

which is interpreted as a sequence of single use keys by partitioning the output into fixed length blocks. All computations involve deterministic functions with no additional randomness beyond the master keys.
An honest client and server that share the same $(\mathsf{MDK}, \mathsf{DID}, \mathsf{KSN})$ therefore compute identical values of EDK, TOK, and TKC. When both parties advance the transaction counter in the same manner, they extract the same pair of keys $(K_{\mathsf{enc}}, K_{\mathsf{mac}})$ for each transaction. This establishes deterministic recoverability. □

6 Provable Security of HKDS

6.1 Reduction for Message Confidentiality

Theorem 1 (IND-CPA Security)**.** *Let* $\mathcal{A}$ *be any probabilistic polynomial time adversary that interacts with the HKDS oracles in the IND-CPA experiment and makes at most* q_{enc} *encryption or test queries. Then there exists a probabilistic polynomial time distinguisher* $\mathcal{B}$ *against the PRF security of* SHAKE *such that:*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{HKDS}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{SHAKE}}(\mathcal{B}) + \frac{q_{\mathsf{enc}}}{2^n},$$

where n is the tag and key length in bits. In particular, if SHAKE *is a secure PRF on its absorbed domains and transaction keys are used once, then HKDS is IND-CPA secure.*

Proof. We use a standard game hopping argument.

Game G_0. In G_0 the challenger runs the real HKDS protocol and answers all oracle queries exactly as specified. The adversary's advantage in distinguishing the hidden bit is $\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{HKDS}}(\mathcal{A})$.

Game G_1. In G_1 we replace all invocations of SHAKE that derive transaction key caches and token pads by calls to a random function with the same input and output interface. Concretely, for each absorbed input of the form (TOK ‖ EDK) or (CTOK ‖ EDK) the challenger draws an independent uniform output and stores it in a table. All subsequent queries on the same input receive the stored value.
If an adversary could distinguish G_0 from G_1 with non negligible advantage, we would build a distinguisher $\mathcal{B}$ that uses $\mathcal{A}$ to distinguish SHAKE from a random function. Distinguisher $\mathcal{B}$ simulates the HKDS challenger for $\mathcal{A}$, forwarding all relevant absorbed inputs to its own oracle, and uses the responses to derive the transaction keys and pads in place of real SHAKE outputs. Any difference in $\mathcal{A}$'s view between the two cases gives $\mathcal{B}$ the same advantage. Therefore

$$|\Pr[G_0(\mathcal{A}) = 1] - \Pr[G_1(\mathcal{A}) = 1]| \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{SHAKE}}(\mathcal{B}).$$

Game G_2. In G_2 we keep the random function behavior of G_1 but modify the way challenge ciphertexts are produced. For each test query, the challenger still uses the next random transaction key K_{enc} but replaces the challenge ciphertext by the encryption of a random message of the same length. Since K_{enc} is uniformly random and used exactly once, the ciphertext $C = M \oplus K_{\mathsf{enc}}$ is a one time pad. From the adversary's point of view, the ciphertext is uniformly distributed over all strings of the correct length, independent of M. Hence changing the encrypted message has no effect on the distribution of C.
The authentication tag is computed under an independent random key K_{mac} and a PRF KMAC instance. The tag distribution is therefore also independent of the hidden bit. It follows that in G_2 the adversary has no information about the challenge bit and

$$\Pr[G_2(\mathcal{A}) = 1] = \frac{1}{2}.$$

Bounding the transition from G_1 to G_2. The only difference between G_1 and G_2 is the choice of message that is encrypted under a one time pad. For each challenge, the view of $\mathcal{A}$ is identical in the two games except with the event that the adversary outputs a correct guess for the hidden bit while the ciphertext distribution is uniform. This event has probability at most $1/2$ plus at most $q_{\mathsf{enc}}/2^n$ due to the possibility of trivial collisions between different keys or messages, which we bound conservatively by $q_{\mathsf{enc}}/2^n$. Thus

$$|\Pr[G_1(\mathcal{A}) = 1] - \Pr[G_2(\mathcal{A}) = 1]| \leq \frac{q_{\mathsf{enc}}}{2^n}.$$

Combining the bounds. Since G_2 gives no information about the challenge bit we have $\Pr[G_2(\mathcal{A}) = 1] = 1/2$. Combining the above inequalities yields

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{HKDS}}(\mathcal{A}) = \left|\Pr[G_0(\mathcal{A}) = 1] - \frac{1}{2}\right| \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{SHAKE}}(\mathcal{B}) + \frac{q_{\mathsf{enc}}}{2^n}.$$

This completes the proof. □

6.2 Reduction for Message Authenticity

Theorem 2 (UF-CMA Security)**.** *Let $\mathcal{A}$ be a probabilistic polynomial time adversary that interacts with HKDS as a chosen message attacker and makes at most q_{mac} encryption queries. Then there exists a probabilistic polynomial time adversary $\mathcal{B}$ against the PRF security of* KMAC *such that*

$$\mathsf{Adv}^{\mathsf{UF\text{-}CMA}}_{\mathsf{HKDS}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{KMAC}}(\mathcal{B}) + \frac{q_{\mathsf{mac}}}{2^n}.$$

Proof. The adversary $\mathcal{A}$ obtains tags of the form

$$\mathsf{TAG} = \mathsf{KMAC}_{K_{\mathsf{mac}}}(C \parallel A),$$

where each K_{mac} is a fresh key derived from the transaction key cache. Under the PRF assumption on SHAKE and the domain separation lemma, these keys are computationally independent and uniformly random from the view of $\mathcal{A}$.
We first conceptually replace the derived keys K_{mac} by independent uniform keys. Any difference between these two settings can be bounded by a PRF adversary against SHAKE as in the confidentiality proof, which we conservatively absorb into the $\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{KMAC}}(\mathcal{B})$ term. In the idealized setting where each tag is computed under an independent random key, the situation reduces to the standard UF-CMA security of KMAC. We construct an adversary $\mathcal{B}$ against KMAC as follows. Adversary $\mathcal{B}$ receives oracle access to either $\mathsf{KMAC}_K(\cdot)$ for a random key K or a random function. It uses this oracle to answer all tag generation queries of $\mathcal{A}$ for a single chosen transaction key. For all other transaction keys it samples independent random keys and simulates KMAC locally.
If $\mathcal{A}$ outputs a forgery $(C^*, A^*, \mathsf{TAG}^*)$ that verifies under some K_{mac} and was not returned by the encryption oracle, then with probability at least $1/q_{\mathsf{mac}}$ this key is the one that $\mathcal{B}$ has embedded in its KMAC oracle. In that case $\mathcal{B}$ outputs the forgery as a distinguishing event. The usual analysis for MACs under a PRF assumption yields

$$\Pr[\text{forge in real world}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{KMAC}}(\mathcal{B}) + \frac{q_{\mathsf{mac}}}{2^n},$$

where the second term accounts for trivial brute force guessing of a correct tag. This bound transfers directly to the HKDS setting, which proves the theorem. □

6.3 Token Authenticity

Theorem 3 (Token Authenticity)**.** *Let $\mathcal{A}$ be a probabilistic polynomial time adversary that observes an arbitrary number of valid encrypted tokens and tags and attempts to forge a new pair $(\mathsf{ETOK}^*, \mathsf{TAG}^*_t)$ that passes client verification. Then there exists a PRF adversary $\mathcal{B}$ against* KMAC *such that*

$$\mathsf{Adv}^{\mathsf{TokAuth}}_{\mathsf{HKDS}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{KMAC}}(\mathcal{B}).$$

Proof. The token tag is defined as

$$\mathsf{TAG}_t = \mathsf{KMAC}_{\mathsf{EDK}}(\mathsf{ETOK}; \mathsf{custom} = \mathsf{TMS}),$$

where the key EDK is derived from $\mathsf{SHAKE}(\mathsf{DID} \parallel \mathsf{BDK})$ and never leaves the device. Under the PRF assumption on SHAKE, the value EDK is computationally indistinguishable from a uniformly random key from the view of $\mathcal{A}$, even in the presence of observed ciphertexts and tags.
We therefore reduce the problem of forging a token tag to the UF-CMA security of KMAC under a fixed key. Adversary $\mathcal{B}$ receives oracle access to either $\mathsf{KMAC}_K(\cdot)$ or a random function, for a random key K and customization string TMS. It emulates the token

generation process for $\mathcal{A}$ by choosing ETOK values according to the HKDS specification and obtaining their corresponding tags from its oracle. When $\mathcal{A}$ outputs a candidate forgery $(\mathsf{ETOK}^*, \mathsf{TAG}_t^*)$ that was not previously returned by the oracle, $\mathcal{B}$ outputs the same pair as its own forgery.
If the oracle is a real KMAC instance then the success probability of $\mathcal{B}$ is exactly the success probability of $\mathcal{A}$ in causing the client to accept a forged token. If the oracle is a random function then the probability that any new input produces a valid tag is negligible. Thus a non negligible advantage for $\mathcal{A}$ would contradict the PRF security of KMAC. This yields the stated bound. □

6.4 Forward Secrecy under Erasure

Definition 1 (Erasure Model)**.** For each client we consider a sequence of epochs indexed by $i \geq 1$. At the start of epoch i the client holds a token TOK_i and the corresponding transaction key cache TKC_i. Let state_i denote the full client state at the end of epoch i. We say that HKDS operates under the erasure model if at the transition from epoch i to $i+1$ the client applies an erasure function

$$\mathsf{Erase}(\mathsf{state}_i) = \mathsf{state}_i',$$

which overwrites all copies of TOK_i, Kpad_i, and all consumed entries of TKC_i. The next epoch is initialized from state_i' and fresh protocol messages.

Theorem 4 (Forward Secrecy)**.** *Consider an adversary that interacts with HKDS across multiple epochs and is allowed to corrupt a client only after the erasure step at the end of epoch i. Under the PRF assumption on* SHAKE *and the erasure model, there exists a distinguisher $\mathcal{B}$ such that the adversary's advantage in distinguishing any key from* TKC_i *from a uniform random string is bounded by*

$$\mathsf{Adv}^{\mathsf{FS}}_{\mathsf{HKDS}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{SHAKE}}(\mathcal{B}).$$

Proof. At the end of epoch i and after applying the erasure function, the client state no longer contains TOK_i, Kpad_i, or any consumed elements of TKC_i. The only remaining values that depend on these secrets are messages that have already been sent and their tags. For any key K in TKC_i, the distribution of transcripts from epoch i is determined by encryptions under K and by KMAC tags under independent keys. Once K and the corresponding token material are erased, the adversary has no further access to these values.
Suppose now that an adversary could distinguish a past transaction key K from random in the forward secrecy experiment. We build a distinguisher $\mathcal{B}$ for the PRF security of SHAKE. Distinguisher $\mathcal{B}$ receives oracle access to either SHAKE or a random function on the absorbed input $(\mathsf{TOK}_i \parallel \mathsf{EDK})$. It uses this oracle to generate the cache TKC_i in the simulated epoch i and then erases the relevant state. When the adversary requests a challenge on a key from TKC_i, $\mathcal{B}$ forwards the corresponding entry to $\mathcal{A}$. If $\mathcal{A}$ can distinguish this key from random with non negligible advantage after erasure, then $\mathcal{B}$ can distinguish whether its oracle is real or random with the same advantage. This contradicts the PRF assumption on SHAKE and yields the stated bound. □

6.5 Predictive Resistance and Post Compromise Security

Theorem 5 (Post Compromise Security)**.** *Consider an adversary that corrupts a client at the end of epoch i before erasure and learns the full state, including* EDK, TOK_i, *and* TKC_i. *The adversary then allows the client to execute the protocol for a fresh epoch $i+1$, in which a new token* TOK_{i+1} *and cache* TKC_{i+1} *are derived. Under the PRF assumption*

on SHAKE *for the absorbed input* ($\mathsf{CTOK}_{i+1} \parallel \mathsf{STK}$) *and* ($\mathsf{TOK}_{i+1} \parallel \mathsf{EDK}$)*, and assuming that* STK *remains secret, the adversary's advantage in distinguishing any key from* TKC_{i+1} *from random is bounded by the PRF advantage against* SHAKE*.*

Proof. At the time of compromise, the adversary learns EDK, TOK_i, and the entire cache TKC_i. It does not, however, learn the secret token key STK. When the client initiates epoch $i+1$, it constructs a new customization string CTOK_{i+1} that contains a fresh token request counter and the same device identity. The server then computes

$$\mathsf{TOK}_{i+1} = \mathsf{SHAKE}(\mathsf{CTOK}_{i+1} \parallel \mathsf{STK}).$$

From the view of the adversary, CTOK_{i+1} is known but STK remains a uniform secret. Under the PRF assumption on SHAKE, the value TOK_{i+1} is indistinguishable from a uniform string independent of the compromised state.
The client then derives

$$\mathsf{TKC}_{i+1} = \mathsf{SHAKE}(\mathsf{TOK}_{i+1} \parallel \mathsf{EDK}).$$

At this point the adversary knows EDK but, under the previous argument, treats TOK_{i+1} as a fresh pseudo-random value. Consequently, the absorbed input ($\mathsf{TOK}_{i+1} \parallel \mathsf{EDK}$) defines a new and independent PRF instance of SHAKE. We now construct a distinguisher $\mathcal{B}$ that uses the adversary to distinguish this instance from a random function, exactly as in the confidentiality proof.
If the adversary could distinguish keys from TKC_{i+1} from random with non negligible advantage, then $\mathcal{B}$ would break the PRF security of SHAKE on the input domain ($\mathsf{TOK}_{i+1} \parallel$ EDK). Therefore the advantage of $\mathcal{A}$ in the post compromise experiment is bounded by the PRF advantage against SHAKE, which proves the theorem. □

6.6 Partial Server Compromise

We conclude with a brief analysis of partial compromise of the server side roots. This does not give full security theorems but clarifies how each root contributes to the overall defense in depth.

6.6.1 Exposure of BDK Only

Theorem 6. *If an adversary learns* BDK *but not* STK*, then it can reconstruct the embedded device key* EDK *for any device identity* DID*. Given* EDK *and any observed encrypted token* ETOK *and customization* CTOK*, the adversary can compute the pad* Kpad = SHAKE(CTOK ∥ EDK) *and decrypt the corresponding token. As a result, confidentiality of future transaction keys for observed sessions is lost, although the generation of new tokens still depends on the secrecy of* STK*.*

6.6.2 Exposure of STK Only

Theorem 7. *If an adversary learns* STK *but not* BDK*, then it can compute* TOK = SHAKE(CTOK ∥ STK) *for any observed customization string* CTOK*. However, without* EDK *it cannot derive the pad* Kpad *or the transaction key cache* TKC = SHAKE(TOK ∥ EDK)*. In this setting token values are exposed, while the confidentiality of transaction keys continues to rely on the secrecy of* BDK *and the PRF security of* SHAKE*.*

Proof. Both statements follow directly from the HKDS key derivation equations. Knowledge of BDK suffices to recompute EDK for any device identity and thus to decrypt any observed encrypted token. Knowledge of STK suffices to recompute tokens from observed customization strings but, in the absence of BDK, does not reveal EDK or any cache entries.

In neither case can the adversary recover both roots simultaneously without an additional compromise, so each root adds a layer of defense. The precise residual guarantees in each scenario are as stated in the theorems above. □

7 Concrete Security Bounds

In this section we collect the bounds derived in the preceding proofs and express them in terms of the security parameter and the number of oracle queries made by the adversary. We let

$$n \in \{128, 256, 512\}$$

denote the effective key and tag length in bits. We write

- q_{enc} for the total number of encryption and test queries,
- q_{mac} for the total number of message authentication queries,
- q_{tok} for the total number of token queries,
- q_{fs} and q_{pcs} for the number of forward secrecy and post compromise experiments respectively.

We also write

$$\varepsilon_{\mathsf{SHAKE}} = \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{SHAKE}}(\cdot), \qquad \varepsilon_{\mathsf{KMAC}} = \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{KMAC}}(\cdot)$$

for the maximum attainable PRF advantages of any adversary with comparable resources against SHAKE and KMAC.

Summary of Bounds

The reductions of Section 6 yield the following inequalities.

Message confidentiality. For any adversary $\mathcal{A}$ that makes at most q_{enc} encryption and test queries,

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{HKDS}}(\mathcal{A}) \le \varepsilon_{\mathsf{SHAKE}} + \frac{q_{\mathsf{enc}}}{2^n}.$$

Message authenticity. For any adversary that makes at most q_{mac} chosen message queries,

$$\mathsf{Adv}^{\mathsf{UF\text{-}CMA}}_{\mathsf{HKDS}}(\mathcal{A}) \le \varepsilon_{\mathsf{KMAC}} + \frac{q_{\mathsf{mac}}}{2^n}.$$

Token authenticity. For any adversary that observes an arbitrary number of valid tokens and attempts to forge a new pair,

$$\mathsf{Adv}^{\mathsf{TokAuth}}_{\mathsf{HKDS}}(\mathcal{A}) \le \varepsilon_{\mathsf{KMAC}}.$$

Forward secrecy. Under the erasure model, any adversary that corrupts clients only after epoch transitions satisfies

$$\mathsf{Adv}^{\mathsf{FS}}_{\mathsf{HKDS}}(\mathcal{A}) \le \varepsilon_{\mathsf{SHAKE}}.$$

Post compromise security. For any adversary that corrupts a client at the end of epoch i and is challenged on keys from epoch $i+1$, assuming that the server token key remains secret,

$$\mathsf{Adv}^{\mathsf{PCS}}_{\mathsf{HKDS}}(\mathcal{A}) \le \varepsilon_{\mathsf{SHAKE}}.$$

Combined View

For practical parameter choices it is convenient to summarize these bounds in a single expression. Let

$$q_{\mathsf{tot}} = q_{\mathsf{enc}} + q_{\mathsf{mac}}$$

denote the combined number of message level oracle queries. Neglecting small constant factors, the overall distinguishing advantage of any adversary that simultaneously attempts to break confidentiality, authenticity, token authenticity, and the epoch level properties is bounded by

$$\mathsf{Adv}^{\mathsf{HKDS}}_{\mathcal{A}} \lesssim \varepsilon_{\mathsf{SHAKE}} + \varepsilon_{\mathsf{KMAC}} + \frac{q_{\mathsf{tot}}}{2^n}.$$

For $n = 128$ and adversaries with at most 2^{64} oracle queries, the generic term $q_{\mathsf{tot}}/2^n$ remains below 2^{-64}. For higher security levels the corresponding bound decays accordingly. These estimates justify the interpretation of HKDS as providing roughly n bits of security against generic attacks under the assumed PRF strength of SHAKE and KMAC and correct implementation of erasure and side channel protections.

Table 2: Concrete Security Bounds for HKDS

Security Property	Adversarial Resources	Upper Bound
IND–CPA Confidentiality	q_{enc} encrypt/test queries	$\varepsilon_{\mathsf{SHAKE}} + \frac{q_{\mathsf{enc}}}{2^n}$
UF–CMA Authenticity	q_{mac} MAC queries	$\varepsilon_{\mathsf{KMAC}} + \frac{q_{\mathsf{mac}}}{2^n}$
Token Authenticity	q_{tok} token observations	$\varepsilon_{\mathsf{KMAC}}$
Forward Secrecy	q_{fs} FS queries	$\varepsilon_{\mathsf{SHAKE}}$
Post Compromise Security	q_{pcs} PCS queries	$\varepsilon_{\mathsf{SHAKE}}$

8 Implementation and System Considerations

8.1 Constant Time Implementations and Side Channels

All provable security guarantees established in this paper rely on the assumption that the underlying cryptographic primitives are implemented in a manner that does not leak secret information through timing or other physical channels. In particular, all calls to SHAKE and KMAC must be executed in constant time with respect to their absorbed inputs and keys. Neither the control flow nor the memory access pattern of these functions may depend on secret values such as EDK, TOK, Kpad, or any element of the transaction key cache.

Processing of token packets must also follow constant time principles. The client must verify the token tag, derive the token pad, and decrypt ETOK without exhibiting data dependent timing behavior. Error handling paths, such as rejection of invalid tokens, must be indistinguishable from acceptance paths except for the final success or failure indicator. These properties ensure that adversaries cannot recover key material via timing analysis or differential response behaviors.

Side channel leakage from the secure module or execution environment lies outside the scope of the model considered in this work. The security theorems do not apply in settings where an adversary can observe intermediate SHAKE state, HMAC internal variables, power traces, electromagnetic emissions, or other fine grained physical leakage. Deployment of HKDS in such environments requires additional countermeasures that are implementation specific and beyond the formal analysis presented here.

8.2 Random Number Generation

The security of HKDS depends critically on the proper generation of the master derivation key

$$\mathsf{MDK} = (\mathsf{BDK}, \mathsf{STK}, \mathsf{KID}).$$

The values BDK and STK must be sampled using a cryptographically secure random number generator (CSPRNG) or a hardware true random number generator (TRNG). These keys must be independent and uniformly distributed over their respective domain sizes. Any bias or reuse of these values directly reduces the security of device key derivation, token generation, and the transaction key cache.
If BDK is generated with insufficient entropy, the derived device keys may become predictable or vulnerable to brute force search, compromising token confidentiality and token MAC verification. If STK lacks entropy or is reused across deployments, the tokens TOK may become guessable, breaking post compromise recovery and enabling reconstruction of transaction keys. It is therefore essential that both BDK and STK be generated using a trusted randomness source and protected with the same rigor as high value symmetric keys.
The key identifier KID must also be generated to avoid collisions between master derivation keys. Although the value itself does not require secrecy, it must uniquely identify the correct MDK for every device to prevent misrouting of verification operations or incorrect token reconstruction.

8.3 Error Handling, Counters, and Replay

Correct implementation of HKDS requires careful handling of counters, replay detection, and state transitions to preserve the correspondence between the operational protocol and the formal security model.

Transaction counter discipline. The transaction counter KTC must evolve monotonically and must never reuse the same value within a single device lifetime. Any reuse of KTC causes the server to reconstruct the same pair of transaction keys, enabling adversaries to mount tag replay or ciphertext replay attacks. The formal security proofs assume that every KTC value is used at most once.

Token request counter and cache exhaustion. The token request counter TRC must be computed in accordance with the cache multiplier and the number of single use keys available in the transaction key cache. When the cache is exhausted, the client must request a fresh token. If the client fails to request a new token at the correct time, the client and server may diverge in their view of the appropriate transaction key index, resulting in decryption failures or acceptance failures. These events fall outside the formal model unless treated as synchronization faults.

Replay prevention. Replay of encrypted tokens is prevented by the customization string CTOK, which includes the token request counter and the device identity. Replaying a ciphertext and tag pair obtained from the client to the server is prevented by the use of per transaction keys and the monotonicity of KTC. The formal security model captures these protections through the design of the oracles and the definition of freshness conditions for test queries.

Error responses and desynchronization. Implementations must ensure that error responses from the server do not leak information about partial verification results. All error messages should follow uniform timing and formatting. When desynchronization

occurs, the client must request a fresh token to restore alignment. The formal recoverability lemma guarantees that the server will recompute the same state when provided with the correct KSN, but it is the responsibility of the implementation to correctly detect and recover from lost or reordered packets.
Overall, these operational details ensure that the deployed protocol behaves consistently with the formal model. When deviations occur, such as counter reuse or incorrect erasure of key material, the formal security guarantees no longer apply.

9 Conclusion

This paper presented a provable security analysis of the Hierarchical Key Distribution System (HKDS), a symmetric key establishment protocol designed for high performance embedded environments. We introduced a complete game based security model that captures the operational behavior of HKDS, including token generation, transaction key derivation, authenticated message encryption, corruption events, and post compromise recovery. The analysis formalized the security goals of confidentiality, message authenticity, token authenticity, forward secrecy under an explicit erasure model, and post compromise security across token epochs.
Using standard reductions to the pseudo-random function security of SHAKE and KMAC, we proved that HKDS achieves IND-CPA confidentiality and UF-CMA authenticity under single use keys, that forged encrypted tokens imply a break of KMAC under the embedded device key, and that forward secrecy and post compromise security follow from the independence of the token key and correct erasure of expired cache material. Domain separation and deterministic recoverability were established as structural properties of the protocol, ensuring that the client and server compute identical transaction keys without requiring per device state. Concrete security bounds were provided for all notions, demonstrating that HKDS inherits the security level of its underlying primitives.
The analysis in this paper operates under several assumptions, including constant time implementations of SHAKE and KMAC, secure erasure of expired keys, and the secrecy of the server root keys. Physical side channels, implementation faults, and denial of service conditions fall outside the present model. The security of optional modes such as reduced round variants or message transmission without authentication are also not covered here. These restrictions reflect the focus on core cryptographic correctness rather than the full breadth of implementation concerns.
Future work includes extending the analysis to cover reduced round Keccak configurations, incorporating explicit leakage resilience models suitable for constrained hardware, and formalizing the behavior of HKDS when integrated into larger system architectures. Additional investigation of robustness under partial or transient key exposure may further strengthen confidence in real world deployments. The results presented here provide a rigorous foundation for HKDS and justify its security claims under well defined and standard cryptographic assumptions.

References

1. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G. *The Keccak Reference.* Submission to the NIST SHA3 Competition, 2011. Available at: https://keccak.team/files/Keccak-reference-3.0.pdf
2. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G. *SHA3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash.* NIST Special Publication 800-185, 2016. Available at: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf
3. National Institute of Standards and Technology. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* FIPS Publication 202, 2015. Available at: https://doi.org/10.6028/NIST.FIPS.202
4. Accredited Standards Committee X9. *Derived Unique Key Per Transaction (DUKPT).* ANSI X9.24-3, 2016. (Standard is paywalled, index available at:) https://www.x9.org/standards/
5. Bergevin, S., Masson, L. *On the Security of Sponge-Based Constructions.* Journal of Cryptographic Engineering, 10(3):235–247, 2020. Available at: https://doi.org/10.1007/s13389-020-00214-z
6. Mouha, N., Van Assche, G. *The Security of Sponge-Based Hybrid Constructions.* In FSE 2016, International Workshop on Fast Software Encryption. Available at: https://doi.org/10.1007/978-3-662-52993-5_5
7. Bellare, M., Rogaway, P. *Introduction to Modern Cryptography.* Lecture Notes, 2000. Available at: https://web.cs.ucdavis.edu/~rogaway/classes/227/spring05/book/main.pdf
8. Krawczyk, H. *Cryptographic Extraction and Key Derivation: The HKDF Scheme.* In CRYPTO 2010, Advances in Cryptology. Available at: https://doi.org/10.1007/978-3-642-14623-7_33
9. National Institute of Standards and Technology. *NIST Cryptographic Security Levels.* 2020. Available at: https://csrc.nist.gov/projects
10. Underhill, J.G. *HKDS Technical Specification.* Technical Specification, 2020. https://www.qrcscorp.ca/documents/hkds_specification.pdf
11. Underhill, J.G. *HKDS Implementation Analysis.* Technical Specification, 2025. https://www.qrcscorp.ca/documents/hkds_analysis.pdf
12. Underhill, J.G. *HKDS Implementation Guide.* Technical Specification, 2025. https://www.qrcscorp.ca/documents/hkds_implementation.pdf
13. QRCS Corp. *HKDS Reference Implementation.* C Source Code, 2024. https://github.com/QRCS-CORP/HKDS