Hierarchal symmetric Key Distribution System – HKDS

Revision 1e, October 20, 2024

John G. Underhill – john.underhill@protonmail.com

This document is an engineering level description of the HKDS symmetric key distribution system. In its contents, is a guide to implementing HKDS, an explanation of its design, a description of the source code, references to its constituent primitives and links to supporting documentation.

Contents **Page**

Figures

Contents **Page**

Tables

Contents **Page**

Foreword

This document is intended as the preliminary draft of a new standards proposal, and as a basis from which that standard can be implemented. We intend that this serve as an explanation of this new technology, and as a complete description of the protocol.

This document is the fifth revision of the specification of HKDS, further revisions may become necessary during the pursuit of a standard model, and revision numbers shall be incremented with changes to the specification. The reader is asked to consider only the most recent revision of this standards draft, as the authoritative expression of the current working model of the specification.

The author of this specification and inventor and owner of HKDS is John G. Underhill, and can be reached at john.underhill@protonmail.com

HKDS, the algorithm constituting the HKDS key distribution system is patent pending, and is owned by John G. Underhill and QRCS Inc.. The code described herein is copyrighted and owned by John G. Underhill.

1: Introduction

One of the most challenging problems faced by both cryptographers and the financial services industry, is that of key distribution. Many millions of terminals employed in many different capacities, each needing to establish secure communications with a server to perform various financial transactions that we all use every day. The scale of these transactions has in just a few decades, become enormous, and electronic payment is steadily replacing fiat currency as the primary means by which consumer purchases are transacted.

With this global change in the primary mode of sales and currency exchange, many serious technological barriers have been overcome in a relatively short period of time, and chief among these, has been the secure transfer of sensitive banking information between remote point-of-sale devices and the servers that process these requests in the financial services industry.

That these barriers of scale and security have been solved, in the short term, and under immense pressure to expand capabilities at a rapid pace, has led to the need to adopt older working protocols, that are computationally expensive, and have limited scalability.

Distributed unique key per transaction, DUKPT, was originally developed as a remote payment system by VISA, in the nineteen-eighties. It was an effective way to check credit availability via a modem and terminal, and scalability and complexity were less an issue, as it was limited to a relatively select set of vendors that chose to install this equipment. This began to change in the 1990's, with the expansion of the online world, and the introduction of debit cards, which by the beginning of the twenty-first century rose in popularity as an alternative, more convenient way for the consumer to pay for goods and services. In the current day, electronic payment has almost completely replaced older forms of payment such as cheques and money orders, and in many places in the world is replacing paper currency as the most often used method of payment.

We have seen a tremendous increase in the use of electronic payment systems, and at some future time, these payment methods may completely replace paper currency. Yet, despite this tremendous increase in scale and the subsequent increase in cost and complexity required by this unprecedented growth of electronic payments, we are still using underlying security systems that are based on forty-year-old technology.

What we propose is a replacement for DUKPT; one that is more scalable to meet the ever-increasing needs of the electronic payment industry. This replacement is authenticated, capable of up to 512-bit security, and a fundamentally more secure architecture than DUKPT. It can be scaled to the massive demands we will face in the future, and outperforms DUKPT by huge margins, that will cut required transaction infrastructure costs by as much as seventy-five percent. What we define here with the HKDS distributed key system, is what we believe will be the future of remote financial transaction security.

1.1 Purpose

The HKDS key management protocol, utilized in conjunction with the Keccak family of message authentication code generators (KMAC), and extended output functions (SHAKE), is used to derive unique symmetric keys employed to protect messaging in a financial services environment. The security of a distributed key scheme is directly tied to the secure derivation of transaction keys, used to encrypt a message cryptogram. This specification presents a distributed key management protocol that generates unique transaction keys from a base terminal key, in such a way that:

1) The terminal does not retain information that could be used to reconstruct the key once the transaction has been completed (forward security).
2) The capture of the terminals state, does not provide enough information to construct future derived keys (predicative resistance).
3) The server can reconstruct the transaction key using a bonded number of cryptographic operations.

HKDS is a two-key system. It uses an embedded key on the client to encrypt token exchanges and as a portion of the key used to generate the transaction key cache. It also uses a token key; an ephemeral key derived by the server, encrypted and sent to the client, as the second part of the key used to initialize the PRF that generates the transaction key cache. There are numerous advantages to using two keys in this way:

1) The client's embedded device key need never be updated, the base token key can be updated instead, to inject new entropy into the system.

2) The client can produce a practically unlimited number of derived transaction keys, there is no upper limit, so long as the base token key is periodically refreshed (after many thousands of token derivations, or millions of transactions).

3) This method provides both forward secrecy, and predictive resistance. Even if the client's state is captured, the adversary will not be able to derive past key caches from the information contained in the state. Likewise, the adversary will not be able to derive future key caches based on the captured state alone.

4) The client's key can be changed, without changing the embedded key itself (which is usually stored in a tamper-proof module, and can only be changed via direct access to the terminal). The client's master key identity can be changed instead, pointing to a different master key, that derives the same embedded device key, but uses a different base secret token key.

5) Exceptional performance; HKDS is highly efficient, outperforming DUKPT-AES by 4 times the decryption speed with a 128-bit key, to as much as 8 times faster than DUKPT using a 256-bit key, and we believe if using embedded Keccak CPU instructions, the performance of HKDS can be vastly improved upon.

6) Security. HKDS can be implemented with 128, 256, and 512-bit security settings, and uses strong NIST standardized (KMAC) authentication, and SHAKE for key derivation.

2: Scope

This document describes the HKDS key distribution protocol, which is used to derive per transaction unique keys from an initial embedded device key on a terminal and recreate that transaction key on a transaction processing server. This document describes the generation of these unique keys on the terminal, the authentication of cryptograms encrypted with those keys, recreating those keys on the transaction processing server, the verification of cryptogram messages, and the messaging protocol used to transmit keys and messages between the terminal client and transaction server. This is a complete specification, describing the cryptographic primitives, the key derivation functions, and the complete client to server messaging paradigm.

2.1 Application

This protocol is intended for institutions that implement distributed key systems to encrypt and authenticate secret information exchanged between remote terminals and transaction processing servers, such as ATMs, point-of-sale devices, or other technology commonly employed by the financial services industry that are used for remote payment transactions.

The key derivation functions, authentication and encryption of messages, and message exchanges between terminals and transaction servers defined in this document must be considered as mandatory elements in the construction of an HKDS key distribution system. Components that are not necessarily mandatory, but are the recommended settings or usage of the protocol shall be denoted by the key-word **SHOULD**. In circumstances where strict conformance to implementation procedures are required but not necessarily obvious, the key-word **SHALL** will be used to indicate compulsory compliance is required to conform to the specification.

3: References

3.1 Normative References

The following documents serve as references for key components of HKDS:

1. NIST FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions
2. NIST SP 800-185: Derived Functions cSHAKE, KMAC, TupleHash and ParallelHash
3. NIST SP 800-90A: Recommendation for Random Number Generation
4. NIST SP 800-108: Recommendation for Key Derivation using Pseudorandom Functions
5. ANS X9.8-1: Personal Identification Number (PIN) Management and Security
6. ISO 16609: Banking Requirements for Message Authentication using Symmetric Techniques
7. ISO 8583: Bankcard Originated Messages Interchange Message Specifications
8. ISO 9797-1: Information technology Security techniques Message Authentication Codes

3.2 Reference Links

1. HKDS C implementation: https://github.com/Steppenwolfe65/HKDS
2. The CEX++ Cryptographic library: https://github.com/Steppenwolfe65/CEX
3. The Keccak Code Package: https://github.com/XKCP/XKCP
4. Cryptographic Sponge Functions: https://keccak.team/files/CSF-0.1.pdf
5. SIMD Extensions for Keccak: https://dl.acm.org/doi/10.1145/2948618.2948622

4: Terms and Definitions

4.1 SHA-3 The Keccak hash function

The SHA3 hash function NIST standard, as defined in the NIST standards document FIPS-202; SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

4.2 SHAKE

The NIST standard Extended Output Function (XOF) defined in the SHA-3 standard publication FIPS-202; SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

4.3 KMAC

The SHA3 derived Message Authentication Code generator (MAC) function defined in NIST special publication SP800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash.

4.4 MDK

The Master Derivation Key structure. This key-set and unique key identifier is maintained by the transaction server. The MDK is composed of the BDK, and the STK random keys, and the KID, the MDK's unique identifier string.

4.5 BDK

The Base Derivation Key; part of the MDK key-set. The BDK is distributed to point-of-sale device manufacturers, who use this key to derive unique embedded keys for each device they produce.

4.6 STK

The Secret Token Key; a part of the MDK, and known only to the transaction processing server. This key is a secret known only to the financial institution processing requests, and used to create a secondary token-key unique to every terminal device, that is used to construct the terminals key-cache.

4.7 KID

The MDK's unique identifier string. This string is a 32-bit identification number used by the transaction processing server to lookup the correct MDK for a device.

4.8 KSN

The Key Serial Number. This is a 128-bit unique identifier string and counter composition. Each terminal device contains a unique KSN, comprised of the MDK's KID, the manufacturer identity string, the terminals device identity string (DID), and a 32-bit transaction counter.

4.9 DID

The terminal device's unique identification string. The DID is a multi-part identification string. The first half of the 64-bit string are the manufacturer identity string and protocol flags, and the second half, the terminals unique 32-bit identity string.

4.10 KTC

The Key Transaction Counter. The terminals internal transaction counter, an unsigned 32-bit monotonic counter that increments every time a transaction key is used from the time the terminal is first initialized.

4.11 EDK

The Embedded Device Key. The unique derivation key that is securely embedded on each terminal device, and derived from the BDK. The EDK is set for the lifetime of the terminal device, and need never be updated, regardless how many transaction keys are generated by the terminal.

4.12 TKC

The Transaction Key Cache. The set of keys residing on the terminal device. The size of the key cache can be changed using the *key cache multiplier*, but by default is set to: 42 * 128-bit keys at a 128-bit security setting, 34 * 128-bit keys as a 256-bit security setting, and 21 * 128-bit keys at a 512-bit security setting.

4.13 TSM

The Token Mac String. A string used to initialize the KMAC state. Used in token exchange authentication. This string is passed to the KMAC customization parameter, and initializes the KMAC state to a unique set of values. The TSM is derived by concatenating the client's KSN and the KMAC formal algorithm name.

4.14 CTOK

The Custom Token string. Used in the client token-key calculation, and consists of the implementation name, the client's DID, and the token request counter, which is the transaction key counter *divided* by the key cache size.

4.15 PRF

The Pseudo Random Function. HKDS uses the SHAKE XOF function as the primary PRF.

4.16 Key Stream

A key-stream is a stream of pseudo-random bytes that is XOR'd with a plaintext to create a cipher-text output in a stream cipher configuration.

4.17 Cryptographic Key

An array of random or pseudo-random bytes that are used to key a cryptographic function, such as a MAC or symmetric cipher.

4.18 Cryptographic Strength

The computational cost of the best-known attack against a cryptographic function or key size.

4.19 Derivation

A cryptographic process that is used to derive one value from another, as in using one cryptographic key, to derive a second cryptographic key.

4.20 Personal Account Number (PAN)

An identification number, typically signifying an account number used by the card holder and issued by a financial institution.

4.21 Personal Identification Number (PIN)

A secret identification number, a shared secret used by an account holder and issued by a financial institution. Used to validate authorized account usage.

4.22 Secure Cryptographic Device (SCD)

A device that provides both physically and logically protected cryptographic services and storage. The SCD module may be integrated into a terminal device used to remotely transact financial services, such as a point-of-sale device (POS), or an automated teller machine (ATM).

5: Structures

5.1 The Master Derivation Key

The master key (MDK) is comprised of three unique byte arrays;

1. The Base Derivation Key (BDK), this is the key that is distributed to the point-of-sale manufacturers, and used to generate a unique embedded device key (EDK) for each device.
2. The Secret Token Key (STK), this is a secret key, known only to the transaction server; a derivation of this key is mixed with the devices embedded device key to seed the PRF which generates the transaction key cache.
3. The Key Identity, this is a 4-byte unique id number that is sent as part of the key serial number during a transaction request, and is used to identify the master key.

The Key sizes are variable according to the desired security strength of the implementation. Both the base derivation key, and the secret token key are set to that security level. HKDS has three possible security levels, 128, 256, and 512-bit security. The key size values illustrated are in bytes; 8-bit unsigned integers.

The **hkds_master_key** contains the master key structure.

Vector	128-bit	256-bit	512-bit
BDK	16	32	64
STK	16	32	64
KID	4	4	4

Table 5.1: The Master Key component array byte sizes per the security level implemented.

5.2 The Key Serial Number

The key serial number (KSN) is sent by the client along with an encrypted message to the transaction processing server. The KSN is a multi-part 16-byte array, consisting of the master key id (KID), the devices identity string (DID), and the key transaction counter (KTC).

The KID is a 4-byte unique identification number that corresponds to a master key.

The DID is comprised of four parts; a 1-byte protocol id number, a 1-byte device mode number, a 2-byte manufacturer id, and a 4-byte unique device identification number.

The transaction counter, is a 4-byte unsigned monotonic counter that represents the number of transactions processed by that device at the time of transmission.

Master Key ID	Device Identity	Transaction Counter

4 bytes	8 bytes	4 bytes

Figure 5.2a: The key serial number structure.

Protocol ID 1 byte	Device Mode 1 byte	Manufacturer ID 2 bytes	Device ID 4 bytes

Figure 5.2b: The device identity structure.

Protocol ID possible values:

1) Standard encryption mode: 0x10 (17)
2) Authenticated encryption mode using KMAC: 0x11 (18)
3) Authenticated encryption mode using SHA3: 0x12 (19) [not implemented]

The protocol ID reflects a versioning number as well as the base encryption mode (authenticated or standard encryption mode). Future revisions of the protocol will reflect a change in this numbering, with subsequent revisions incrementing these values in an extended numbering from this base protocol specification.

Device Mode possible values:

1) SHAKE-128 PRF mode: 0x09 (9)
2) SHAKE-256 PRF mode: 0x0a (10)
3) SHAKE-512 PRF mode: 0x0b (11)

The device mode identifier indicates the primary pseudo random function (PRF) used in the encryption and derivation-key generation functions. Additions of alternate PRF functions **SHALL** be added to subsequent versions of the protocol incrementing the device mode identifier from these base values.

Manufacturer ID possible values:

The manufacturer identity string is assigned to the point-of-sale device manufacturer, this is a unique identifying number, that **SHALL NOT** be assigned to any other device manufacturer. The manufacturer ID **SHALL** be a statically assigned identifier indicating the point-of-sale device's manufacturing company, this value **SHALL NOT** be changed, and can only be reassigned through general industry consensus and in the event the company's ownership or operating status has changed. The two-byte numbering scheme allows for more than 16 thousand uniquely numbered device manufacturers. In the unlikely event this number is exceeded, the first byte of the device identity, may be used to extend the manufacturer's identity address space in a future specification of the protocol as required.

Device ID possible values:

The device ID is a unique identifier that is used to number the point-of-sale device supplied by the POS device manufacturer. This is a monotonic incrementing counter, used to assign a unique identity to each individual POS device. Each device produced **SHALL** be programmed with a unique device identity. The same device identity **SHALL NOT** be assigned to more than one device. More than 4 billion devices can be numbered per each device manufacturer. In the unlikely event that a manufacturer has exhausted this address space, the manufacturer should begin renumbering again at zero, after first ensuring that earlier devices that may retain this identity, have been removed from service, OR, apply for a secondary manufacturing identification number.

5.3 HKDS Packet Header

The **hkds_packet_header** contains the HKDS packet header structure.

Data Name	Data Type	Bit Length	Function
flag	hkds_packet_type	0x08	The type of packet
protocol	hkds_protocol_id	0x08	The protocol id
sequence	Uint64	0x40	The packet sequence number
length	Uint64	Variable	The packet size including header

Table 5.3a HKDS packet header structure.

The HKDS packet header is 4 bytes in length, and contains:

1. The **Flag**, the type of message contained in the packet; Token Request (0x01), Token Response (0x02), Message Request (0x03), Message Response (0x04), Administrative Message (0x05), or, an Error Message (0x1F-0xFF).
2. The **Protocol ID**, this signals the security mode of HKDS being used differentiated by the underlying pseudo-random function; SHAKE-128 (0x09), SHAKE-256 (0x0A), or SHAKE-512 (0x0B). This value **SHALL** correspond to the KSN protocol identity (PID).
3. The **Packet Sequence**, this indicates the sequence number of the packet exchange.
4. The **Packet Size**, this includes the message header and the message payload.

Flag **1 byte**	**Protocol ID** **1 byte**	**Packet Sequence** **1 byte**	**Packet Size** **1 byte**
Message **Maximum 64 bytes**			

Figure 5.3b: The message packet structure.

Client Token Request

The client token request is sent to the server on initialization and subsequently each time the transaction key cache has been exhausted. It includes the client's key serial number (KSN); a concatenation of the master key identity string, the client's unique identification number, and the transaction key counter. The transaction counter, along with the KSN, are used by the server's token authentication mechanism, to generate a unique message authentication code for each token request.

Flag	**Protocol ID**	**Packet Sequence**	**Packet Size**
0x01	**0x09|0x0A|0x0B**	**0x01**	**20 bytes**
KSN **16 bytes**			

Figure 5.3c: The client token packet structure.

Server Token Response

The server's response to a token request. This packet contains an encrypted token (ETOK) sent from the server to the client device. The token is the same size as the embedded device key on the client (EDK), plus a MAC authentication tag. A 16-byte authentication code is appended to the encrypted token, which is verified by the client.

Flag	**Protocol ID**	**Packet Sequence**	**Packet Size**
0x02	**0x09|0x0A|0x0B**	**0x02**	**36/52/84 bytes**
Encrypted Token **{ 32, 48, 80 } bytes**			

Figure 5.3d: The server token response packet structure.

Client Message Request

The client's encrypted message request packet. This packet includes 16 bytes of encrypted message and the client's key serial number, and may include the authentication tag as indicated by the Protocol ID flag.

<table>
<tr><th>Flag</th><th>Protocol ID</th><th>Packet Sequence</th><th>Packet Size</th></tr>
<tr><td>0x03</td><td>0x09|0x0A|0x0B</td><td>0x01</td><td>36/52 bytes</td></tr>
<tr><td colspan="4">KSN – 16 bytes
Encrypted Message – 16 bytes
[Optional] Authentication tag – 16 bytes</td></tr>
</table>

Figure 5.3e: The client request message packet structure.

Server Message Response

The server's plaintext message response, typically a verification response sent to the client terminal that indicates the success or failure of a transaction request. Note that future revisions may append a MAC code used to verify this response.

<table>
<tr><th>Flag</th><th>Protocol ID</th><th>Packet Sequence</th><th>Packet Size</th></tr>
<tr><td>0x04</td><td>0x09|0x0A|0x0B</td><td>0x02</td><td>20 bytes</td></tr>
<tr><td colspan="4">Server Response – 16 bytes</td></tr>
</table>

Figure 5.3f: The server message response structure.

Administrative Message

An administrative message is used to signal requests, status updates, or as a post-error condition reset of a communications session. These signals invoke specific handling measures by the receiver, can indicate an internal state of the device, and are used for logging or administrative action. See section 5.4: the Administrative Message enumeration for details.

<table>
<tr><th>Flag</th><th>Protocol ID</th><th>Packet Sequence</th><th>Packet Size</th></tr>
<tr><td>0x05</td><td>0x09|0x0A|0x0B</td><td>0x00-0x03</td><td>6 bytes</td></tr>
<tr><td colspan="4">Administrative Message Code – 2 bytes</td></tr>
</table>

Figure 5.3g: The administrative message packet structure.

Error Message

An error message indicates a serious failure between the client or server. This message is bi-directional and can be sent by either the client or server. The base error code (0x1F-0xFF) is contained in the sequence number.

<table>
<tr><th>Flag
0xFF</th><th>Protocol
0x09|0x0A|0x0B</th><th>Packet
0x1F-0xFF</th><th>Packet Size
20 bytes</th></tr>
<tr><td colspan="4"><b>Error Message – 16 bytes</b></td></tr>
</table>

Figure 5.3h: The error message packet structure.

5.4 Message Types

The following are a preliminary list of message types used by HKDS:

Message Name	Numerical Value	Message Purpose
Synchronize Token	0x01	Sent by the client in the event that a token key received from the server fails an authentication check, and is rejected by the client. This message triggers an increment of the transaction counter on the client by the key cache size, and the amended KSN is sent along with a new token request.
Reinitialized Token	0x02	This message is the server's [optional] response to the client token key rejection, acknowledging a new token is being constructed. The maximum concurrent token re-initialization requests are **3**, after three attempts the server **SHALL** consider the client device compromised and ignores all subsequent transactions, until a client re-initialization has been performed. The server **SHALL** log all client token rejections, and if tokens are rejected beyond a weighted-fair threshold, the client should be disconnected and assumed to be compromised. For example, if within 1000 requests, there are 6 token failures, the server should consider the device or communications channel as

		compromised, and raise an administrative alert. If after 1000 successful token exchanges, have occurred, the failed token counter is reset to zero.
Token Requests Exceeded	0x03	In the event that the maximum number of token failures have occurred within a given threshold, the server will send the token-request exceeded message to the client. Upon receiving this notification, the terminal device **SHALL** enter a state of 'error condition', whereby, the terminal ceases all transactions capabilities, and awaits a remote reset.
Remote Reset	0x04	The server can send a remote reset to the client terminal, only if the terminal is in an 'error condition' state. The remote reset purges the current key cache, incrementing the transaction counter to the start of a new cache cycle. The client then sends a new token request packet, generates a new transaction key cache, and resynchronizes with the server.
Diagnostic	0x05	The server can request a diagnostic output from the terminal's hardware components. This output should be encrypted, by using an available key in the cache to key the PRF, generate a key-stream, and use the pseudo-random output to encrypt (XOR) the diagnostic output.
Reserved	0x06	
Reserved	0x07	
Reserved	0x08	

Table 5.4: Packet header message types.

6: Operational Overview

6.1 Client Initialization

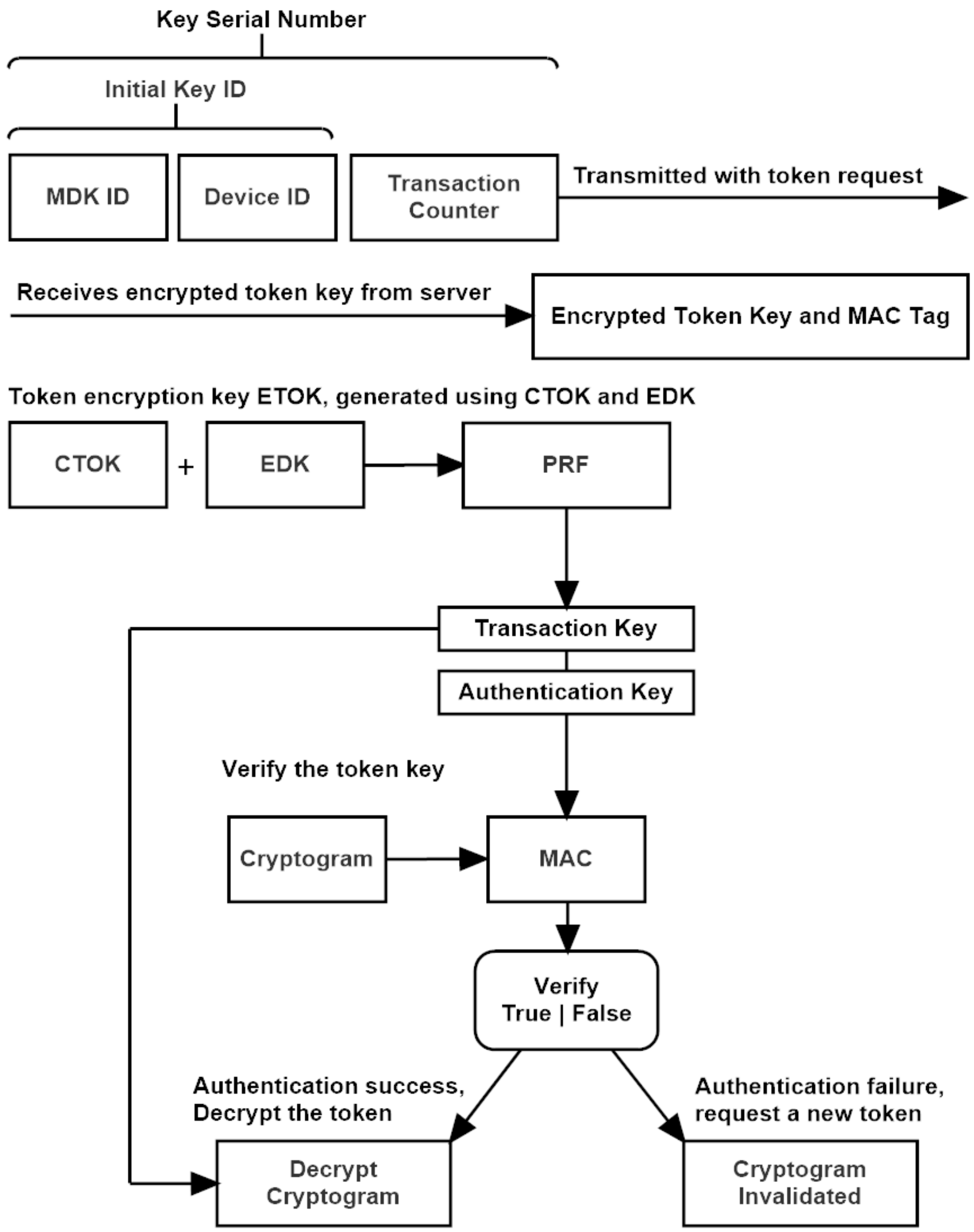
Key Serial Number
Initial Key ID
MDK ID
Device ID
Transaction Counter
Transmitted with token request
Receives encrypted token key from server
Encrypted Token Key and MAC Tag
Token encryption key ETOK, generated using CTOK and EDK
CTOK
+
EDK
PRF
Transaction Key
Authentication Key
Verify the token key
Cryptogram
MAC
Verify True | False
Authentication success, Decrypt the token
Authentication failure, request a new token
Decrypt Cryptogram
Cryptogram Invalidated

Figure 6.1a: HKDS client device initialization.

The client terminal device must undergo initialization before being ready to process transaction requests. The client begins the initialization by sending a token request packet to the transaction server, requesting an encrypted token key.

This token key is generated by the server, using the secret token key (STK) residing in the associated master derivation key (MDK), stored on the server. The STK from that master key is concatenated with the custom token string (CTOK); the client's unique device identity string (DID), the implementation name, and the token request counter. The concatenated array of CTOK and STK is used to key the PRF.

The transaction counter, is a monotonic counter used by the client to count the number of transaction keys that have been used, this counter is *divided* by the size of the client's key cache, to calculate the token requests counter; the number of times the client has requested a token from the server. Using this counter as a part of the PRF key, guarantees that a unique token is generated each time, and using the client's unique device identity, guarantees that each token generated by the server using that STK, is unique to the requesting client, so that combined these inputs provide a guarantee that every unique client, will receive a different token, every time a token request is made, while the secret token key, remains known only to the server.

6.2 Client Message Encryption

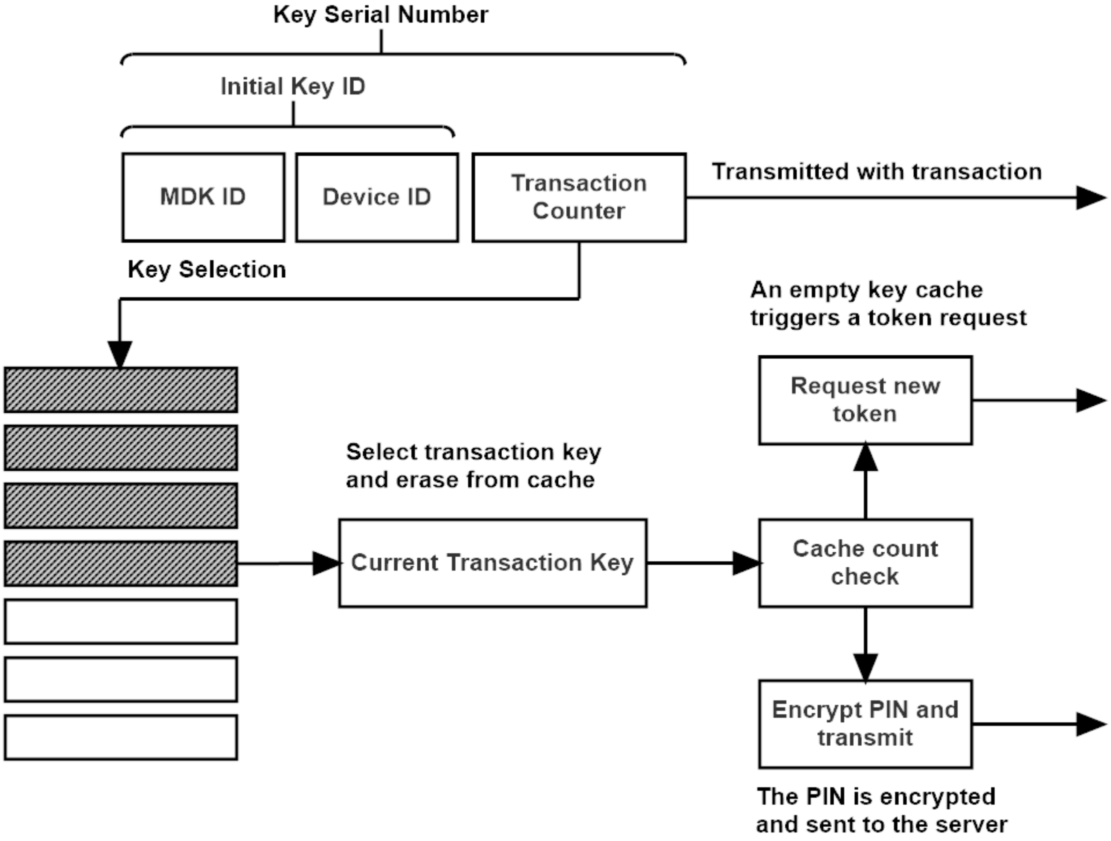

Figure 6.2a: HKDS client message encryption.

The client encrypts a message by selecting a transaction key from its internal key cache, using that pseudo-random key in a stream cipher like application, XORing the PIN message with the pseudo-random transaction key to produce the cipher-text. When a transaction key is selected, the cache is first checked for available keys remaining in the cache. If the transaction key cache is empty, the client sends a token request to the server, and rebuilds the key cache before the transaction can begin.

6.3 Server processing a Client token request

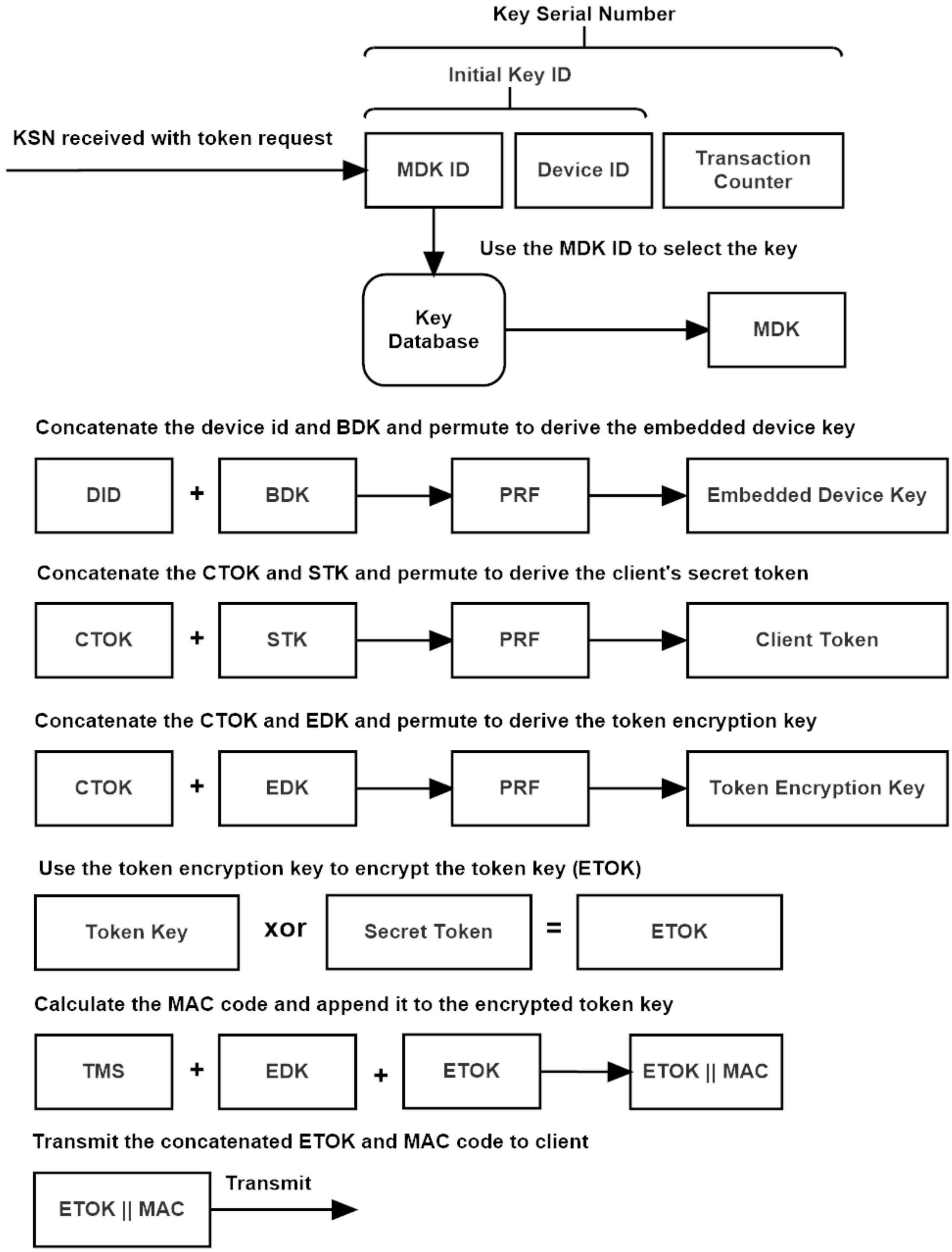
Key Serial Number
Initial Key ID
KSN received with token request
MDK ID
Device ID
Transaction Counter
Use the MDK ID to select the key
Key Database
MDK
Concatenate the device id and BDK and permute to derive the embedded device key
DID
+
BDK
PRF
Embedded Device Key
Concatenate the CTOK and STK and permute to derive the client's secret token
CTOK
+
STK
PRF
Client Token
Concatenate the CTOK and EDK and permute to derive the token encryption key
CTOK
+
EDK
PRF
Token Encryption Key
Use the token encryption key to encrypt the token key (ETOK)
Token Key
xor
Secret Token
=
ETOK
Calculate the MAC code and append it to the encrypted token key
TMS
+
EDK
+
ETOK
ETOK || MAC
Transmit the concatenated ETOK and MAC code to client
ETOK || MAC
Transmit

Figure 6.3a: Server token request processing.

The server derives the client’s embedded device key by concatenating the device identity string (DID), and the master key’s base derivation key (BDK), and permuting them to derive the EDK. The server generates the client’s secret token by concatenating the custom token string (CTOK), and the master key’s secret token key (STK), and permuting them to create the token. The token is encrypted with the token encryption key, which is derived by concatenating the custom token string, with the client’s EDK, and permuting them to generate the token encryption key. The server then encrypts the token (ETOK) using the token encryption key.

The encrypted token key is then processed by the MAC function, and a 16-byte authentication code is appended to the encrypted token. The MAC state is first initialized to a unique value by hashing a combination of the client’s KSN, and the MAC functions formal name (the token MAC string; TSM), then keyed with the EDK, which generates the MAC code using the encrypted token as the message. This generates a unique MAC code for every token generated by the server; the transaction counter in the client’s KSN, guarantees that the MAC state will be initialized to a unique value each time. It also serves to authenticate the KSN, along with the encrypted token key.

6.4 Client Key Cache Generation

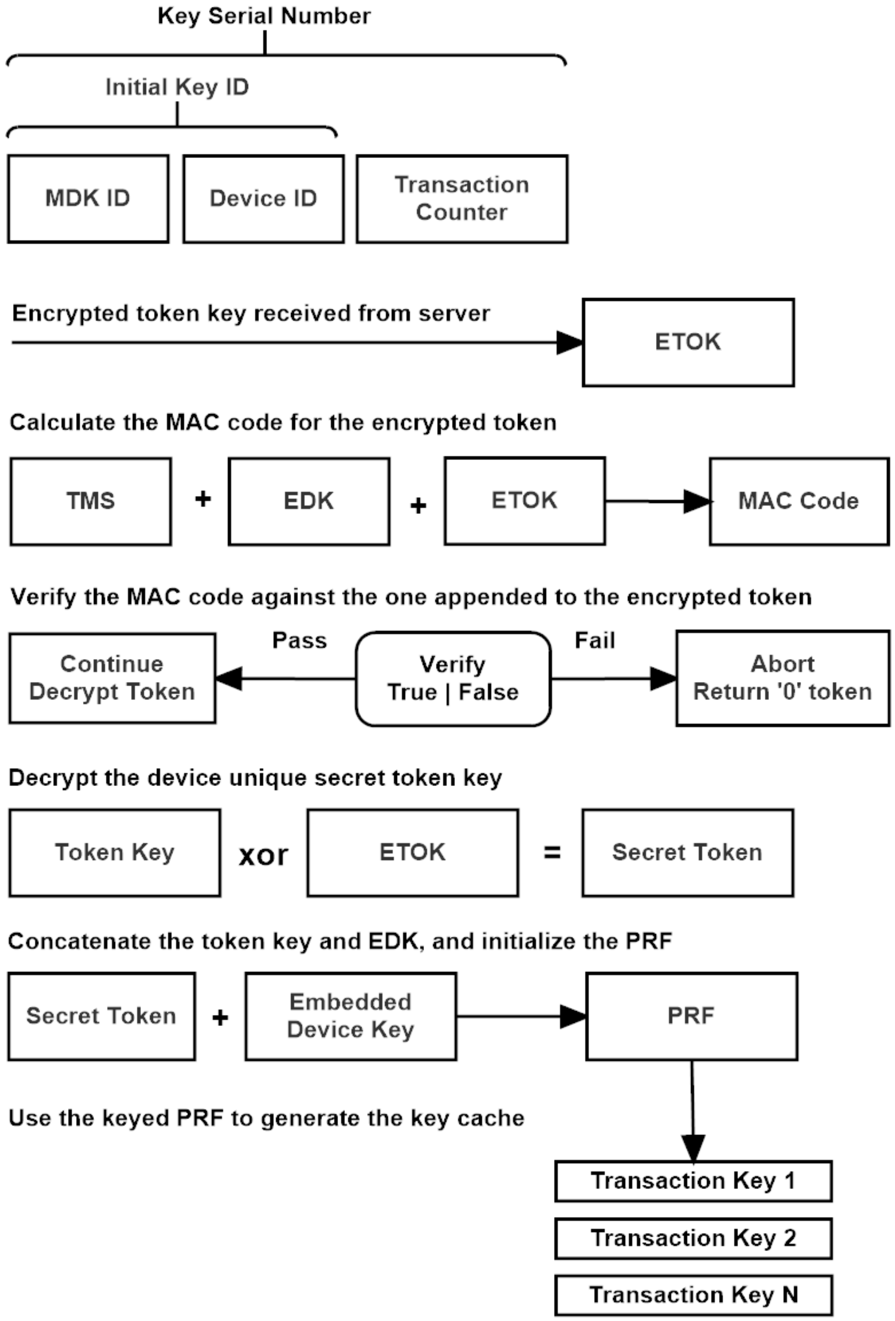
Key Serial Number
Initial Key ID
MDK ID
Device ID
Transaction Counter
Encrypted token key received from server
ETOK
Calculate the MAC code for the encrypted token
TMS
+
EDK
+
ETOK
MAC Code
Verify the MAC code against the one appended to the encrypted token
Continue Decrypt Token
Pass
Verify True | False
Fail
Abort Return '0' token
Decrypt the device unique secret token key
Token Key
xor
ETOK
=
Secret Token
Concatenate the token key and EDK, and initialize the PRF
Secret Token
+
Embedded Device Key
PRF
Use the keyed PRF to generate the key cache
Transaction Key 1
Transaction Key 2
Transaction Key N

Figure 6.4a: Client key cache generation.

When the client has received the encrypted token response from the server, it generates a MAC code for the encrypted cipher-text (ETOK). The MAC function is first set to a unique state, using the Token MAC String (TMS). The TMS is a concatenation of the client's KSN and the MAC functions formal name (ex. 'KMAC256'). The TMS string is passed to the KMAC customization parameter, and the client's EDK is used to key KMAC. The encrypted token is passed to KMAC as the message parameter, and a 16-byte MAC codes is generated.

This code is compared to the MAC code appended to the encrypted token key. If the MAC codes are identical, the client decrypts the token, and uses it to generate the key cache. If verification fails, the token is zeroed, and the function returns a failure state, and the token request is either sent again to the server up to a maximum retry count, or the client signals a critical failure to the server, and awaits reset.

Once the encrypted token is authenticated successfully, the client then decrypts the token, and concatenates the token with its embedded device key (EDK) to key SHAKE. The SHAKE implementation then 'squeezes' the number of blocks of pseudo-random required to populate the client's key cache.

The SHAKE squeeze function applies a permutation to 1600 bits of state, a portion of that state (2n the desired security size) remains secret, the rest of the state is the pseudo-random output. This allows for a large block of usable pseudo-random output that can be applied to building the key cache, with each call to the underlying permutation function.

Using the 128-bit security setting, 1344 bits (168 bytes) of the state can be used to build keys with each permutation call. 256-bit security produces 1088 bits (136 bytes) of pseudo random, and a 512-bit security setting returns 576 bits (72 bytes) of pseudo random per permutation call.

The HKDS key cache size is calculated using a 2n multiple of these return sizes. This is because PIN message sizes are typically set at 128-bits, or 16 bytes, and 16 bytes fits into 2n Keccak squeeze function return output size evenly. For example, the minimum *key cache multiplier* is 2, using the 128-bit security setting, 2 calls to the Keccak squeeze function with a rate of 168, returns 336 bytes of usable pseudo-random, or exactly 21 * 16-byte transaction keys.

The default setting for the cache multiplier is 4, which produces exactly 42 transaction keys with the 128-bit security setting, 34 transaction keys with the 256-bit security setting, or 18 transaction keys with the 512-bit security implementation.

The key-cache multiplier is an implementation definable constant, and can be set by adjusting the HKDS_CACHE_MULTIPLIER constant within the code, but both the client and server must be set to the same multiplier. The valid settings for the key cache multiplier are 2, 4, 6, 8, 10, 12, 14, and 16. Changing the multiplier increases the size of the internal key cache of the client by a factor of 2n the base permutation output size, but this also increases the time required to recreate the clients key cache on the server. This is why we have set the default key cache multiplier to 4, which we feel is a good trade-off between token exchange intervals, and increased latency caused by larger key-cache reconstructions during server-side decryption.

6.5 Server Transaction Key Derivation

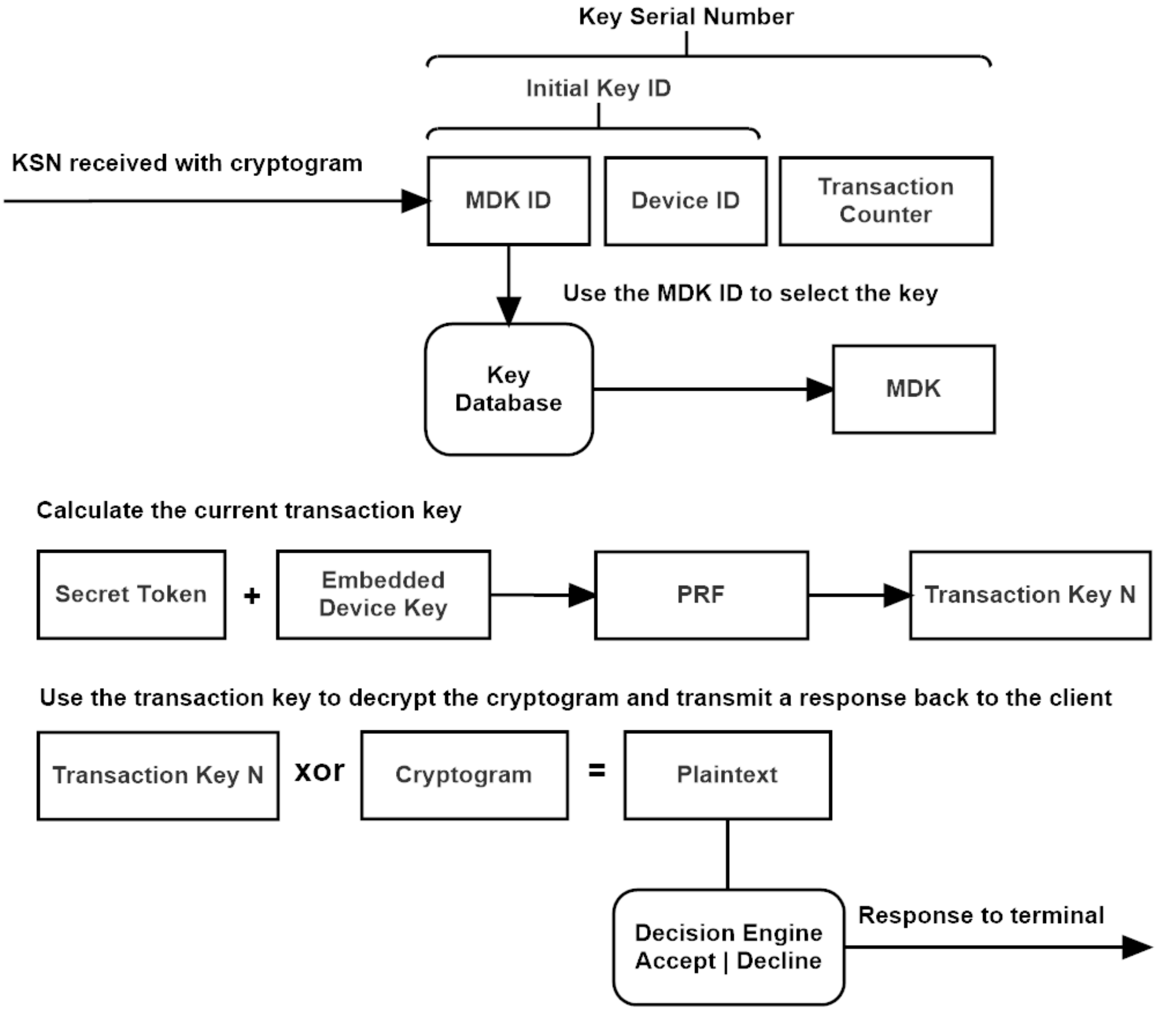
Key Serial Number
Initial Key ID
KSN received with cryptogram
MDK ID
Device ID
Transaction Counter
Use the MDK ID to select the key
Key Database
MDK
Calculate the current transaction key
Secret Token
+
Embedded Device Key
PRF
Transaction Key N
Use the transaction key to decrypt the cryptogram and transmit a response back to the client
Transaction Key N
xor
Cryptogram
=
Plaintext
Decision Engine Accept | Decline
Response to terminal

Figure 6.5a: Server transaction key generation.

The server receives the KSN from the client, and uses the KID to select the correct master key from the key database. The client's EDK is derived by permuting the concatenated DID and BDK. The client's token is derived by permuting the concatenated CTOK and STK.

The server concatenates the EDK and the secret token to key the PRF, which generates the client's transaction key cache. The server will only generate keys up to the permutation block containing the requested transaction key, performing the minimum number of permutations calls necessary to extract the target key. The server then XORs the transaction key with the cipher-text, decrypting the clients PIN, and passing the message to the server's PIN verification and transaction decision engine.

6.6 Client Encryption and Authentication

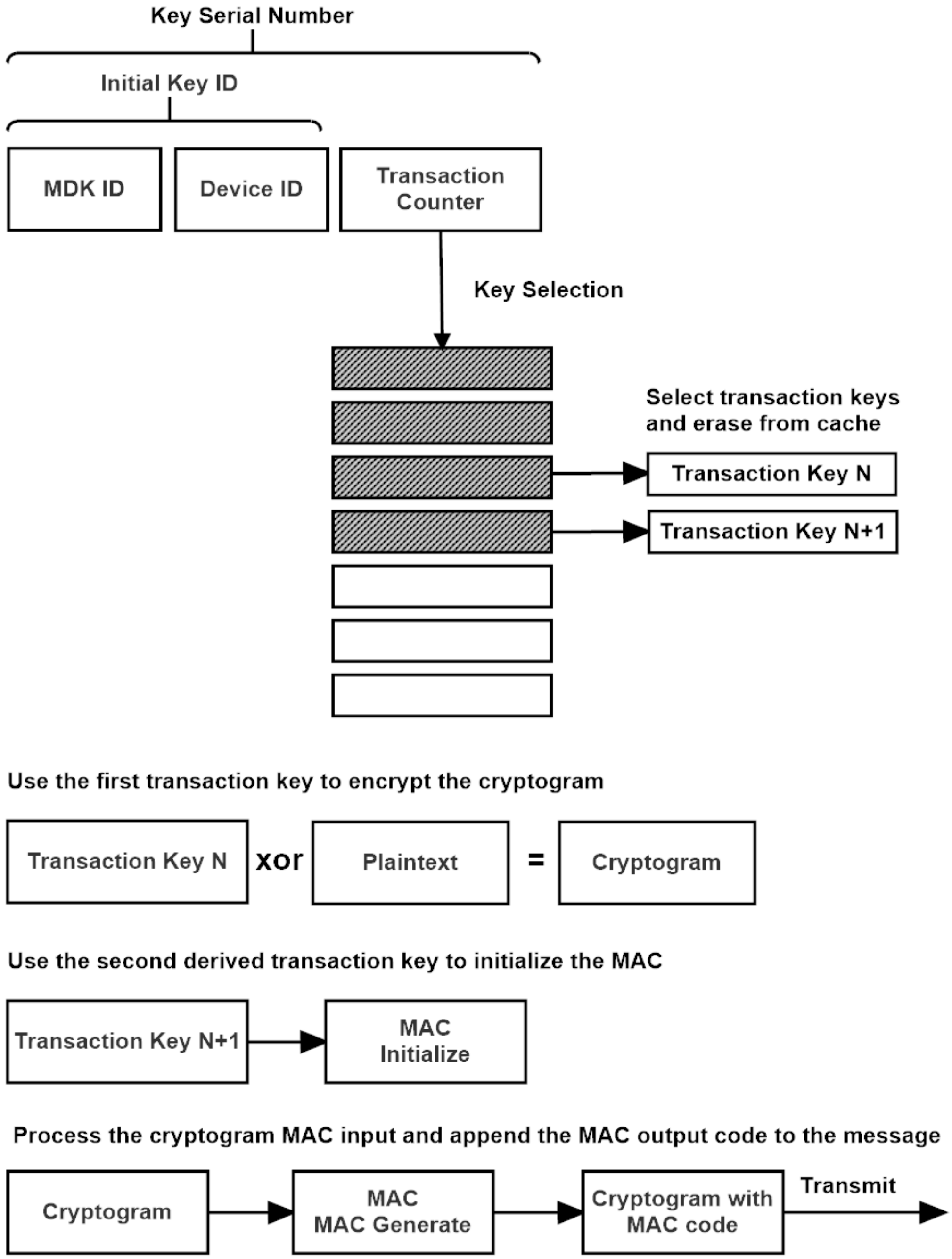

Figure 6.6a: Client encryption and authentication.

The client uses two transaction keys taken from the key cache to encrypt and the authenticate a message. The first key is XOR’d with the message to produce the cipher-text. The second key is used to key KMAC, which takes the cipher-text as an input and produces a MAC code. The MAC code is then appended to the cipher-text and sent to the server.

6.7 Server Verification and Decryption

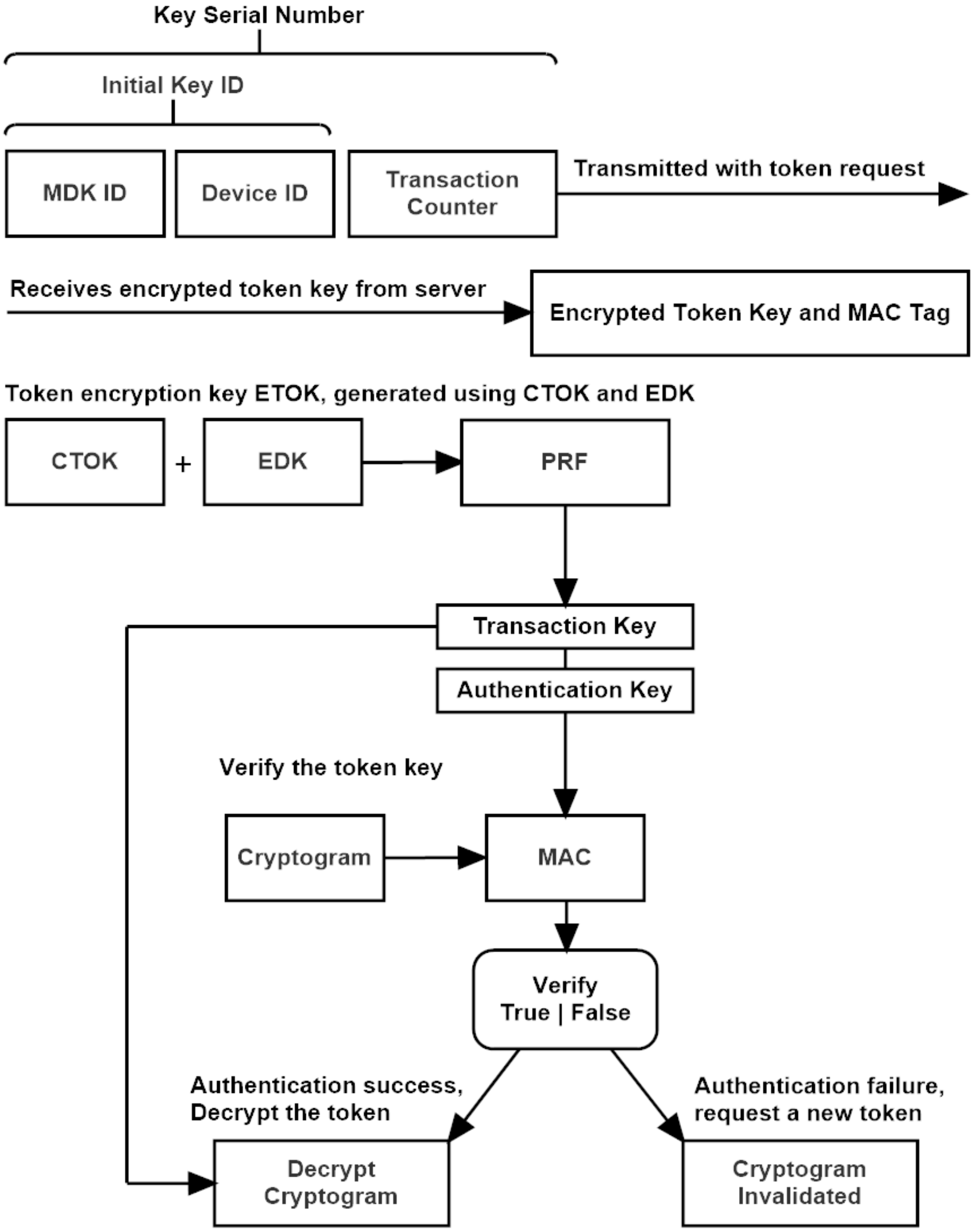

Figure 6.7a: Server verification and message decryption.

The server receives the authenticated cryptogram, and the client's KSN. It uses the KID to select the master key associated with the client terminal. The server then derives the client's embedded device key, and the client's token key, uses them to key the PRF and reconstruct the client's key cache.

The server extracts two keys from the cache, the first key is later used to decrypt the cipher-text, the second key is used to key KMAC and verify the cipher-text. Only if the new code generated by KMAC for the cipher-text, matches the MAC code appended to the cipher-text, is the key decrypted.

If the decryption succeeds, the cipher-text PIN is decrypted and the message is passed to the transaction servers decision engine, and the client is then notified of the transaction outcome (accept or decline). If the authentication fails, the server sends a message to the client, requesting a new cryptogram. The client **SHALL** then generate a new authenticated cryptogram. The server **SHALL** log this event, and treat repeated failures as a potential compromise of the client device and/or its communications channel.

7: Formal Description

Legend

← ↔ →	- Assignment and direction symbols
:=, !=, ?=	- Equality operators; assign, not equals, evaluate
bdk	- The Base Derivation Key array
C	- Client
did	- The clients Device Identity string
edk	- The embedded device key array
etok	- Encrypted token
fn	- The implementations formal name
kid	- The master Key Identity string
ks	- A pseudo-random array generated by the PRF
ksn	- Key serial number
mdk	- The Master Derivation Key structure
P	- Permutation function: SHAKE
PRF	- Pseudo random bytes generator: SHAKE
S	- Server
stk	- The Secret Token Key array
tk	- The transaction key used to encrypt or decrypt a message
tkc	- The Transaction Key Cache, a set of key stream arrays used to encrypt messages
tok	- The temporary token key
tsm	- The token MAC string
TR	- Token request
trc	- Token request counter
x	- A plain-text message
y	- A cipher-text message
||	- Concatenate two arrays or strings
⊕	- A bitwise XOR

7.1 Master Key Generation

The master key structure (MKD); base derivation key (BDK), and secret token key (STK), **SHOULD** be generated using a true random number generator (TRNG).

The RNG used in the implementation is ACP, which uses the system RNG along with various timers and system unique values, CPU and memory jitter, is used to key an instance of cSHAKE-512 which produces the pseudo-random output. If a true random number generator is not available, it is recommended that these keys are derived with a high-quality random generator such as ACP.

The key identity (KID) is intended as a database identity reference, it should be a monotonic counter, unique for each master key set.

7.2 Token Key Exchange

The client initializes the exchange by sending a token request packet to the server, which includes its key serial number (KSN).

The server looks up the master key identity (KID), the first 4 bytes of the KSN, and matches it to the instance of the master derivation key (MDK) associated with that device.

The server then combines the implementation name, the client's device identity (DID), and the token-request counter to form the custom token string (CTOK). The token-request counter is calculated from the KSNs transaction counter, the last 4 bytes of the KSN, *divided* by the size of the transaction key cache.

The CTOK is concatenated with the secret token key (STK) and permuted, creating the new device token.

The server derives the clients embedded device key (EDK) by concatenating the DID with the base derivation key (BDK) and permuting the array, outputting the EDK. The CTOK is concatenated with the EDK and permuted to create the token encryption key.

The token is XOR'd with the token encryption key, a MAC code is generated and appended to the key, and it is sent back to the client.

The client then executes these steps in reverse; generating the MAC key using the EDK and TMS, authenticating the cipher-text, then creating the token customization string, and concatenating that with its EDK and permuting the array, to create the token encryption key.

Once authenticated, the client decrypts the token using an XOR of the token encryption key and the encrypted token (ETOK).

The token is then concatenated with the EDK, and used to key the PRF, which generates the client's key cache.

7.3 Client Request:

The exchange begins with the client sending the server a token request packet along with the clients KSN.

$C \rightarrow S\{x\}TR_{ksn}$

Server Response:

The server creates the custom token string (CTOK), using the token request counter, concatenated with the algorithm's formal name, and the device id string.

a) $ctok \leftarrow (trc \,\|\, fn \,\|\, did)$

The server then generates the unique secret token, by concatenating the CTOK string and the secret token key (STK), and permuting them to create a unique token key.

b) $tok \leftarrow \mathrm{P}(ctok \| stk)$

The server derives the devices embedded device key (EDK), by concatenating the devices identity string (DID), and the master key's base derivation key (BDK), and then permuting them to derive the EDK.

c) $edk \leftarrow \mathrm{P}(did \| bdk)$

The server creates a key stream; a pseudo random array of bytes, by permuting the CTOK concatenated with the devices embedded device key (EDK).

d) $ks \leftarrow \mathrm{P}(ctok \| edk)$

The server encrypts the new token using a bitwise XOR of the key-stream and the secret token key.

e) $etok \leftarrow tok \oplus ks$

The MAC function state is initialized with the TMS, a concatenation of the client's KSN and the MAC functions formal name string.

f) $tms \leftarrow ksn \| mname$

The server then generates the MAC code used to authenticate the cipher-text, by initializing the MAC function state with the TMS. The EDK is used to key the function, which processes the encrypted token message, and outputs a MAC code, which is appended to the encrypted token key.

g) $etok \leftarrow etok \| \mathrm{M}(tms, edk, etok)$

The server then transmits the authenticated and encrypted token key to the client.

h) $\mathrm{S} \rightarrow \mathrm{C}\{y\}\mathrm{T}_k$

7.4 Client Response:

After receiving the encrypted and authenticated token, the client recreates the TMS.

a) $tms \leftarrow ksn \| fname$

The client then keys the MAC function using the TMS and EDK and generates a MAC code for the encrypted token key (ETOK)

b) $m \leftarrow \mathrm{M}(tms, edk, etok)$

The newly generated MAC code is compared to the MAC code appended to the ETOK cryptogram, if equivalent, the token is decrypted. If authentication fails the function returns a failure state, which is propagated up the application stack for handling.

c) $m1 \mathrel{?=} m2, true : false$

If the token is authenticated successfully, the client recreates the token customization string (CTOK) using the token request counter, the algorithm's formal name, and the devices identity string.

d) $ctok \leftarrow (trc \,||\, fn \,||\, did)$

The pseudo random key-stream is then generated by permuting the custom token concatenated with the embedded device key (EDK).

e) $ks \leftarrow \mathrm{P}(ctok \,||\, edk)$

The token is decrypted by XORing the encrypted token and the key-stream, producing the decrypted token.

f) $tok \leftarrow etok \oplus ks$

The client then concatenates the secret token key and the embedded key, to key the PRF, and generates the transaction key cache (TKC).

g) $tkc \leftarrow \mathrm{PRF}(tok \mid edk) \rightarrow \{k^1, k^2, \ldots k^n\}$

7.5 Transaction Key Cache Generation

The transaction key cache is generated each time the key cache is exhausted. The sequence starts when the client requests a secret token key from the server.

a) $\mathrm{C} \rightarrow \mathrm{S}\{x\}\mathrm{TR}_{ksn}$

The server responds by sending the encrypted token back to the client.

b) $\mathrm{S} \rightarrow \mathrm{C}\{y\}\mathrm{T}_k$

The client generates the key-stream used to decrypt the token by concatenating the custom token string (CTOK) and the embedded device key (EDK) and permuting them:

c) $ks \leftarrow \mathrm{P}(ctok \,||\, edk)$

The keystream is then used to decrypt the token:

d) $x \leftarrow -\mathrm{E}_k(y) = (tok = (etok \oplus ks))$

The token is concatenated with the embedded device key (EDK) and used to key the PRF, the token key is then erased to promote predictive resistance.

e) $tok = 0$

The keyed PRF is then used to generate the set of transaction cache keys:

f) $tkc \leftarrow \mathrm{PRF}(tok \,||\, edk) \rightarrow \{k^1, k^2, \ldots k^n\}$

7.6 Message Encryption

The client selects the next available transaction key (TK) from the transaction key cache (TKC), and increments the key transaction counter (KTC).

a) $tk \leftarrow (K_{ktc}),\ ktc = ktc + 1$

The message is encrypted using the transaction key.

b) $y \leftarrow \mathrm{E}_k(x) = (x \oplus tk)$

The transaction key is erased from the key cache to preserve forward security.

c) $tkc \leftarrow (K_{ktc} = 0)$

The encrypted message along with the client’s key serial number (KSN) are then sent to the server:

d) $C \rightarrow S\{y\}_{ksn}$

7.7 Message Decryption

The server receives the encrypted message and the KSN from the client, and concatenates the client’s device identity string (DID) and the base derivation key (BDK) and permutes them, to derive the clients embedded device key (EDK).

a) $edk \leftarrow P(did \| bdk)$

The server then uses the transaction counter in the KSN to calculate the token request counter, concatenates this with the algorithms formal name and the client’s device identity to create the custom token string:

b) $ctok \leftarrow (trc \| fn \| did)$

The custom token string (CTOK) is concatenated with the secret token key (STK) and permuted, to create the token key (TOK).

c) $tok \leftarrow P(ctok \| stk)$

The token key is concatenated with the EDK to key the PRF, which generates the transaction key cache:

d) $tkc \leftarrow PRF(tok \| edk) \rightarrow \{k^1, k^2, \ldots k^n\}$

The PRF only generates the number of keys up to the block containing the target transaction key. Once the correct transaction key is generated, it is used to decrypt the message:

e) $x \leftarrow -E_k(y) = (x = (y \oplus tk))$

8: Client Description

8.1 Client Constants

The client constants must remain static, unchanged in the implementation in order to match constants in the server implementation. The one constant that can be defined in the implementation, but must be set identically in both server and client, is the key cache multiplier, the constant that determines the number of key registers in the client's internal key cache. All constant values are defined as unsigned integers.

The constant names and format correspond to the C reference implementation, linked in the References section.

Constant Name	Value	Description
HKDS_CACHE_MULTIPLIER	4	The cache multiplier; the key cache total byte size is a multiple of this value times the output size of the PRF function.
HKDS_DID_SIZE	12	The size of the device identity string, 12 bytes.
HKDS_KID_SIZE	4	The size of the master key identity string, 4 bytes. The KID is part of the device identity string, and the MDK.
HKDS_KSN_SIZE	16	The size of the key serial number, 16 bytes. The key serial number is comprised of the device identity string, and the transaction key counter.
HKDS_MESSAGE_SIZE	16	The size of the HKDS encrypted message output, 16 bytes.
HKDS_NAME_SIZE	7	The size of the HKDS implementation name string, 7 bytes. The name string is a component in the CTOK token string.
HKDS_TKC_SIZE	4	The size of the key transaction counter, 4 bytes.
HKDS_TAG_SIZE	16	The size of the KMAC authentication tag, 16 bytes.
HKDS_SHAKE_XXX	0x09-0x0B	The SHAKE security mode used by the HKDS implementation. Possible values are 128=0x09, 256=0x0A, 512=0x0B.

HKDS_CTOK_SIZE	23	The byte length of the custom token string CTOK.
HKDS_NAME_SIZE	7	The byte length of the HKDS implementations formal name.
HKDS_PRF_RATE	168, 136, 72	The rate size of the SHAKE permutation function. Possible values are 168 for 128-bit security, 136 for 256-bit security, and 72 for 512-bit security setting.
HKDS_EDK_SIZE	16, 32, 64	The byte length of the embedded device key, corresponding to the HKDS security mode setting; 128-bit, 256-bit, or 512-bit security.
HKDS_FORMAL_NAME	0x48, 0x4B, 0x44, 0x53…	The HKDS implementation name. The ascii name that represents the formal name of the HKDS implementation based on the security mode: HKDSXXX, where XXX represents the security level, 128, 256, or 512.
HKDS_MAC_NAME	0x75, 0x4B, 0x77, 0x65, …	The formal string name of the MAC function used to authenticate messages.
HKDS_TMS_SIZE	23	The byte size of the token MAC string

Table 8.1: HKDS Client constants.

8.2 Client State

The client state contains the state members used by the HKDS terminal client for its internal operations. This includes unchangeable values stored within a secure tamper-proof module, the embedded device key (EDK) and the portion of the client device identity string (DID) that represents the unique client identity string (the last 32-bits of the DID). Working variables required as persistent state by the client’s functions, such as the key transaction counter, key cache, cache state flag, and the master key identity **SHALL** be stored in non-volatile memory. The master key identity (KID) portion of the KSN **SHALL** be programmable, should the master key ever need to be changed. Whereas the device identity (DID), shall be considered a permanent assignment, and protected along with the embedded key (EDK) in a tamper-proof module.

The example software implementation of the client state is represented here as a struct, a composite structure containing the arrays and variables used by the client’s function calls, this has been done for the sake of providing clarity in the example implementation. However, the programming of a terminal client device **SHALL** separate the components into shared non-volatile memory, and the secure memory module, as described.

The **hkds_client_state** contains the HKDS client state.

State Member Name	Value Type	Description
KTC	Array; 4 bytes.	The key transaction counter. A monotonic big-endian aligned counter, incremented each time a transaction key is extracted from the key cache. This array is a part of the KSN.
KID	Array; 4 bytes.	The master key identity. Used by the server to identify the master key associated with the device.
DID	Array; 12 bytes.	The device identity string. A unique string that identifies both the client terminal, and the clients master key. This array is part of the KSN.
KSN	Array 16 bytes.	The KSN combines the master key identity (KID), the device identity (DID), and the key transaction counter (KTC), into a single array that comprises the key serial number.
EDK	Array; 16,32,64 bytes.	The embedded device key. This key is stored in the SCD, and cannot be changed. The EDK size is determined by the security level of the HKDS implementation; 128, 256, and 512-bits, for 16, 32, and 64-byte key sizes.
TKC	Multidimensional byte array; variable size.	The transaction key cache; holds a number of 16-byte transaction keys, used to encrypt messages sent to the server and key the MAC function. The size of the key cache is dependent on the

		security configuration and the key cache multiplier setting.
Cache Empty	Boolean; cache state.	Indicates the key cache state readiness.

Table 8.2: HKDS Client state.

8.3 Client Initialization

The client begins initialization by transferring the EDK to the security module, and loading the client's device identity string (DID) into the KSN. This function represents a simplified transfer of the EDK, which will have to be loaded per the requirements of the hardware security module used by the client device. Other state like the state variables and the transaction key cache, should be stored in non-volatile memory on the device.

Once the devices internal state has been initialized with the EDK and device identity, and the device is connected to the network, it must make a token key request to the server, before the transaction key cache can be constructed, and the device can begin managing user transaction requests.

All transactions between the client and server **SHOULD** use a reliable network transport mechanism, we recommend TCP, the transport control protocol. The network transport must handle the underlying transport of packets at a layer beneath the HKDS protocol operation, and manage the reliable exchange of packet data between the client terminal and transaction server. For guidance on the HKDS packet structures, refer to section 5.3: The HKDS packet structure.

Once the encrypted token key has been received by the client, the client authenticates the token. If the MAC code appended to the encrypted token, matches the one generated by the client for the cipher-text received from the server, the client decrypts the token. If the codes do not match, the function returns false and the token is set to zero, to be processed by the application stack with either a new token request, or by placing the device in an error state pending reset.

The client uses the custom token array (CTOK) concatenated with the embedded device key (EDK), to key the PRF and generate the key-stream. The client then uses this key-stream to decrypt the secret token key.

8.4 Client Key Cache Generation

The token key is concatenated with the EDK to key the PRF, which in turn, generates the client's transaction key cache.

When the client is initialized, the encrypted secret token key dispatched by the server, is decrypted using the custom token (CTOK) and embedded device key (EDK), permuted to create a key-stream that is XOR'd with the encrypted token (ETOK).

After the token decryption, a new key cache is generated using the secret token key, concatenated with the client's EDK to initialize the pseudo random function (SHAKE), used to generate the transaction key cache.

Each time a key cache is generated, the secret token key, **SHALL** be erased from the client's memory. Erasure of the token key ensures that previously used keys in the cache that have been erased, cannot be reconstructed (forward security).

The secret token key is concatenated with the EDK, to key the PRF which then generates the transaction key cache, the token is then erased from the client's memory. In this way, even if the client's internal state is captured, the adversary cannot predict future keys, unless the adversary possesses both the protected EDK, and owns the communication channel and can capture future token exchanges, this provides a high degree of predictive resistance to all future transaction keys, excepting in the complete compromise of both the terminal, and the communications channel.

8.5 Client Encryption

Client encryption of a PIN message consists of loading the next available transaction key from the transaction key cache, incrementing the transaction counter, and XORing the transaction key with the plain-text message to produce the cipher-text, and then erasing that key from the cache. Because of the simplicity of this step, client encryption is very fast and efficient.

It should be noted, that although HKDS is designed primarily as a means to encrypt small messages, such as PIN numbers used by point-of-sale devices, it can also be used to key a cipher and mode, such as AES-GCM, for establishing an encrypted 2-way communications channel between terminals. The terminals would only need to extract the number of keys required to initialize the underlying cipher at the desired security. HKDS could easily be ported to this application, allowing a low cost, symmetric-based encryption scheme between remote terminals that have been preloaded with embedded keys. Because of the low-cost of the HKDS cryptosystem, both in the small implementation code size, and the low computational cost of the mechanism, it is ideally suited to IOT devices with limited memory and CPU configurations.

8.6 Client Authenticated Encryption

HKDS uses KMAC, the Keccak based keyed-hash function for authentication. KMAC uses the same permutation function as SHAKE, re-using that function, to keep code size small, and KMAC, like SHAKE, uses 64-bit integers, making it very fast in hardware implementations. Benchmarks of KMAC versus HMAC(SHA-256), on a 64-bit compilation, show KMAC is significantly faster at authenticating both large and small messages.

The client extracts two transaction keys from the key cache, the first key is used to encrypt the message. The second key then keys KMAC, which takes the encrypted message as input, and generates a MAC code in an encrypt-then-MAC configuration. The 16-byte MAC code is appended to the cipher-text, and used by the server to verify that the message has not been altered in transit.

8.7 Client API

Decrypt Token

Decrypt an encrypted token key sent by the server.

```
bool hkds_client_decrypt_token(hkds_client_state* state, const uint8_t* etok, uint8_t* token)
```

Encrypt Message

Encrypt a message to be sent to the server.

```
bool hkds_client_encrypt_message(hkds_client_state* state, const uint8_t* plaintext, uint8_t* ciphertext)
```

Encrypt Authenticate Message

Encrypt a message and add an authentication tag. The PIN is first encrypted, then the cipher-text is used to update a keyed KMAC function. An optional data can be added to the MAC update, such as the IP address of the client. The authentication tag is appended to the encrypted PIN and returned by the function.

```
bool hkds_client_encrypt_message(hkds_client_state* state, const uint8_t* plaintext, uint8_t* ciphertext)
```

Generate Cache

Initialize the state with the secret key and nonce.

```
void hkds_client_generate_cache(hkds_client_state* state, const uint8_t* token)
```

Initialize

Initialize the state with the embedded device key and device identity.

```
void hkds_client_initialize_state(hkds_client_state* state, const uint8_t* edk, const uint8_t* did)
```

9: Server Description

9.1 Server State

The master derivation key (MDK) is comprised of three arrays; the base derivation key (BDK), used to derive the clients embedded device key (EDK). The secret token key (STK), which is used to derive unique secret token keys sent to the client, and the key identity (KID), a unique 4-byte identity string, which is used by the server to select the master key associated with the client device. The size of the BDK and the STK keys are dependent upon the implementation security used by HKDS; 128-bit security uses 16-byte BDK and STK random keys, 256-bit security uses 32-byte keys, and 512-bit security use 64-byte keys.

It may be proposed that these key sizes can be reduced by half, in that the keys generated by the client and subsequently derived by the server, are generated with a combination of both the BDK and STK derivation keys. However, HKDS uses a two-tier security paradigm, one that uses the secondary token key to inject entropy into the system, and that the integrity of this system against future attacks will depend on both keys being equal to the desired security level, thereby, an implementation of HKDS **SHALL** employ the key sizes as described.

HKDS is a fast, efficient, and highly optimized key distribution system, reducing key sizes to save a very small amount of cost, is not considered a valid optimization technique, and could undermine the security of the mechanism. Any implementations that aim to reduce key sizes, will be rejected as non-compliant to this specification, and shall be regarded as potentially insecure.

The **hkds_server_state** contains the HKDS server state.

State Member Name	Value Type	Description
KID	Array; 4 bytes.	The master key identity. Used to select the master key associated with a client device.
STK	Array; 16, 32, 64 bytes.	The secret token key array. The secret token key is used as a base key that is the combined with the custom token string (CTOK), to derive unique secret tokens sent to the client.
BDK	Array; 16, 32, 64 bytes.	The base derivation key. The base derivation key is combined with the client's device identity, to create the

		client's embedded device key (EDK).

Table 9.1a: MDK state members.

State Member Name	Value Type	Description
Rate	Unsigned integer constant.	The rate is the byte rate at which the underlying pseudo random function (SHAKE), performs permutations and the pseudo-random output size of the permutation.
Count	Unsigned integer constant.	The client's transaction key counter. Used to select the correct key from the reconstructed client key cache.
MDK	Structure; variable size.	The master derivation key structure associated with the client device.
KSN	Array 16 bytes.	The KSN combines the master key identity (KID), the device identity (DID), and the key transaction counter (KTC), into a single array that comprises the key serial number.

Table 9.1b: HKDS Server state.

9.2 Server Constants

The server constants must remain static, unchanged in the implementation in order to match constants in the client implementation. The one constant that can be defined in the implementation, but must be set identically in both server and client, is the key cache multiplier, the constant that determines the number of key registers in the client's internal key cache. All constant values are defined as unsigned integers.

The constant names and name format correspond to the C reference implementation, linked in the References section.

Constant Name	Value	Description

HKDS_CACHE_MULTIPLIER	4	The cache multiplier; key cache total byte size is a multiple of this value times the output size of the SHAKE PRF function.
HKDS_DID_SIZE	12	The size of the device identity string, 12 bytes.
HKDS_KID_SIZE	4	The size of the master key identity string, 4 bytes. The KID is part of the device identity.
HKDS_KSN_SIZE	16	The size of the key serial number, 16 bytes. The key serial number is comprised of the device identity string, and the transaction key counter.
HKDS_MESSAGE_SIZE	16	The size of the HKDS encrypted message output, 16 bytes.
HKDS_NAME_SIZE	7	The size of the HKDS implementation name string, 7 bytes. The name string is a component in the CTOK token string.
HKDS_TKC_SIZE	4	The size of the key transaction counter, 4 bytes.
HKDS_TAG_SIZE	16	The size of the KMAC authentication tag, 16 bytes.
HKDS_SHAKE_XXX	0x09-0x0B	The SHAKE security mode used by the HKDS implementation. Possible values are 128=0x09, 256=0x0A, 512=0x0B.
HKDS_CTOK_SIZE	23	The byte length of the custom token string CTOK.
HKDS_NAME_SIZE	7	The byte length of the HKDS implementations formal name.
HKDS_PRF_RATE	168, 136, 72	The rate size of the SHAKE permutation function. Possible values are 168 for 128-bit security, 136 for 256-bit security, and 72 for 512-bit security setting.
HKDS_EDK_SIZE	16, 32, 64	The byte length of the embedded device key, corresponding to the HKDS security mode setting; 128-bit, 256-bit, or 512-bit security.

HKDS_FORMAL_NAME	0x48, 0x4B, 0x44, 0x53…	The HKDS implementation name. The ascii name that represents the formal name of the HKDS implementation based on the security mode: HKDSXXX, where XXX represents the security level, 128, 256, or 512.
HKDS_MAC_NAME	0x75, 0x4B, 0x77, 0x65, …	The formal string name of the MAC function used to authenticate messages.
HKDS_TMS_SIZE	23	The byte size of the token MAC string
HKDS_KECCAK_HALF_ROUNDS	N/A	Sets the Keccak permutation function at half rounds (12 vs 24) for a high-performance implementation.

Table 9.2: HKDS Server constants.

9.3 Server Initialization

The server receives the plain-text KSN along with an encrypted message from the client. The server uses the KID, the master key identity, which is the first 4 bytes of the KSN, to identify the master key associated with the client. The server then loads the master key, and loads the client’s device identity and transaction count.

9.4 Server Token Generation

The server generates a unique secret token-key for each client, and for every token exchange. The token key is derived by concatenating the secret token key (STK), with the implementation name, the client’s device identity, and the token requests counter (CTOK), and uses this to key the pseudo random function (SHAKE). The PRF then derives a new token key, keeping the primary STK secret from the client, and the secret token unique for each client device and token request.

The server initializes the MAC function state to the token MAC string (TMS), keys the function with the client’s EDK, and processes the encrypted token key as the message, appending a 16-byte MAC code to the encrypted token key (ETOK).

The server encrypts the token by recreating the client’s embedded device key (EDK), concatenating the custom token string (CTOK) with the EDK and permuting them to create the token encryption key. The server encrypts the token key by a bitwise XOR of the token, and the token encryption key.

9.5 Server Transaction Key Derivation

The server generates the transaction key through a series of steps. The first step is to derive the client’s embedded device key, by concatenating the client’s device identity (DID), with the base derivation key (BDK) to key the PRF. The client’s embedded device key (EDK), is the output of the PRF.

The server then generates the second key, the client’s secret token key, and concatenates the two keys to initialize the PRF. The server generates the client’s key cache, up to the block containing the transaction key specified in the transaction counter portion of the KSN, and selects the correct key from the block.

9.6 Server Message Decryption

The server decrypts a client message by deriving the transaction key as described above, and then performing a bitwise XOR of the message cipher-text and the transaction key to create the message plain-text.

9.7 Server Message Verification and Decryption

The server performs decryption by deriving two of the client’s transaction keys; the first key is used to decrypt the cipher-text, the second to key the MAC function. HKDS uses an encrypt-then-MAC authentication scheme, whereby the message is first encrypted, and then the message cipher-text is used as the message input for the MAC function.

The second key is used to initialize KMAC, and then the cipher-text sent by the client is hashed to calculate the MAC code. The newly generated MAC code is compared to the MAC code appended to the cipher-text by the client’s authentication process. If the MAC codes match, the function uses the first key to decrypt the cipher-text to the plaintext array, and the function returns true. If the codes do not match, the function skips decryption, and returns false, and a zeroed plaintext array.

9.8 Server API

Decrypt Message

Decrypt a message sent by the client.

```
void hkds_server_decrypt_message(hkds_server_state* state, const uint8_t* ciphertext, uint8_t* plaintext)
```

Decrypt Verify Message

Verify a ciphertext's integrity with a keyed MAC, if verified return the decrypted PIN message. This function uses KMAC to verify the cipher-text integrity before decrypting the message. An optional data can be added to the MAC update, such as the originating clients IP address. If the MAC verifies the cipher-text, the message is decrypted and returned by this function. If the MAC authentication check fails, the function returns false and the plaintext is zeroed.

```
void hkds_server_decrypt_message(hkds_server_state* state, const uint8_t* ciphertext, uint8_t* plaintext)
```

Encrypt Token

Encrypt a secret token key to send to the client.

void hkds_server_encrypt_token(hkds_server_state* state, uint8_t* etok)

Generate EDK

Generate the embedded device key of a client.

void hkds_server_generate_edk(const uint8_t* bdk, const uint8_t* did, uint8_t* edk)

Generate MDK

Generate a master key set.

void hkds_server_generate_mdk(void (*rng_generate)(uint8_t*, size_t), hkds_master_key* mdk, const uint8_t* kid)

Initialize

Initialize the state with the embedded device key and device identity.

void hkds_server_initialize_state(hkds_server_state* state, hkds_master_key* mdk, const uint8_t* ksn)

Initialize

Initialize the state with the embedded device key and device identity.

void hkds_server_initialize_state(hkds_server_state* state, hkds_master_key* mdk, const uint8_t* ksn)

9.9 Parallel Server API

Contains the **hkds_server_x8_state**, the parallel x8 server state.

Data Name	Data Type	Bit Length	Function
ksn	Uint64 2D Array	Variable	The client's key serial number 2d array
mdk	hkds_master_key	0x40	A pointer to the master derivation key struct

Table 9.9 HKDS parallel state structure.

Decrypt Message X8

Decrypt a 2-dimensional x8 set of client messages.

```
void hkds_server_decrypt_message_x8(hkds_server_x8_state* state,
   const uint8_t ciphertext[HKDS_CACHX8_DEPTH][HKDS_MESSAGE_SIZE], uint8_t
plaintext[HKDS_CACHX8_DEPTH][HKDS_MESSAGE_SIZE])
```

Decrypt Verify Message X8

Verify a 2-dimensional x8 set of ciphertext's integrity with a keyed MAC, if verified return the decrypted PIN message. This function uses KMAC to verify the cipher-text integrity before decrypting the message. An optional data can be added to the MAC update, such as the originating clients IP address. If the MAC verifies the cipher-text, the message is decrypted and returned by this function. If the MAC authentication check fails, the function returns false and the plaintext is zeroed.

```
void hkds_server_decrypt_verify_message_x8(hkds_server_x8_state* state,
   const uint8_t ciphertext[HKDS_CACHX8_DEPTH][HKDS_MESSAGE_SIZE +
HKDS_TAG_SIZE],
   const uint8_t data[HKDS_CACHX8_DEPTH][HKDS_MESSAGE_SIZE], size_t datalen,
   uint8_t plaintext[HKDS_CACHX8_DEPTH][HKDS_MESSAGE_SIZE], bool
valid[HKDS_CACHX8_DEPTH])
```

Encrypt Token X8

Encrypt a 2-dimensional x8 set of secret token keys.

```
void hkds_server_encrypt_token_x8(hkds_server_x8_state* state, uint8_t
etok[HKDS_CACHX8_DEPTH][HKDS_STK_SIZE + HKDS_TAG_SIZE])
```

Generate EDK X8

Generate a 2-dimensional x8 set of client embedded device keys.

```
void hkds_server_generate_edk_x8(hkds_server_x8_state* state,
const uint8_t did[HKDS_CACHX8_DEPTH][HKDS_DID_SIZE], uint8_t
edk[HKDS_CACHX8_DEPTH][HKDS_EDK_SIZE])
```

Initialize X8

Initialize a 2-dimensional x8 set of server states with the embedded device keys and device identities.

```
void hkds_server_initialize_state_x8(hkds_server_x8_state* state,
hkds_master_key* mdk, const uint8_t ksn[HKDS_CACHX8_DEPTH][HKDS_KSN_SIZE])
```

Decrypt Message X64

Decrypt a 3-dimensional 8x8 set of client messages.

```
void hkds_server_decrypt_message_x64(hkds_server_x8_state
state[HKDS_PARALLEL_DEPTH],
const uint8_t
ciphertext[HKDS_PARALLEL_DEPTH][HKDS_CACHX8_DEPTH][HKDS_MESSAGE_SIZE], uint8_t
plaintext[HKDS_PARALLEL_DEPTH][HKDS_CACHX8_DEPTH][HKDS_MESSAGE_SIZE]
)
```

Decrypt Verify Message X64

Verify a 3-dimensional 8x8 set of ciphertext's integrity with a keyed MAC, if verified return the decrypted PIN message. This function uses KMAC to verify the cipher-text integrity before decrypting the message. An optional data can be added to the MAC update, such as the originating clients IP address. If the MAC verifies the cipher-text, the message is decrypted and returned by this function. If the MAC authentication check fails, the function returns false and the plaintext is zeroed.

```
void hkds_server_decrypt_verify_message_x64(hkds_server_x8_state
state[HKDS_PARALLEL_DEPTH], const uint8_t
ciphertext[HKDS_PARALLEL_DEPTH][HKDS_CACHX8_DEPTH][HKDS_TAG_SIZE +
HKDS_MESSAGE_SIZE],
const uint8_t
data[HKDS_PARALLEL_DEPTH][HKDS_CACHX8_DEPTH][HKDS_MESSAGE_SIZE],
size_t datalen, uint8_t
plaintext[HKDS_PARALLEL_DEPTH][HKDS_CACHX8_DEPTH][HKDS_MESSAGE_SIZE]
, bool valid[HKDS_PARALLEL_DEPTH][HKDS_CACHX8_DEPTH])
```

Encrypt Token X64

Encrypt a 3-dimensional 8x8 set of secret token keys.

```
void hkds_server_encrypt_token_x64(hkds_server_x8_state
state[HKDS_PARALLEL_DEPTH],
uint8_t etok[HKDS_PARALLEL_DEPTH][HKDS_CACHX8_DEPTH][HKDS_STK_SIZE +
HKDS_TAG_SIZE])
```

Generate EDK X64

Generate a 3-dimensional 8x8 set of client embedded device keys.

```
void hkds_server_generate_edk_x64(hkds_server_x8_state state[HKDS_PARALLEL_DEPTH],
const uint8_t
did[HKDS_PARALLEL_DEPTH][HKDS_CACHX8_DEPTH][HKDS_DID_SIZE], uint8_t
edk[HKDS_PARALLEL_DEPTH][HKDS_CACHX8_DEPTH][HKDS_EDK_SIZE])
```

Initialize X64

Initialize a 3-dimensional 8x8 set of server states with the embedded device keys and device identities.

```
void hkds_server_initialize_state_x64(hkds_server_x8_state
state[HKDS_PARALLEL_DEPTH],
hkds_master_key mdk[HKDS_PARALLEL_DEPTH], const uint8_t
ksn[HKDS_PARALLEL_DEPTH][HKDS_CACHX8_DEPTH][HKDS_KSN_SIZE])
```

9.10 HKDS Factory API

Serialize Header

Serialize a packet header to a byte array.

```
void hkds_factory_serialize_packet_header(uint8_t* output, const hkds_packet_header* header)
```

Serialize Client Message

Serialize a client message request to a byte array.

```
void hkds_factory_serialize_client_message(uint8_t* output, const
hkds_client_message_request* header)
```

Serialize Client Token

Serialize a client token request to a byte array.

```
void hkds_factory_serialize_client_token(uint8_t* output, const hkds_client_token_request*
header)
```

Serialize Server Message

Serialize a server message response to a byte array.

```
void hkds_factory_serialize_server_message(uint8_t* output, const
hkds_server_message_response* header)
```

Serialize Server Token

Serialize a server token response to a byte array.

```
void hkds_factory_serialize_server_token(uint8_t* output, const hkds_server_token_response*
header)
```

Serialize Administrative Message

Serialize an administrative message to a byte array.

void hkds_factory_serialize_administrative_message(uint8_t* output, const hkds_administrative_message* header)

Serialize Error Message

Serialize an error message to a byte array.

void hkds_factory_serialize_error_message(uint8_t* output, const hkds_error_message* header)

Extract Packet Header

Extract a packet header structure from a byte array.

hkds_packet_header hkds_factory_extract_packet_header(const uint8_t* input)

Extract Client Message

Extract a client message request from a byte array.

hkds_client_message_request hkds_factory_extract_client_message(const uint8_t* input)

Extract Client Token

Extract a client token request from a byte array.

hkds_client_token_request hkds_factory_extract_client_token(const uint8_t* input)

Extract Server Message

Extract a server message response from a byte array.

hkds_server_message_response hkds_factory_extract_server_message(const uint8_t* input)

Server Token Response

Extract a server token response from a byte array.

hkds_server_token_response hkds_factory_extract_server_token(const uint8_t* input)

Extract Server Token

Extract a server token response from a byte array.

hkds_server_token_response hkds_factory_extract_server_token(const uint8_t* input)

Extract Administrative Message

Extract an administrative message from a byte array.

hkds_administrative_message hkds_factory_extract_administrative_message(const uint8_t* input)

Extract Error Message

Extract an error message from a byte array.

hkds_error_message hkds_factory_extract_error_message(const uint8_t* input)

Create Client Message Request

Build a client message request from components.

hkds_client_message_request hkds_factory_create_client_message_request(const uint8_t* message, const uint8_t* ksn, const uint8_t* tag)

Create Client Token Request

Build a client token request from components.

hkds_client_token_request hkds_factory_create_client_token_request(const uint8_t* ksn)

Create Server Message Response

Build a server message response from components.

hkds_server_message_response hkds_factory_create_server_message_response(const uint8_t* message)

Create Server Message Response

Build a server token response from components.

hkds_server_token_response hkds_factory_create_server_token_reponse(const uint8_t* etok)

Create Administrative Message

Build an administrative message from components.

hkds_administrative_message hkds_factory_create_administrative_message(const uint8_t* message)

Create Error Message

Build an error message from components.

hkds_error_message hkds_factory_create_error_message(const uint8_t* message, hkds_error_type err)

Extract Packet Type

Extract the packet type enumeral from a serialized packet header.

hkds_packet_type hkds_factory_extract_packet_type(const uint8_t* input)

Extract Protocol Id

Extract the protocol id numeral from a serialized packet header.

hkds_protocol_id hkds_factory_extract_protocol_id(const uint8_t* input)

Extract Packet Size

Extract the packet size from a serialized packet header.

size_t hkds_factory_extract_packet_size(const uint8_t* input)

Extract Packet Size

Extract the packet sequence from a serialized packet header.

uint8_t hkds_factory_extract_packet_sequence(const uint8_t* input)

9.11 HKDS Queue API

Contains the **hkds_queue_message_queue**, the HKDS queue state.

Data Name	Data Type	Bit Length	Function
tag	Uint8 Pointer	0x40	The tag associated with this queue
state	qsc_queue_state	Variable	The queue state context

Table 9.11 HKDS queue state.

Destroy

Resets the queue context state.

```
void hkds_queue_destroy(hkds_queue_message_queue* ctx)
```

Flush

Flush the contents of the queue to a byte array.

```
void hkds_queue_flush(hkds_queue_message_queue* ctx, uint8_t* output)
```

Initialize

Initializes the queues state context.

```
void hkds_queue_initialize(hkds_queue_message_queue* ctx, size_t depth, size_t width,
uint8_t* tag)
```

Pop

Removes an item from the queue and copies it to the output array.

```
void hkds_queue_pop(hkds_queue_message_queue* ctx, uint8_t* output, size_t outlen)
```

Pull

Adds an item from the queue.

```
void hkds_queue_push(hkds_queue_message_queue* ctx, const uint8_t* output, size_t outlen)
```

IsFull

Returns true if the queue is full.

```
bool hkds_queue_isfull(hkds_queue_message_queue* ctx)
```

IsEmpty

Returns true if the queue is empty.

```
bool hkds_queue_isempty(hkds_queue_message_queue* ctx)
```

Count

Returns the number of items in the queue.

```
size_t hkds_queue_count(hkds_queue_message_queue* ctx)
```

Extract Block X8

Export a block of 8 messages to a 2-dimensional message queue.

```
size_t hkds_queue_extract_block_x8(hkds_queue_message_queue* ctx, uint8_t output[HKDS_CACHX8_DEPTH][HKDS_MESSAGE_SIZE])
```

Extract Block X64

Export 8 slots of 8 blocks of messages (8x8) to a 3-dimensional message queue.

```
size_t hkds_queue_extract_block_x64(hkds_queue_message_queue* ctx, uint8_t output[HKDS_PARALLEL_DEPTH][HKDS_CACHX8_DEPTH][HKDS_MESSAGE_SIZE])
```

Extract Block X64

Serialize a set of messages to an array.

```
size_t hkds_queue_extract_stream(hkds_queue_message_queue* ctx, uint8_t* stream, size_t items)
```

10: Performance

10.1 DUKPT-AES versus HKDS Performance Comparison

The most relevant comparison is one in which we compare the performance of HKDS with the algorithm it is meant to replace; DUKPT-AES. So that the reader may gauge the performance characteristics of both algorithms in the context of the tasks they perform. To make this comparison, we have written a C implementation of DUKPT, based on the ANSI revision; X9.24-3-2017, the AES based DUKPT implementation. The DUKPT implementation is based on the code presented in that document, but has been optimized for best possible performance while strictly adhering to that specification.

We use two versions of the DUKPT code for comparison, the first uses a standard software-based AES C implementation, the core permutation function in DUKPT. The second implementation uses an optimized version of AES that utilizes embedded CPU instructions (AES-NI), which offers a significant speedup to the DUKPT performance benchmarks.

We test with both implementations, because the Keccak implementation used, is not optimized, it is a software implementation of SHAKE, with a standard Keccak permutation function. That Keccak permutation function can however, also be placed on a CPU's instruction set, as was done with AES, but we do not have access to a processor with those instructions, so thought it would be best to present both versions, asking the reader to bear in mind, that the HKDS implementation can also be optimized in a similar way, and that those optimizations could offer a near doubling of performance.

We measure each relevant operation; encryption and decryption of a cryptogram, as well as authenticated encryption and verified decryption of a cryptogram. HKDS uses KMAC for authentication, the Keccak based MAC function, and for DUKPT, we use the recommended HMAC function with SHA2-256.

These performance benchmarks are not complete however; the reader must also understand that the performance of DUKPT degrades on a server-side implementation as the transaction counter increments. The number of AES-128 re-keys and decryption calls increase to a maximum work factor of 16, meaning that as the transaction counter approaches the upper limit of keys that can be generated, the number of initializations and executions of the AES transform also increases, and performance, on the server managing decryption of client messages, steadily degrades. For example, with the transaction counter at 7, decryption requires 3 re-keys and transformation function executions, at 63 transactions, 6 executions, at 8191 transactions 13 AES re-keys and transform execution calls are required to recover the transaction key. This goes all that way up to a maximum work factor of 16 to generate the working key, plus one more to derive the initial intermediate key from the base derivation key, and a final derivation to produce the transaction key, for a total possible 18 AES re-keys and transformations required to recreate the transaction key. This number is **doubled** if using AES-256, for each AES key of 256 bits, requires two complete re-keys of AES and transformation calls for each cycle. This number is **doubled again**, if using authenticated encryption, as HMAC requires its own unique key.

HKDS performance is constant, regardless of the number of transactions. The key cache multiplier in HKDS determines the maximum number of permutations required to completely reconstruct the key cache. The default setting is 4, if the transaction key is within a permutation boundary, it will exit the permutation loop with only the number of permutation calls required to regenerate the transaction key. So, the variance is always a constant within that range, between 1 and the multiplier maximum, regardless how many previous keys have been generated.

For this reason, only a complete benchmark to the maximum number of keys that can be derived by DUKPT could produce a true average of performance for this comparison, given that algorithms performance degradation, and this limited benchmark favors DUKPT by not including the complete metric.

HKDS also only requires one transaction key for encryption or decryption, whether it be for 128, 256, or 512-bit security profiles, and only requires one additional key for authentication.

Though the most important benchmark figures, are the ones related to server-side processing, which are the most expensive operations, and directly impacts the costs incurred by financial transaction processing institutions, the client-side benchmarks are also included, so that we can compare the impact on point-of-sale devices, which have limited processing power and memory resources.

Using a standard software-based form of AES with DUKPT:

Task	HKDS-128	HKDS-256	DUKPT-128	DUKPT-256
Decryption	1761	1675	12013	22534
Encryption	101	102	4226	11686
Encrypt-Auth	1461	1479	13276	26139
Decrypt-Verify	3545	3548	27733	48615

Table 10.1a: HKDS versus DUKPT-AES performance benchmarks.

Using AES-NI and an optimized DUKPT:

Task	HKDS-128	HKDS-256	DUKPT-128	DUKPT-256
Decryption	1866	1751	7414	11615
Encryption	96	112	2497	4495
Encrypt-Auth	1404	1475	10413	14188
Decrypt-Verify	3683	3563	19071	27094

Table 10.1b: HKDS versus DUKPT-AESNI performance benchmarks.

These benchmarks are the average time in nanoseconds, using an average across iteration counts based on multiples of the HKDS key-cache size. For 128-bit implementations, 42 thousand iterations were sampled. For 256-bit benchmarks, 34 thousand iterations were measured. For authenticated encryption and decryption, the sample rate was halved to account for two keys being used in every iteration to 21 and 17 thousand respectively. The numbers are a one-off sample, using an Intel i7-9750H CPU, with 16 gigabytes of RAM.

As we can see from the benchmark figures, HKDS outperforms DUKPT by a huge margin, in every type of operation, in some cases, by more than a 100 to 1 ratio, even with the use of an optimized DUKPT-AES using embedded CPU instructions.

Using CPUs with embedded Keccak instructions on servers processing transactions should further reduce HKDS processing times by a substantial margin. These savings could translate to enormous cost reductions in the management of remote banking transactions handled by financial institutions.

The potential of HKDS, as a primary mode of secure encryption and authentication for point-of-sale transactions, is staggering, costs could potentially be reduced by the institutions managing those transactions, by as much as 75 percent or more. HKDS also has a clear trajectory towards long-term security, with low-cost 256-bit security, and up to 512-bit security capable implementations, HKDS is capable of handling secure transactions for many decades to come.

HKDS can be further enhanced by reducing the number of rounds used by the Keccak permutation function. This option is enabled by using the Keccak half rounds flag, which reduces the permutation rounds from the standard twenty-four to twelve rounds. This nearly doubles the performance of the HKDS server functions as demonstrated in the table below. Given that Keccak is one of the most thoroughly studied cryptographic functions of modern times, and that to date, only a handful of rounds can be broken, and that encrypting PIN numbers can be considered in the context of a low-to-medium security operation, we believe this is an acceptable performance enhancement, at least as it concerns pre-quantum computer implementations.

Performance comparisons between full and half round Keccak:

Task	**HKDS-128**	**HKDS-256**	**PHKDS-128**	**PHKDS-256**
Decryption	1.353	1.268	0.833	0.796
Encryption	0.112	0.119	0.069	0.080
Decrypt-Verify	2.545	2.480	1.546	1.495
Encrypt-Auth	0.668	0.673	0.389	0.388

Table 10.1c: HKDS full versus half rounds performance benchmarks.

The benchmarks measure the number of keys encrypted and decrypted, using 1 million keys, with the result in fractional seconds. These are the standard HKDS functions, significant speedups can be realized using the SIMD and parallel forms of these functions.

11. Cryptanalysis of the Hierarchical Key-Distribution System (HKDS)

11.1 Threat Model and Security Goals

We follow the symmetric-key AKE framework of **Bellare & Rogaway** enriched with two HKDS-specific notions:

- **Forward Secrecy (FS)** – compromise of all client state *after* a token key has been erased reveals no earlier transaction keys (TKs).
- **Predictive Resistance (PR)** – compromise of all client state *before* the next token refresh reveals no *future* TKs.

The active adversary $\mathfrak{A}$ may:

- eavesdrop, modify, inject, and replay any network message;
- corrupt any client or server at any time and read every secret it stores (KSN, token cache, BDK, STK, …);
- obtain chosen-message MAC tags and chosen-ciphertext outputs of the XOR-stream cipher.

HKDS must achieve:

1. **Token Authenticity** – only the legitimate server can produce a token that any client accepts.
2. **Message Confidentiality & Integrity** – a PIN block encrypted under a fresh TK is IND-CPA / INT-CTXT secure.
3. **FS & PR** as defined above.
4. **Efficient Server Computation** – reconstructing a token key requires at most μ = HKDS_CACHE_MULTIPLIER Keccak permutations.
5. **Replay Resistance** – replay of any message or token is always detected.

Symbols follow § 7 “Formal Description”. HKDS Specification

11.2 Token-Exchange (Security of ETOK)

Server step ETOK ← TOK ⊕ P(CTOK ‖ EDK) with

- **EDK** – escrow-derived key, retained only inside the server’s secure cryptographic device (SCD);
- **P** – full-round Keccak permutation;
- **CTOK** – control block = H(KSN‖“KMAC”) ‖ counter encoding.

Client step TOK ← ETOK ⊕ P(CTOK ∥ EDK)

Property	Security argument	Result
Authenticity	Forging ETOK that passes client verification requires MAC-forging under EDK. KMAC is UF-CMA; EDK never leaves SCD. HKDS Specification	**Secure**
Confidentiality	Without EDK, ETOK is a one-time pad of TOK; Keccak outputs are pseudorandom.	**Secure**
Freshness / Anti-replay	KSN. counter part of CTOK is strictly increasing; reuse causes counter mismatch.	**Replay-safe**

Observation – Compromise of **STK** (server token key) allows decryption of *past* ETOKs but does **not** help to forge *new* ones because EDK is still required. Periodic STK rotation binds a validity epoch to each token batch.

11.3 Transaction-Key Derivation & PIN-Message Security

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

$\mathrm{TK}_i = \mathrm{TKC}[i],\ 0 \le i < |\mathrm{TKC}|$,

$Y = x \oplus \mathrm{TKi}$,

$r = \mathrm{KMAC}(\mathrm{TK}_{i+1}, y)$.

Security property	Argument	Result
IND-CPA	Each TK_i is uniformly random (SHAKE as PRF). The XOR stream cipher with fresh TK_i yields ciphertexts indistinguishable from random.	**Secure**
INT-CTXT	Tag τ uses independent key TK_{i+1}; adversary must forge KMAC ⇒ break UF-CMA.	**Secure**
Forward Secrecy	TK_i is erased immediately after use; later disclosure of all other material cannot recover TK_i.	**Achieved**
Predictive Resistance	Deriving TK_i requires a fresh TOK, which in turn depends on server-only STK. Compromise of a client before refresh cannot compute future TKs.	**Achieved**

Key-Compromise Impersonation	Purely symmetric setting: client impersonation impossible without EDK; server impersonation impossible without STK/EDK.	**Not vulnerable**
Computation bound	Server reconstructs ≤ μ permutations per refresh (constant).	**Efficient**

Implementation note – The optional 12-round reduced Keccak halves the security margin. Deploy the full 24-round permutation for 256-bit and 512-bit profiles in high-assurance environments.

11.4 Security Gaps & Hardening Recommendations

1. **STK single-point compromise** – loss of STK exposes all *past* TOKs in its epoch.
   Mitigation: rotate STK on short epochs (e.g., weekly) and embed epoch ID in CTOK.
2. **MAC tag length** – 128-bit tags suffice for HKDS-128 but truncate security for HKDS-256.
   Recommendation: use 256-bit tags in the 256-bit profile.
3. **Reduced-round mode** – limit 12-round Keccak to low-risk deployments; mandate 24 rounds for FIPS-140-3 scope.
4. **Clock-free replay defense** – augment KSN with a per-token random nonce to survive power-loss resets and maintain monotonicity.
5. **Side-channel hygiene** – ensure constant-time Keccak inside SCD; mask EDK fetches to prevent DPA/EMA attacks.

11.5 Comparative Snapshot (HKDS vs Legacy DUKPT)

Scheme	Tokens / Refresh	FS	PR	Server Work	Tag Length
HKDS-128	42	✓	✓	≤ 4 P(Keccak)	128 b
HKDS-256	34	✓	✓	≤ 4 P(Keccak)	256 b
DUKPT-AES	≤ 100 k	✗	✗	≤ 18 AES	64 b

HKDS offers *stronger* security properties with bounded, constant-time server cost while remaining computationally cheaper than DUKPT.

11.6 Concluding Assessment

Under the standard assumptions that **SHAKE** behaves as a pseudorandom function and **KMAC** is unforgeable, HKDS realizes authenticated and confidential distribution of transaction keys with both forward secrecy and predictive resistance. Implementing the hardening steps listed in § 11.4 will close the residual gaps and align HKDS with high-assurance payment environments such as PCI HSM v4 and FIPS 140-3.

Annex A: Known Answer Tests

The known answer tests compare the output of a client message encryption with a known answer.

The same client message is used for all HKDS implementations.

The client message: 000102030405060708090A0B0C0D0E0F

HKDS-128

The server MDK, which is used to generate the client's EDK:

BDK: 000102030405060708090A0B0C0D0E0F

STK: 000102030405060708090A0B0C0D0E0F

KID: 00010203

The client's DID: 010000000A09010001000000

Expected cipher-text: 21EDC540F713649F38EDB3CB9E26336E

HKDS-256

BDK: 000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F

STK: 000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F

KID: 00010203

The client's DID: 010000000A0A010001000000

Expected cipher-text: 4422FD14DC32CF52765227782B7DF346

HKDS-512

BDK:
000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F2021222324
25262728292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F

STK:
000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F2021222324
25262728292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F

KID: 00010203

The client's DID: 010000000A0B010001000000

Expected cipher-text: 8F8237E723C13AC5C07BDDE483F586DB

Annex B: Design Decisions

The original design of HKDS did not use the SHAKE function as the primary PRF, instead it used the cSHAKE XOF function as defined in NIST SP 800-185. The cSHAKE XOF was keyed by using CTOK as the customization parameter when generating the token encryption key, and the secret token as the customization parameter when generating the transaction key cache, rather than a concatenation of input arrays used as a single-key mechanism. The choice was made to use the standard SHAKE form instead, primarily as a performance consideration, though we believe that the cSHAKE form of HKDS could possess better security properties, at the cost of an additional permutation call required for this form of HKDS. Should this form of HKDS be preferred in the future, it can easily be changed and the specification revised, with a small penalty to the overall performance. Likewise, KMAC was also considered as the primary PRF in the generation of the key cache, but that SHAKE possesses strong diffusion characteristics, and that the key, and nonce sizes are constant in this design, we did not feel the burden of an additional permutation call required by cSHAKE or KMAC to pre-initialize the state was required, or added a significant enough security benefit to be justified, given the specific aspects of this design.

The choice of PRF and permutation functions however, are not absolutely essential to the definition of the HKDS protocol. So long as the PRF used is suitably cryptographically-strong and robust, alternative functions could be considered, such as a CTR DRBG used as the primary PRF, and HKDF(SHA2) as a key derivation function. We feel though, that Keccak based primitives are the ideal functions for this protocol, given their high performance, strong diffusion characteristics, range of security modes, and that they are the NIST SHA3 standard, and thereby FIPS compliant. However, if at some future time it is determined that SHAKE can be out-performed, or is in some other way less than ideal, these alternatives can be explored, and added as need be, to a revised form of this specification.

SHAKE is used in several different ways in this design; as a key derivation function, as a counter hash-based RBG, and producing a keystream for an *ad hoc* stream cipher. In fact, all of these functions rely on SHAKE to act as a pseudo random function, and are thereby simplified to the generic term 'PRF' in this documentation. The pseudo-random token and embedded device key generation mechanisms use SHAKE as a key derivation function, using a key and counter to initialize SHAKE and generate a unique pseudo-random output used as keying material in subsequent operations. The key-cache generation uses SHAKE as an RBG, generating a block of pseudo-random bytes, which are subsequently used as a key-stream; XOR'd with the message in a construction similar to the ChaCha stream cipher. All of these uses are supported in the Keccak documentation, which provides detailed cryptographic proofs of their security, and SHAKE is itself, FIPS approved and part of the NIST SHA3 standard.

From the Keccak-team paper, Cryptographic Sponge Functions, section 3.2:

"A sponge function can also be used as a stream cipher. One can input the key and some initial value and then get the key stream in the squeezing phase. Similarly, a simple pseudo random bit generator can be constructed by absorbing the seed data and then producing the desired number

of bits. For having a mask generating function (also called key derivation function) one simply uses as input the seed and one truncates the output to the requested number of bits."

All of the uses of both SHAKE and KMAC in the HKDS design are fully supported in the Keccak documentation, and backed by extensive cryptanalysis, both by the Keccak team, and during the NIST SHA3 competition, in which these designs were scrutinized by many of the best cryptographers in the world, before winning that competition. In the 5 years that Keccak has been the SHA3 standard, no serious breaks in the design have been discovered, and SHAKE has since been integrated extensively into the cryptographic mainstream, evidenced by its widespread use and acceptance in the cryptographic community. SHAKE and KMAC using 512-bit security are also provided, these are our constructions and as of yet are not officially recognized, but work in the exact same way as SHA3-512, taking less data from the state with each permutation call (2n the expected security), but in all other respects are identical to the supported versions. We include these implementations to provide a post-quantum pathway, such that if 512-bit keys are ever mandated because of unexpected leaps in post-quantum computing, HKDS is ready to meet the new requirements.

That HKDS uses Keccak for both key generation, encryption, and message authentication, creates a small software footprint, as both KMAC and SHAKE, use the same permutation function, creating a compact software for memory constrained devices.

The two-key system used by HKDS adds an additional small cost to the transaction process, but in return it provides a far more secure system in several ways. Using the token injection system, introduces new entropy into the encryption process at regular intervals. Unlike DUKPT, which uses a complex and expensive re-mixing process to derive transaction keys using the same base key for potentially hundreds of thousands of transactions, this injection of new entropy ensures that the keys for these transactions are derived from new keying material, which can be refreshed on the server at any time. DUKPT has an upper limit on the number of derived keys documented at 500 thousand, far less than is used in the lifetime of some busy terminals, which require complex and expensive re-keying of the terminal device. HKDS can change the token key at the server, requiring no modification of the terminal. The complex methods used by DUKPT to derive keys is computationally expensive, and requires an increasing number of AES re-keys and transformations as the total number of transaction keys used increases. HKDS key generation is constant, it does not change, regardless the number of keys that are derived, and is much faster than DUKPT. The token injection used by HKDS is periodic, and that interval can be defined by the implementor, and can be set to only one token exchange per 168 keys used by the terminal, and even at this high-end of the exchange limit, it still outperforms DUKPT-AES128 by a margin of more than 2-to-1, and a margin of 4-to-1 for DUKPT-AES256. The use of token injection also enables predictive resistance; the token key is erased once the key-cache is created, meaning an adversary that captures the terminals state, will not be able to predict future keys, unlike DUKPT. The token system is also forward secure, in that each token sent to the terminal contains a new unique key, so that even if the state is captured, previous keys cannot be deduced. We believe this is a much more secure system in several important ways; it is more flexible, more extensible, less costly, and provides a more secure distributed key infrastructure than DUKPT.

The performance of HKDS can be increased quite dramatically using embedded Keccak instructions. The current decryption ratios benchmarked with our C implementations place HKDS at a 4 to 1 performance ratio against DUKPT-AES128, and an 8 to 1 ratio against DUKPT-AES256. However, studies have been made that show that embedded Keccak instructions can offer more than 6-fold performance increase in the execution of Keccak based functions. From the authors of SIMD Instruction Set Extensions for Keccak:

"Our design is integrated on a 128-bit SIMD interface, applicable to the ARM NEON and Intel AVX (128 bit) architecture. The proposed instruction set is optimized for flexibility and supports multiple variants of the Keccak-f[b] permutation, for b equal to 200, 400, 800, or 1600 bit. We investigate the performance of the design using the GEM5 micro-architecture simulator. Compared to the latest hand-optimized results, we demonstrate a performance improvement of 2 times (over NEON programming) to 6 times (over Assembly programming). For example, an optimized NEON implementation of SHA3-512 computes a hash at 48.1 instructions per byte, while our design uses 21.9 instructions per byte. The NEON optimized version of the Lake Keyak AEAD uses 13.4 instructions per byte, while our design uses 7.7 instructions per byte. We provide comprehensive performance evaluation for multiple configurations of the Keccak-f[b] permutation in multiple applications (Hash, Encryption, AEAD). We also analyze the hardware cost of the proposed instructions in gate-equivalent of 90nm standard cells, and show that the proposed instructions only require 4658 GE, a fraction of the cost of a full ARM Cortex-A9."

Specialized Keccak instructions embedded on server CPUs similar to the AES-NI instruction set, could yield tremendous increases in performance on an already exceptionally robust key distribution protocol. These performance increases could transform the server rooms of financial transaction processing entities world-wide, that would require only a small fraction of the hardware and support infrastructure they currently use, and providing a level of scalability and performance that is impossible using existing schemes and protocols. We believe HKDS can transform the industry, providing a level of performance and security that can meet the challenges we face in this century and beyond.

Annex C: Analysis Summary

One of the best attributes of the HKDS design, is that it is based upon the Keccak family of functions, which have been studied extensively for more than a decade. There are dozens of cryptanalytic papers on the design, by both the Keccak team, and third-party analysis. It spent more than 4 years being studied by many of the world's leading cryptanalysts before being selected as the winner of the NIST SHA3 competition winner. It was written by leading cryptologists, including Joan Daemen, the co-author of Rijndael, the AES standard. There is no doubt as to the security and efficacy of the design of SHAKE or KMAC, they have been studied by many different people, and as of yet, no serious reductions of security have been found.

The best classical cryptanalytic attack on SHAKE-256 to date, is a preimage attack [1] that uses reduced-rounds implementation of 3 rounds, that is broken in 2^{45} operations, which leaves an impressive security margin of 21 rounds. The best attacks on KMAC, are cube based [2] against reduced-round versions, breaking 7 rounds in 2^{71} operations, leaving a huge security margin of 17 rounds.

In quantum computing attacks, it is well understood that Grover's algorithm could reduce the security of most symmetric keyed primitives to *d/2*, or half the key-space. Both SHAKE and KMAC, like the many other algorithms affected by this attack, would require a doubling of the key size in order to remain secure once quantum computers become powerful enough to effectively break most symmetric cryptosystems. This is why 256-bit keys will soon become the minimum standard key size, and 128-bit ciphers, MACs, and hash functions will become obsolete. New cryptanalytic techniques, such as Boolean equation solving [3], or new and as of yet undiscovered attacks, will further reduce this security margin at some future time, and could eventually make 256-bit keys obsolete. HKDS can provide up to 512-bit security, so that even if these attacks are one day realized, and quantum computers evolve to the scale and efficiency required to execute these attacks, HKDS is ready, and can easily be transitioned to a 512-bit key size.