Ledger Key Ring Protocol

Technical White paper

This page is intentionally left blank

Table of contents

Introduction

Since its foundation, Ledger has been working on decentralized technologies. From blockchain to self-sovereign identity, a number of decentralized applications need to ensure privacy and ownership to their users. This technical white paper offers a deep dive into the design of an information sharing protocol which may benefit other decentralized application developers.

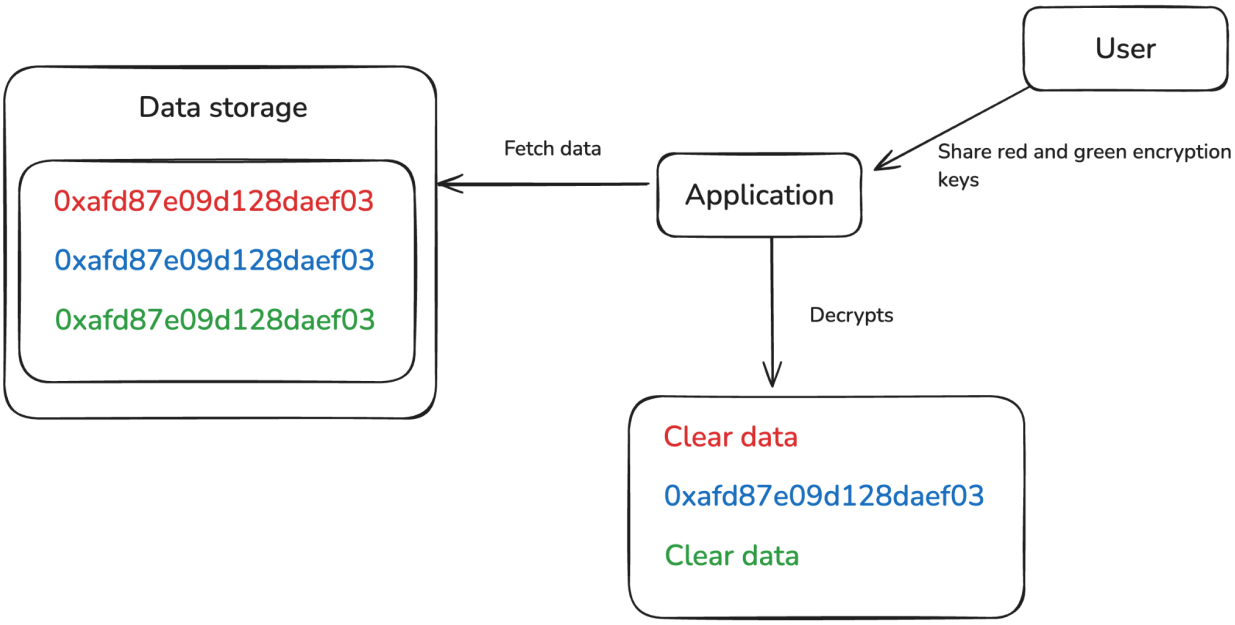
The Ledger Key Ring Protocol (LKRP) can be used in a wide variety of applications such as an end to end encrypted communication application or an application able to selectively disclose part of user data. Ledger's goal with this protocol is to allow multiple application instances to have access to a shared encryption key, which they can use to encrypt or decrypt arbitrary data. LKRP is suitable for applications where re-encrypting the whole data-set each time a new entity is added to the group is not an option. It offers operational flexibility by decorrelating the storage mechanism from the data management. Meaning that it allows decentralized or distributed data management while being agnostic of the way data is stored.

This protocol was designed for applications where users need to share some of their personal information to another entity (such as a service provider) or another trusted application they own. It is developed in the spirit of the "Clear Sign all the things" philosophy carried by Ledger. In other words the protocol was made to improve user ownership over their data, by letting users know precisely what they approve on their trusted display while providing the improved security brought by secure devices (i.e hardware keeping one of multiple data in a secure element).

In this document, we'll be revealing the foundational design and key security targets of the underlying cryptographic protocol employed by the Ledger Key Ring Protocol. This will involve insights into the system's structural layout, as well as a broader view of its primary design and security objectives. Our aim is to provide you with a clear understanding of the intricate mechanisms behind the Ledger Key Ring Protocol.

System design overview

LKRP is designed as a privacy tool for applications needing to share information without relying on a centralized entity knowing the content of the data. The protocol has been developed to implement a "selective disclosure" pattern. The goal is for users to approve which information they want to share. In a centralized application, the central server will take into account the user intent and just send the agreed upon information to a third party application. In this case the central server has access to user information and is able to understand it. For decentralized applications, the data can be stored on multiple servers which can't be trusted by users. First the data is encrypted by a trusted application, then it is sent to the decentralized network servers. In order to read this data, applications need to get the encrypted copy from the network and the encryption key from the user. One way to implement a selective disclosure scheme is to use multiple encryption keys. We split the user information into parts and encrypt each part with its own encryption key. To share a specific part of the data, users share a specific encryption key.



With this implementation, the data is encrypted once by an application and then stored. Finally, it can be retrieved by other applications or application instances. Multiple applications can participate in building the data and adding new participants doesn't require changing the stored data. The downside is that sharing an encryption key requires

trust in the recipient application. If an application can't be trusted anymore, the data needs to be re-encrypted and encryption keys need to be shared again with trusted applications.

This architecture requires an entity gathering all encryption keys, something users can trust. This concept is very close to a blockchain wallet but instead of registering private keys, it registers encryption keys. This wallet can generate random encryption keys, record the encryption key usage (which data an encryption key is encrypting) and share them to trusted applications. When an encryption key needs to be changed, the wallet will generate a new one and forget about untrusted ones. However this approach is fairly limited when it comes to secure devices.

Limitation of randomly generated encryption key

Secure devices are only able to keep a limited amount of data inside their storage. The goal is to never share the content of this data and instead perform operations with this data which doesn't reveal it (like signing some data and revealing the signature instead of the private key) inside the device. In our case it would mean that the device is able to receive encrypted data and return unencrypted data and on the other hand receive unencrypted data and return encrypted data. The time necessary to perform these operations is dependent on the amount of data to encrypt or decrypt. The bigger the data is, the longer the operation will take. This approach is mandatory when the use-case requires the utmost security at the expense of the user experience.

The Ledger Key Ring Protocol is designed for applications requiring encryption with a balance of security and ease of use. The secure device is used to manage access to encryption keys instead of an encryption and decryption device. In a sense, the secure device acts like an identity medium, the user "connects" its application to a service by authenticating with the secure device in an anonymous way and chooses which encryption keys to share to the application.

LKRP never shares private keys kept by the secure device storage i.e derived from the user 24 words seed phrase, since the goal here is to share the encryption keys to an "unsafe" space (i.e the key recipient is not necessarily running on a software and hardware primarily focused on security). Instead it must share randomly generated encryption keys to other

applications. This directly outlines one of the requirements of the protocol: the ability to create a copy of the encryption key which can be stored somewhere else and still be read exclusively by the secure device. However generating one key per information can quickly become a heavy operation for a secure device. Since the secure device can't keep all encryption keys inside its limited storage space, the secure device needs to get an encrypted version of the encryption key from an external source, decrypt it and send it to the application. If an application requires hundreds of encryption keys, it will take several minutes for the device to share those keys. As a workaround we chose to base our protocol on the same principles as the BIP 32 standard. Instead of managing encryption keys one by one the protocol uses a derivation tree. It allows us to derive an infinite number of encryption keys from a source encryption key.

Example

To illustrate in which case the protocol can be useful, we'll introduce an example application which can benefit from the protocol. Our example application allows users to create notes on their mobile phone. These notes are then serialized in JSON and sent to a backend server. To maintain privacy and prevent the backend from knowing the content of the notes, we want the mobile applications to encrypt the data before sending it. Moreover a user can share a specific note to another user. Let's consider notes are serialized using the following JSON format:

```JavaScript
{
	"author": "Alice",
	"title": "My Note",
	"content": "This is an example note<br/>",
	"created_at": "2024-06-11T13:36:59.603Z",
	"updated_at": "2024-06-14T08:03:43.212Z"
}
```

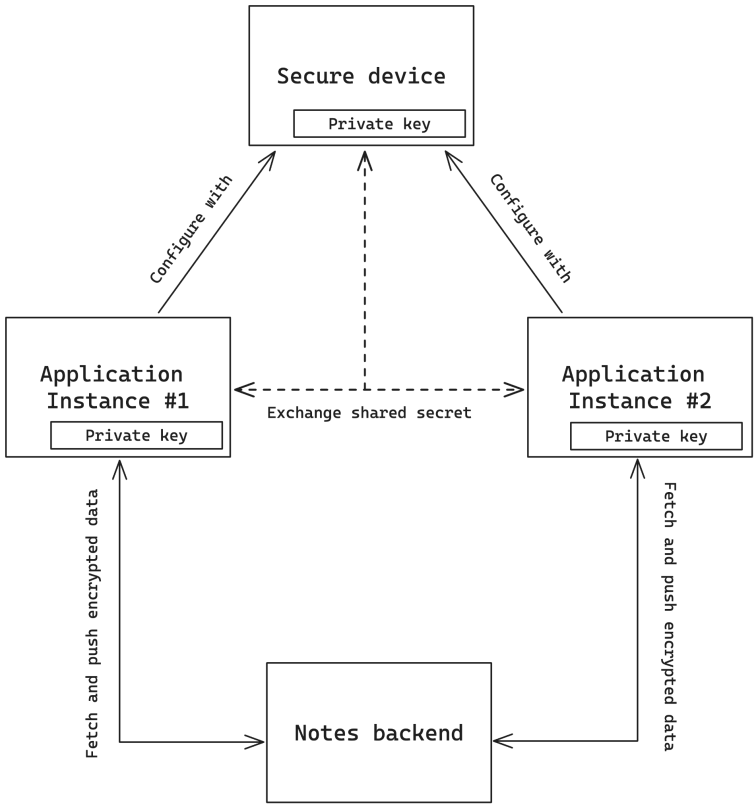
The application allows users to create categories to organize their notes. The application packs all notes in a global object materializing these categories:

```JavaScript
{
	"categories": [
		{
			"name": "Default",
			"notes": [... list of notes ...]
		}
	]
}
```

We want to be able to share parts of the data to other users by encrypting each note with a different encryption key. By sharing an encryption key, other users are able to read notes they did not create.

The application implements the Ledger Key Ring Protocol to exchange shared encryption keys between application instances. These encryption keys are used to encrypt the global JSON object to prevent the backend from reading user data. Mobile phone application instances are responsible for formatting the data and implementing the data synchronization algorithm (merging updates coming from two instances, preventing conflict, handling data versioning...). The backend is only responsible for storing encrypted notes and authentication (making sure an instance has rights to update notes).

The secure device is responsible for managing the "LKRP object," a data structure which acts as a key agreement medium. This object represents a group of entities (secure device, application instances) called **members.** Each member of a LKRP object must own a private key. They are identified inside the LKRP object by their public key and all encryption keys are encrypted specifically for it.



Example of high level architecture for the Notes application

Design goal

The Protocol is designed as a privacy protection tool while providing users ownership over their data. Since its goal is to share encryption keys to devices and applications which are not designed for security, the protocol is not suitable for securing blockchain assets. It offers privacy and convenience by providing the key to encrypt data end to end. Therefore secure devices never share data derived from the user 24 words seed, LKRP encryption keys are always randomly generated and encrypted inside the LKRP object.

Our goal with this protocol is for secure devices to manage access to user data. These devices, designed for security, are often limited in terms of memory and computation power. They are not directly connected to the internet and need an application to send them orders and context. Accesses to a shared encryption key must be verified by the secure device and approved by the user with a trusted display.

Designed for selective disclosure use-cases, the protocol must be able to handle hundreds of encryption keys in such a way that each element of a document can be encrypted with a unique encryption key. Sharing a specific part of the document is equivalent to sharing a specific encryption key.

Finally the protocol must allow common access protocol mechanisms i.e giving and removing access to shared encryption keys.

- Share encryption keys without impacting the security of private keys owned by secure devices
- Manage hundred of encryption keys while running on a secure device
- Identify the data being shared so users are able to review requests before approving them.
- Allow common access processes (allowing and disallowing access)

Notation

Notation	Description
$(.)^{e}$	Ephemeral key
k_{name}	Symmetric encryption key for name
d_{name}	Asymmetric private key for name
P_{name}	Asymmetric public key for name
X_{name}	Extended private key for name
x_{name}	Encrypted extended private key for name
$AsymKeyGen()$	Key-pair generation function
$PrivToPub(d_{name})$	Compute a public key from a private key
$Random(n)$	Generate *n* random bytes
$M1 \mid\mid M2$	Concatenation of messages *M1* and *M2*
$Hash(M)$	Hash message *M*
$Sign(d,\ M)$	Signature of message *M* using private key *d*
$ECDH(d,\ P)$	Elliptic curve Diffie-Hellman key agreement between private key *d* and public key *P*
$CKDpriv(parent,\ index)$	Compute a child extended private key from the parent extended private key (*parent*) using the given *index*
$BIP32(X_{name},\ path)$	Derive extended private key X_{name} using BIP32 $path$ to create a child extended private key

$\{\cdot\}_K$	Encryption using key K
$\{\cdot\}_K^{-1}$	Decryption using key K

Algorithms

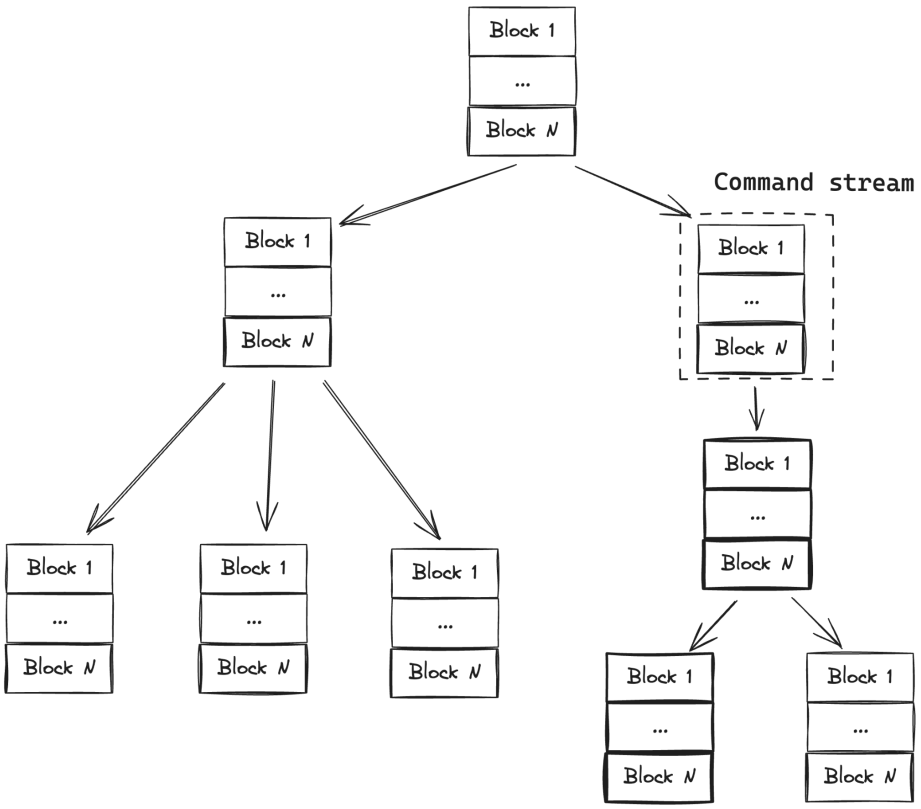
Operation	Algorithm
Hash	NIST FIPS 180-4 Secure Hash Standard (SHS) SHA-256
Key agreement	NIST Special Publication (SP) 800-56A Rev. 3, Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography ECDH over secp256k1
Signature	NIST FIPS 186-5 Digital Signature Standard (DSS) ECDSA over secp256k1
Symmetric encryption	AES-256-GCM
Private key derivation	BIP32

LKRP object

The previously introduced "LKRP object", is both used as a communication and backup medium. It contains encrypted data and the list of members having rights to read those data. The "LKRP object" is composed of 2 types of data structures:

- A list of "commands" called *Command Stream*
- A tree where each node is a *Command Stream*

A *command stream* contains a single shared encryption key and a list of members. Members are identified by their public keys (over secp256k1). A member of a *command stream* has access to the shared encryption key and is able to read it. The *LKRP tree* is the top-level object of the protocol. It contains all *command streams*. Its main purpose is to organize and identify shared encryption keys..



LKRP tree and command streams

Command stream

A *Command stream* is a list of blocks. Blocks are created by an *issuer,* identified by its public key. A block contains a reference to its parent block, a list of commands and a signature. The integrity of a *command stream* can be verified without any additional information than the command stream itself. A software willing to verify the integrity of a *command stream* needs to read each block one by one in their creation order, verify if the referenced parent hash matches the hash of the parent block, and verify the signature signed. Any member of a *command stream* can issue the block except for the first block where the *issuer* is considered as the owner of the command stream.

The table below describes the commands which can be part of a block:

Command	Description
Seed	Indicates the generation of a random root encryption key. The content of this command is encrypted for the issuer of the command. This command must be the first command of the first block of the root command stream.
Derive	Indicates the derivation of a child command stream. The content of this command is encrypted for the issuer of the command. This command must be the first command of the first block of a child command stream.
AddMember	Add new member to the *command stream*
PublishKey	Publishes the encryption key to a member. This command is only allowed if a *AddMember* command was previously issued. The content of this command is encrypted for the command recipient.
CloseStream	Indicates the *command stream* is closed. A closed stream should not be used anymore and its encryption key is considered unsafe to use.

Command streams are expected to start with either a *Seed* or *Derive* command. By parsing the list of blocks, the secure device can perform integrity checks. It can verify if a block issuer was previously added to the stream, if keys are only published to added members, if the stream start with a *Seed* or *Derive* command, verify if blocks are parsed in creation

order (by check the parent hash), verify if the issuer actually created the block (by checking the block signature).

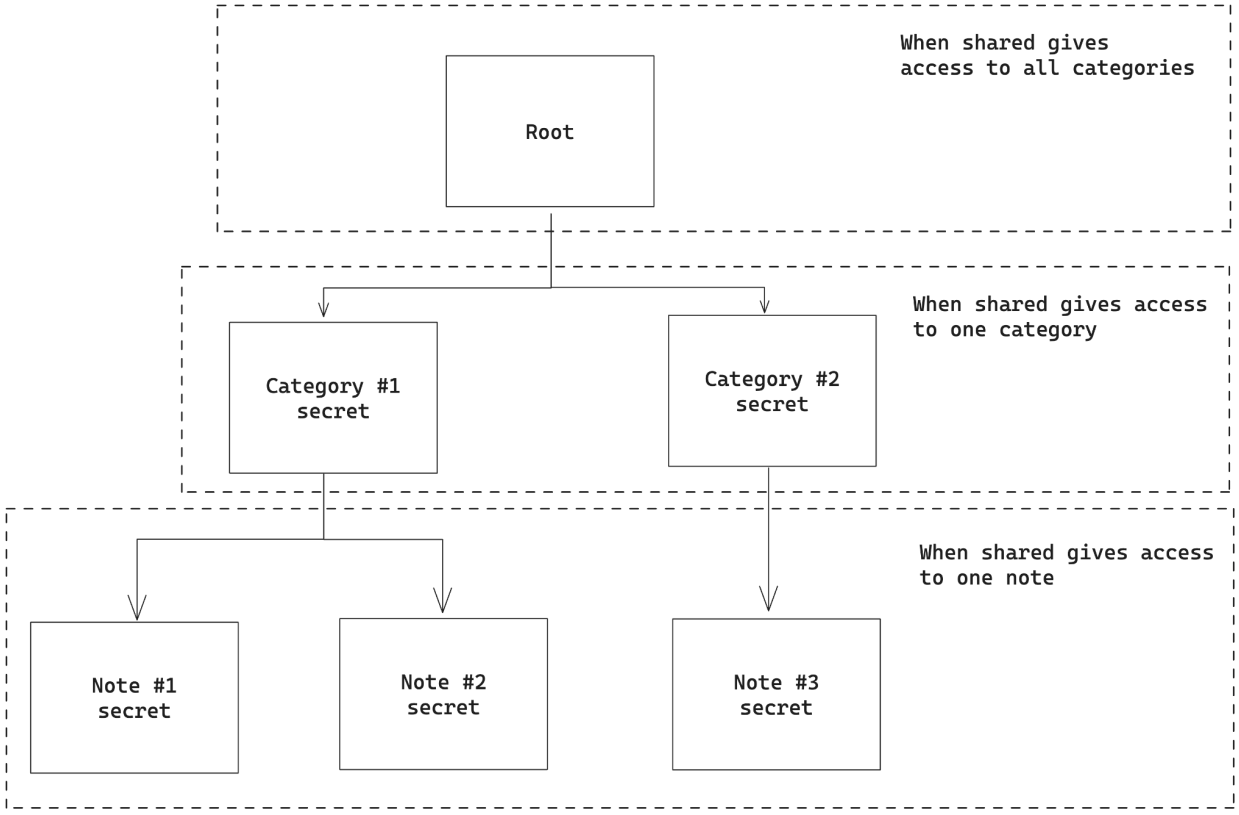
LKRP tree

A first approach for sharing an encryption key to a trusted application would be to generate a random encryption key on the secure device and encrypt it for the trusted application. This works when the number of encryption keys is fairly limited and we could use a single command stream to do it. The first command would act as a backup for the secure device (*Seed*), and the secure device would share the encryption key with the *AddMember* and *PublishKey* commands. But if an application requires more encryption keys, it will quickly hit a wall because the time for the secure device to encrypt encryption keys will constantly grow, leading to poor user experience where each time users try to perform an action, they will need to wait several seconds. The protocol can't help a secure device encrypt keys faster but it can help optimize the way the application shares its keys.

Let's suppose we would like to implement the note application presented in the introduction. For this application we would like to send notes to a storage service with an end to end encryption scheme where the storage service would not be able to read the content of the user data. We would like to implement a feature allowing other users to be able to edit specific notes or having access (read and write) to a full category. A simple solution for this would be to encrypt each note with an encryption key and when a user wants to share a note to another user it will simply share the encryption key. On the other hand, if the user wants to share a category it would share one encryption key per note (for the example we will pass on the issue of adding a note in the category and letting the application know about the new encryption key for the added note). In this case the more a user has notes the more the secure device will be slow, the more the user uses the application the more the user experience worsens.

This problem can be solved by using key derivation algorithms allowing to create multiple encryption keys from another one. The Ledger Key Ring Protocol uses BIP32, a well known bitcoin standard used to create hierarchical deterministic wallets. By using derivation algorithms, we can structure data for sharing purposes by grouping information. For our "note application", we would create a parent key for each category, then we would derive (from the parent encryption key) one key per note. If a user wants to share a single note,

we would share a child key. If a user wants to share a category we can share the parent key.



The tree structure simplifies key sharing however it brings 2 new challenges:

- How to change an encryption key ?
- How to identify the resource being encrypted ?

Key rotation & identification

By sharing a key to a third party, a user trusts this third party at a given time. The user can however at some point not trust this third party anymore or worse the third party gets compromised. At this point there is no way to make the third party or the hacker forget about the encryption key, the only way forward is to use another encryption key and encrypt the data with this new key. This is simple for random keys, you just need to generate a new random key and share it. However this is not a very practical solution for a deterministic tree because all keys are linked together. If an application changes the

random key at the root, it changes all child keys and forces to re-encrypt the whole dataset whereas in the best case solution if a child key gets compromised the application only rotates the compromised key and re-encrypt the associated data.

Another challenge, mainly in the context of secure devices with trusted screens, is to identify the purpose of the encryption key i.e. the type of data a key encrypts. When a secure device needs to perform an action with a key it must first ask for the user's approval. This pattern is secure as long as users understand what they are approving.

Both these issues are addressed by the Ledger Key Ring Protocol, by enforcing a specific derivation layout. Derivations in the BIP32 standard are expressed using "derivation paths''. A path is a series of indexes, each index is used by the algorithm to create a new key. For example the derivation path "m/16h/2h" would translate to

$$CKDpriv(CKDpriv(source,\ 0x80000000 \mid 16),\ 0x80000000 \mid 2) \ \Leftrightarrow\ BIP32(source,\ m/16h/2h)$$

where *source* is a random extended private key. By using a common way of performing the derivation we can give special meaning to specific parts of a derivation path and build them accordingly.

The derivation policy of the Ledger Key Ring Protocol is built around pairs of indexes each pair is composed of a level designed for key identification and another index used for rotating encryption keys. Derivation path are built using the following policy:

`m/SeedRoot(h)/AppSeed(h)/AppRoot(h)/ObjectSeed(h)/ObjectRoot(h)/…`

- *m* is the root of the tree, used to derive every single encryption keys
- *SeedRoot* is the level used for rotating the whole tree
- *AppSeed* is the level used to identify a domain of application in the tree.
- *AppRoot* is the level used to rotate the domain of application sub-tree.
- *ObjectSeed* level identifies an object i.e. the data being encrypted by the *command stream.*
- *ObjectRoot* level is used to rotate the object sub-tree or *command stream*

After the *AppRoot* level there can be multiple layers of *ObjectSeed* and *ObjectRoot* pairs, depending on the data structure being encrypted. Note that the Ledger Key Ring Protocol

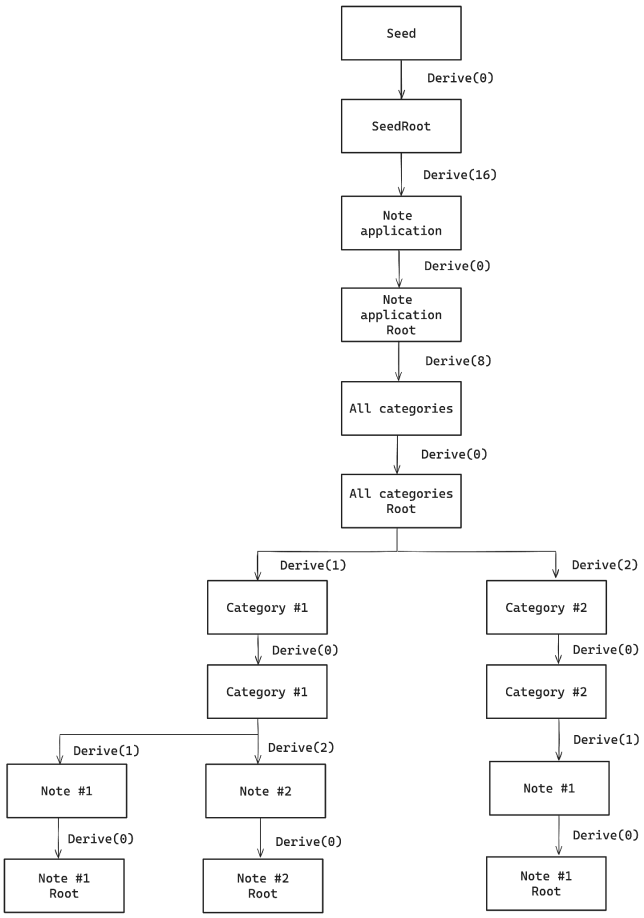
only uses hardened derivation (expressed by a *h* at the right of the index). We want to prevent the use of extended public keys (even if they would be convenient for consistency checks). Extended public keys would introduce a weakness in the protocol because an actor knowing both a parent extended public key and a child private key can recover the parent extended private key. Since our goal is to share children private keys, non-hardened derivations would allow access to private keys which were not allowed by the user.

All levels suffixed with *"Seed"* are used for identification, these indexes are specified by the application developer. We can define a stable ID for the encrypted resource by only taking into account the indexes from this level. For example the *command stream* derived "m/0h/16h/0h/1h" has "m/16h/1h" as its stable ID. In order to display this information to the user, the secure device requires a mapping table to transform the stable ID to a human readable string. Private keys derived at these levels must not be used to encrypt data, since this level is used to identify a resource, if the application needs to change the encryption it would also need to change its identification.

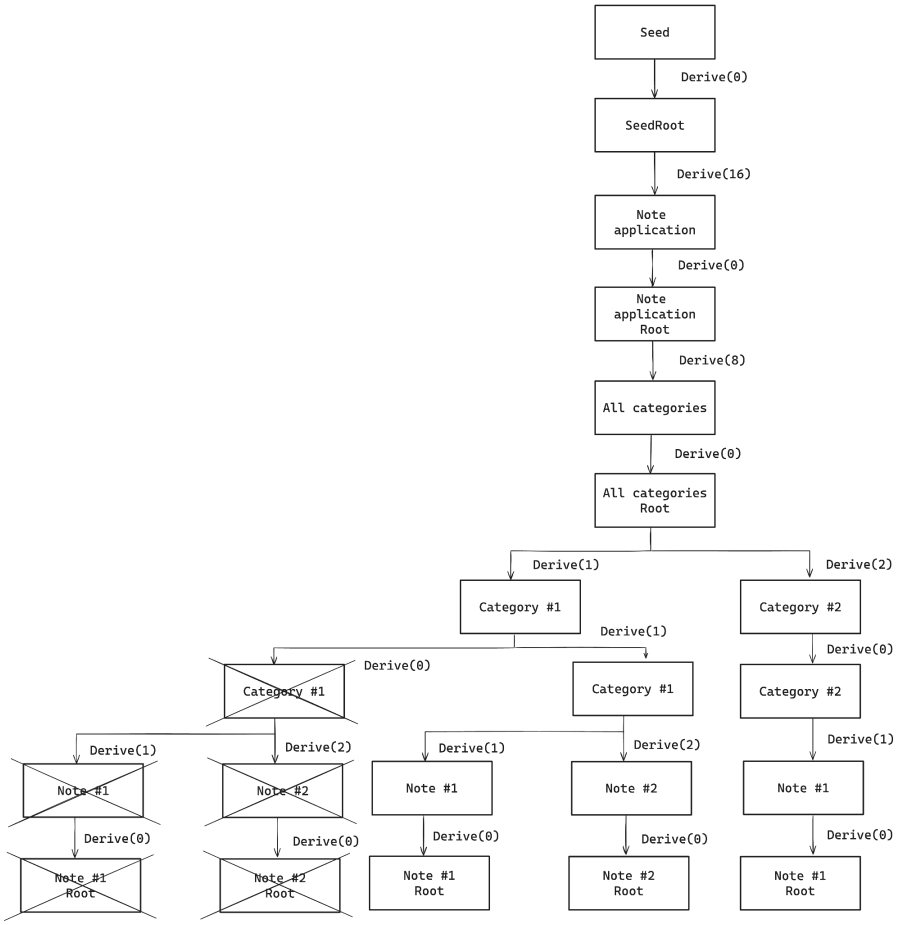
The rotation level (suffixed with *"Root"*) is used to change the encryption key of a given resource. It is expected that the index at these levels start at 0 and are incremented each time a new version of the key is created. Private keys derived at these nodes are used to encrypt user data.

For example, the "Notes" application needs one encryption key per note and we want to be able to share all notes from a category at once. The application creates the tree root which contains a random main encryption key (technically a private key used to encrypt data via ECDHE). From the tree root we choose 16 as our application identifier and immediately create the first rotation at index 0. We create a node at index 8 which will contain all categories so we still have the option to share all categories at once. We don't directly share the app level, to allow future types of data to be added to the tree. Now we are able to derive one node for each category from the "m/0h/16h/0h/8h/0h" derivation path such as the first category will be derived at "m/0h/16h/0h/8h/0h/1h/0h", the second category derived at "m/0h/16h/0h/8h/0h/2h/0h" and so forth. Then we derive one node per note such as the encryption key for the first note in the first category will be at "m/0h/16h/0h/8h/0h/1h/0h/1h/0h", the second note in the first category will be at "m/0h/16h/0h/8h/0h/1h/0h/2h/0h"... Now the application can share the leaf of the tree to

share a single note. If the application needs to share a category, it can share one of the nodes at "m/0h/16h/0h/8h/0h". By sharing a "category" node, an application can compute all leaves and decrypt notes.



Let's assume we want to share the first category with another user. The application will add this new user to the node derived at "m/16h/0h/8h/0h/1h/0h". New members should always be added to nodes which can be rotated otherwise this new member can't be removed without rotating the entire tree and re-encrypting the whole data set. At this point our new user is able to derive all leaves and get encryption keys for all notes in the first category. If, at some point, we don't want our new user to have access anymore we need to rotate the category node. First we derive a new node for the first category by incrementing by one the index of the node, the new node for the first category will be "m/16h/0h/8h/0h/1h/1h". Then we can derive all leaves of the "m/16h/0h/8h/0h/1h/1h" subtree to get the encryption keys. Finally we can re-encrypt all notes of the first category.



Note that the application doesn't need to create all nodes every time. Creating a node is only useful for sharing its encryption key with another member. The root of the tree alway needs to be created, all child nodes need to reference the hash of the first block of the root. Otherwise the application creates nodes depending on its encryption needs. If the application needs to encrypt all categories, it will query the secure device to create the node derived at "m/0h/16h/0h". With this node the application can derive everything else and encrypt the data set with unique encryption keys. On the other hand, if the application want to share the first note of the second category to another application, it will create the node at "m/0h/16h/0h/8h/0h/1h/0h/2h/0h" and add this other application in the node.

Block structure and commands

Blocks

Blocks are structured in 3 parts:

- A block header composed of the following elements:
  - **version**: specifies the protocol version used to create the block. It is expressed as a 8 bits unsigned integer
  - **parent**: the hash of the parent block. The length of the hash is 256 bits.
  - **issuer**: the public key of the block issuer in its compressed form. The length of a compressed public key is 264 bits.
  - **length**: the number of commands in the block. Expressed as an 8 bits unsigned integer.
- A list of commands
- A signature to verify the integrity of the block and allow applications to verify if the block issuer is allowed to issue blocks. (see Block signature for details)

Blocks are serialized using a TLV (Type Length Value) encoding where each value is encoded with a 8 bits type tag followed by a 8 bits length (the length in bytes of the value) and the value (see TLV Specifications for details).

Once encoded, each block is concatenated in order (parent block followed by its direct child) to form a CommandStream.

TLV specification

Blocks are serialized using a TLV format (Type-Length-Value). Type tags and length are encoded on a single byte. The TLV format uses the following scalar types:

Type	TLV type tag	Description
`NULL`	`0x00`	Represent an empty field
`VARINT`	`0x01`	Represents any type of integer value (Big endian)
`HASH`	`0x02`	A hash

SIG	0x03	A signature
STRING	0x04	UTF8 encoded string
BYTES	0x05	Byte array
PUBKEY	0x06	DER encoded compressed public key

Blocks are built by using the above scalar types. Each field is expected to be found in the order defined in the table below when parsing a block

Field	Type	Length	Description
Version	VARINT	1	Format version
Parent	HASH	32	The hash of the parent command block (Random 32 bytes for the first command block)
Issuer	PUBKEY	33	The public of the issuer (in its compressed form)
Length	VARINT	1	Number of command in this command block
Commands			
Signature	SIG	variable	Signature of the command block

Each command uses a specific type tag described below. Commands are encoded as TLV sub-objects. To encode a command inside a block one must encode the command then prefixing it with the command type tag and its length (the length of the encoded command).

Command	Type tag	Description
Seed	0x10	The root command of a command stream tree. Generates a random main key for the tree and encodes it for the tree owner.
Derive	0x15	The alternative root of a command stream, derives a child stream from the root command stream. The block containing this command must use the hash of a block containing the Seed command as parent.

`AddMember`	`0x11`	Add a new member to the group. The new member of the group cannot have more permissions than the issuer (it can have the same rights, or less rights)
`PublishKey`	`0x12`	Publish an encrypted key to a single recipient.
`CloseStream`	`0x13`	Closes the current stream. After this command the stream cannot accept new commands and the underlying encryption key is considered deprecated.

Seed command			
Field	**Type**	**Length**	**Description**
`Topic`	`BYTES`	16 (max)	Context/rules used to enforce derivation and sharing rules for the tree
`Protocol version`	`VARINT`	2	Version of the protocol used to serialized the block
`Group key`	`PUBKEY`	33	The public key of the group
`Initialization vector`	`BYTES`	16	The Initialization vector used for the encryption of xpriv (must be random)
`Encrypted xpriv`	`BYTES`	80	The private key (32 bytes) followed by the BIP32 chain code (32 bytes). This payload is encrypted for the issuer of the block. "xpriv" stands for extended private key from the BIP32 standard.
`Ephemeral public key`	`PUBKEY`	33	An ephemeral public key used to encrypt the xpriv.

Derive command			
Field	**Type**	**Length**	**Description**
`Path`	`BYTES`	var	The derivation path from the root

Group key	PUBKEY	33	The public key of the group
Initialization vector	BYTES	16	The Initialization vector used for the encryption of xpriv (must be random)
Encrypted xpriv	BYTES	80	The private key (32 bytes) followed by the BIP32 chain code (32 bytes). This payload is encrypted for the issuer of the block. "xpriv" stands for extended private key from the BIP32 standard.
Ephemeral public key	PUBKEY	33	An ephemeral public key used to encrypt the xpriv.

AddMember command			
Field	**Type**	**Length**	**Description**
Name	STRING	var (max 20)	The name of the member
Public key	PUBKEY	33	The public of the member to add
Permissions	VARINT	4	The set of permission for the new member. Current protocol only accepts the OWNER permission sets at 0xFFFFFFFF

PublishKey command			
Field	**Type**	**Length**	**Description**
Initialization vector	BYTES	16	The Initialization vector used for the encryption of xpriv (must be random)
Encrypted xpriv	BYTES	80	The private key (32 bytes) followed by the BIP32 chain code (32 bytes). This payload is encrypted for the recipient of the command. "xpriv" stands for extended private key from the BIP32 standard.
Recipient	PUBKEY	33	The public key of the recipient. The public

			key must have been previously added to the chain (via a AddMember command)
`Ephemeral public key`	`PUBKEY`	33	An ephemeral public key used to encrypt the xpriv.

CloseStream command
The value of this command is empty

Commands are serialized, then prefixed with the appropriate tag and length before being concatenated to the block.

Python

```
VARINT_TAG = 0x01
STRING_TAG = 0x04
PUBKEY_TAG = 0x06
BYTES_TAG = 0x05
HASH_TAG = 0x02
SIG_TAG = 0x03
ADD_MEMBER_TAG = 0x11

version = bytes([0x01])
parent_hash = bytes([0xef, 0x01, 0x22, 0xe9, 0xa1, 0x11, 0xf4, 0x87, 0xef
, 0x01, 0x22, 0xe9, 0xa1, 0x11, 0xf4, 0x87, 0xef, 0x01, 0x22, 0xe9, 0xa1, 0x11
, 0xf4, 0x87, 0xef, 0x01, 0x22, 0xe9, 0xa1, 0x11, 0xf4, 0x87])
issuer_public_key = bytes([0x03, 0xba, 0xe5, 0xc6, 0x6f, 0xdd, 0xcb, 0x39
, 0x55, 0x9a, 0x15, 0xf4, 0xb3, 0x76, 0xd6, 0x8b, 0xb1, 0xff, 0xf0, 0xf1, 0x3e,
0xed, 0x73, 0x5e, 0xf5, 0x85, 0xa0, 0x74, 0x55, 0x59, 0x38, 0x40, 0x4b])
command_count = 1
member_public_key = bytes([0x03, 0xaa, 0xb2, 0xc8, 0x6f, 0xdd, 0xcb, 0x39,
 0x55, 0x9a, 0x15, 0xf4, 0xb3, 0x76, 0xd6, 0x8b, 0xb1, 0xff, 0xf0, 0xf1, 0x3e,
 0xed, 0x36, 0x91, 0x17, 0xa5, 0xb8, 0x64, 0x44, 0x87, 0x67, 0x88, 0x55])
member_name = "Alice"
permissions = bytes([0xff, 0xff, 0xff, 0xff])

# Encode block header
block = bytes([VARINT_TAG, len(version)]) + version
```

```
block += bytes([HASH_TAG, len(parent_hash)]) + parent_hash
block += bytes([PUBKEY_TAG, len(issuer_public_key)]) + issuer_public_key
block += bytes([VARINT_TAG, 1, command_count])

# Encode commands
add_member = bytes([STRING_TAG, len(member_name)]) + bytes(member_name,
 'UTF-8')
add_member += bytes([PUBKEY_TAG, len(member_public_key)]) + member_public_key
add_member += bytes([VARINT_TAG, len(permissions)]) + permissions
block += bytes([ADD_MEMBER_TAG, len(add_member)]) + add_member

# Sign and encode signature
...
```

Tree, branch and block identifiers

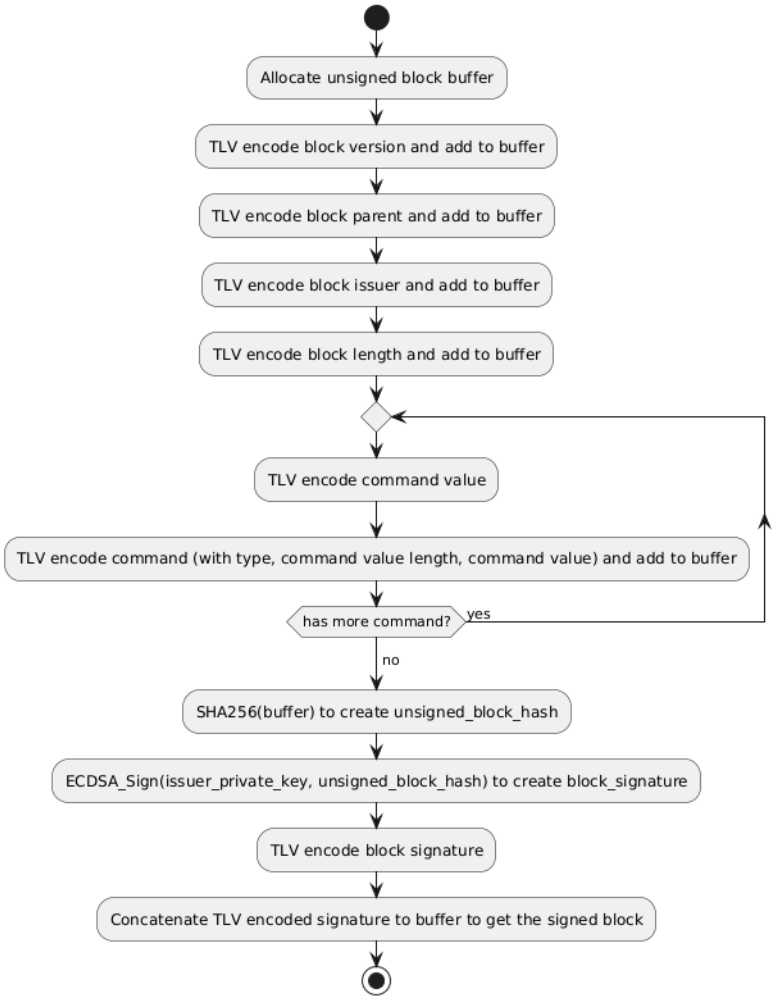
A block is identified by its hash, this hash is computed by serializing a block in TLV and executing the SHA-256 hash function. Block hashes are used to identify different objects:

- The tree is identified by the hash of the block containing the Seed command. This identifier is stable since the root block is immutable and changing the root block basically means creating a new tree
- A branch is identified by the hash of a block containing the Derive command. This identifier can be used to identify a sub-tree.
- Parent hash is used when creating a new block, it helps to maintain the integrity of the chain.

Block signature

Blocks are signed using the private key of the issuer of the block. In order to sign a block, the application first encodes the unsigned block using the TLV specification. Then the unsigned block is hashed (using SHA-256) and signed by the private key.

Finally the signed blocked is encoded by concatenating the TLV encoded unsigned block and the signature (TLV encoded)



Tree initialization

Before encrypting data, the application needs to create a main private key (seed) for the tree. This operation is done by the secure device. The secure device randomly generates a random extended private key.

$$(d_{seed}, P_{seed}) = AsymKeyGen()$$

$$C_{seed} = Random(32)$$

$$X = d_{seed} \,||\, C_{seed}$$

The secure device generate P_{seed} from d_{Seed} to compute the command stream public key (called Group key in the TLV specification). Later on this public key can be used by new members to verify the private key they receive from the *PublishKey* command. P_{seed} is serialized inside the Seed command. The next step is to encrypt X with ECDHE. The secure device generates an ephemeral private key and computes a symmetric key between the ephemeral public key and the private key of the block issuer (d_{issuer}) to finally encrypt X.

$$(d_c^e, P_c^e) = AsymKeyGen()$$

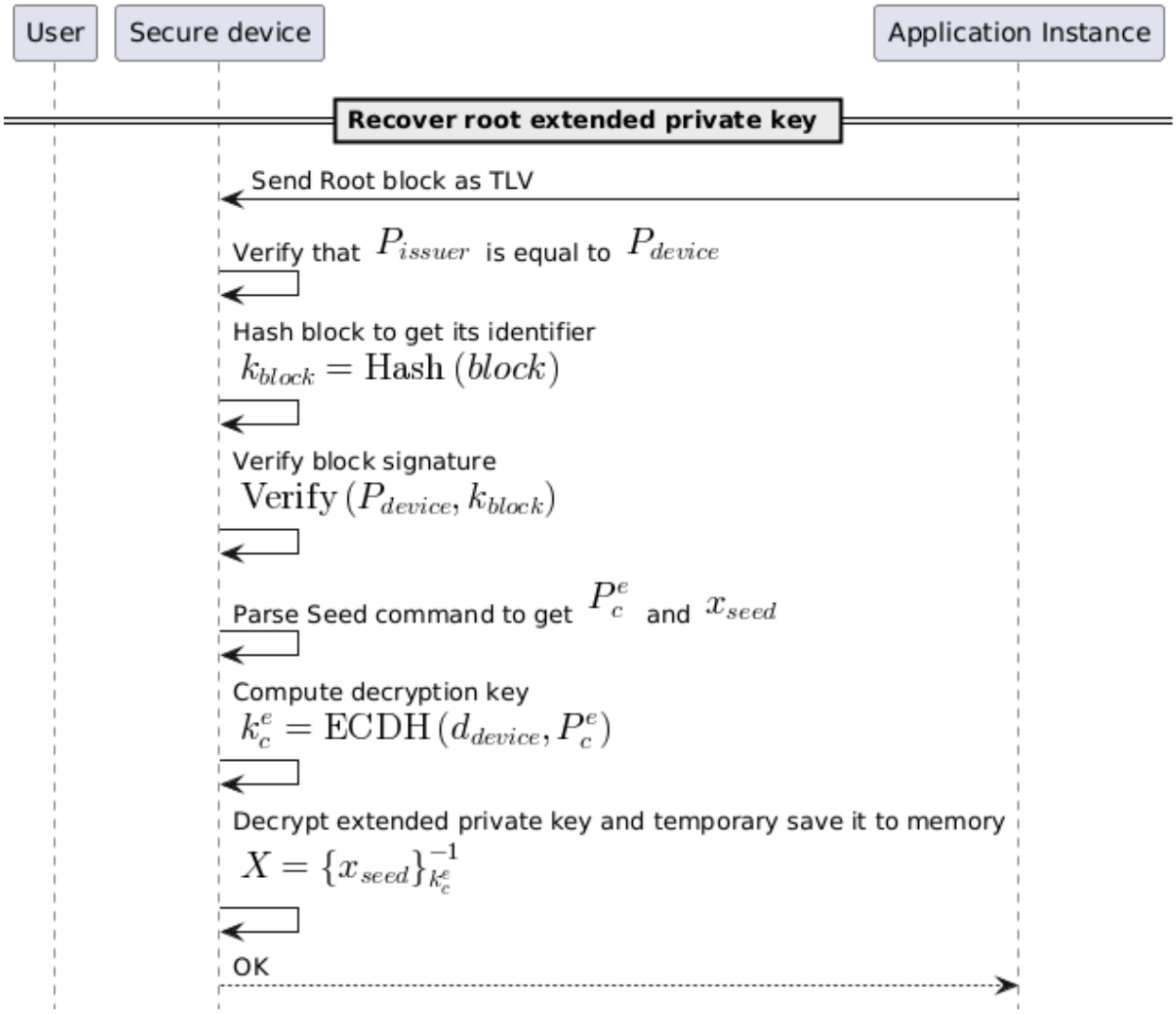
$$k_c^e = ECDH(d_{issuer}, P_c^e)$$

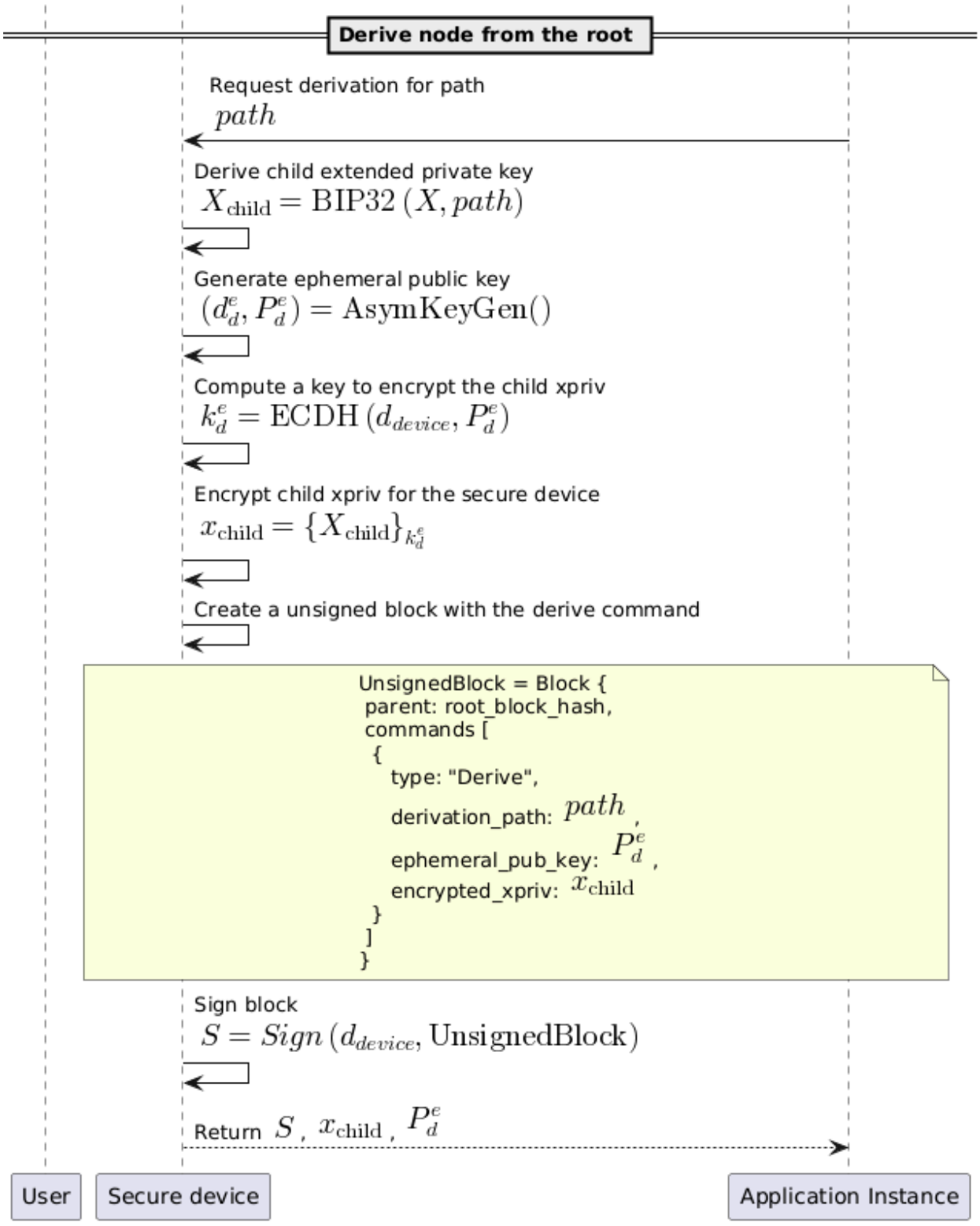
$$x_{seed} = \{X\}_{k_c^e}$$

P_c^e and x_{seed} are finally serialized inside the Seed command payload. The Seed command is used as a backup for the secure device to retrieve the private key. Since we don't want to share internal private keys, we create a random private key, create a backup and read the backup when needed.

Node derivation

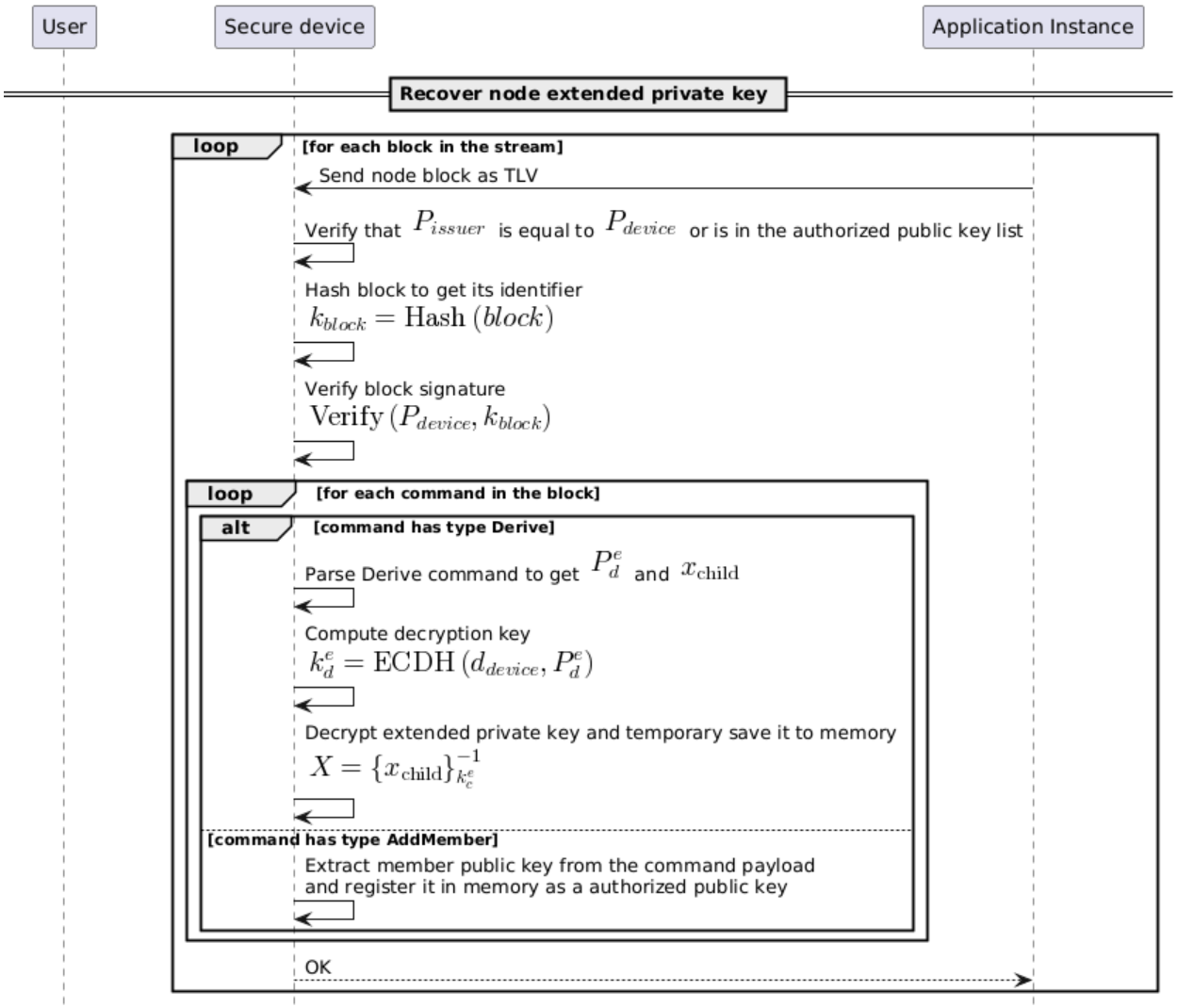
Once the root of the tree is created, applications require one or multiple nodes to work. First the application sends the root command stream to the device to recover the root extended private key. Then the application requests the device to create a new block with the requested derivation.




Derive node from the root
Request derivation for path
$path$
Derive child extended private key
$X_{\text{child}} = \text{BIP32}\,(X, path)$
Generate ephemeral public key
$(d_d^e, P_d^e) = \text{AsymKeyGen}()$
Compute a key to encrypt the child xpriv
$k_d^e = \text{ECDH}\,(d_{device}, P_d^e)$
Encrypt child xpriv for the secure device
$x_{\text{child}} = \{X_{\text{child}}\}_{k_d^e}$
Create a unsigned block with the derive command
UnsignedBlock = Block {
parent: root_block_hash,
commands [
{
type: "Derive",
derivation_path: $path$,
ephemeral_pub_key: P_d^e,
encrypted_xpriv: x_{child}
}
]
}
Sign block
$S = Sign\,(d_{device}, \text{UnsignedBlock})$
Return S, x_{child}, P_d^e
User
Secure device
Application Instance

Sharing a node encryption key to a third party application

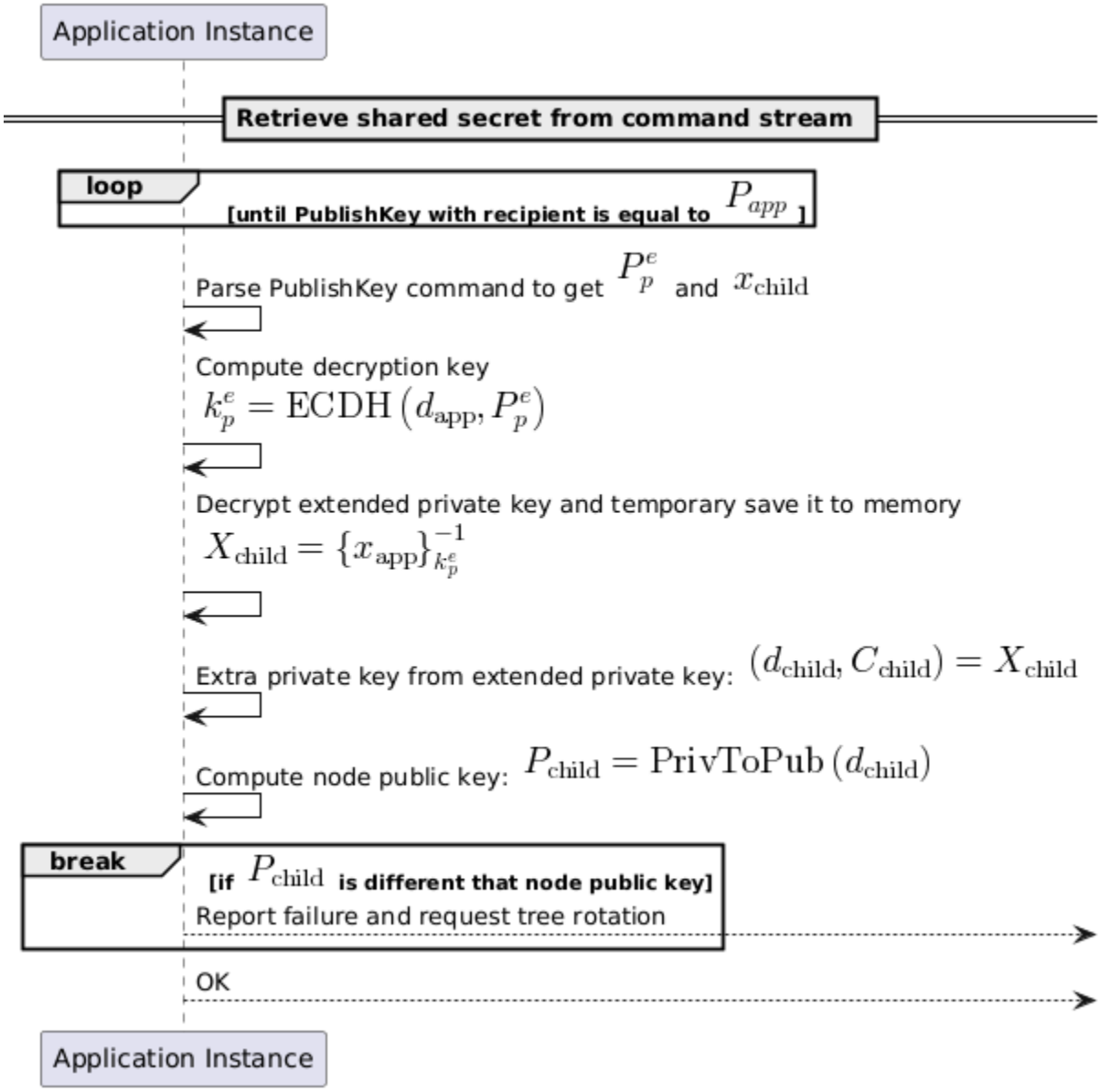
Sharing a node encryption key to another application is done by adding a AddMember command and PublishKey command to the stream. This can be done in one or two blocks. First the application streams the node command stream to the secure device to allow the secure device to temporarily keep the node encryption key in memory. Then the secure device encrypts the private key for the new member.



The application instance can finally decrypt the node encryption key. If the key was not published by the same issuer as the issuer of the derivation block, the instance can compute the group public key and verify if its computed public key is equal to the group public key.

Retrieving a shared key from member

Third-party applications can retrieve their keys by reading the command stream. They must iterate through all blocks until they find the PublishKey command which was issued for them. Finally they can decrypt its content with their public key.



Conclusion

We learned three major takeaways from this paper: first, how the Ledger Key Ring Protocol can be used to efficiently share multiple encryption keys from a secure device, second how the tree structure helps to manage a great deal of encryption keys while preserving the limited resources of secure devices, and third how the derivation policy helps to identify the underlying data and allows it to rotate encryption keys.

The Ledger Key Ring Protocol acts like a certificate chain, where a command stream can be used to prove that an application instance is linked to a secure device. This architecture can bring end-to-end encryption to service for encryption, and authentication of instances all linked to a secure device which acts like an identity token for the user.

This is a new era for end-to-end encryption, privacy, and decentralized use cases built on Ledger's secure technology.