Carmentis Protocol Specification

Version 1

Authored by

PhD Gaël MARCADET

Abstract This document introduces the formal specification of the Carmentis cryptographic protocol. This document explains the security model in which the protocol is considered secure, introduces how the protocol is constructed and provide arguments on the security of the protocol.

Work in progress: The presented whitepaper is currently in progress and has *not* been proven secure yet.

Contents

List of Figures

List of Tables

Abstract

The rise of artificial intelligence has revolutionised numerous domains but has also introduced significant challenges, particularly in the realm of document authenticity. With AI-generated content becoming increasingly convincing, the trust paradigm around documents has shifted. Today, a document is often presumed false unless its authenticity can be unequivocally established. This new dynamic underscores the pressing need for robust mechanisms to verify and maintain trust in digital documents.

One straightforward solution to this challenge is the use of a central authority to sign and validate documents. While effective, this approach inherently relies on a trusted third party, which may not always be desirable or feasible in decentralised settings. An alternative, more decentralised approach is exemplified by systems like OpenPGP, where users employ personal signature keys to sign emails or other documents. However, such systems have a critical limitation: they depend on the preservation of signatures. To authenticate older documents, the associated signatures must remain intact and accessible for extended periods, necessitating reliance on a trusted entity to store these signatures securely without risking erasure.

To address these limitations, blockchain technology offers a promising solution. By recording signatures within a blockchain, reliance on a single trusted third party is mitigated. Instead, trust is distributed across a decentralised network, making signatures resilient to tampering or loss. However, blockchain-based systems are not without challenges.

A particularly complex issue arises when it is necessary to prove the authenticity of a specific subset of data while maintaining the confidentiality of the remaining information. In such cases, traditional blockchain-based signature schemes fall short, as they lack mechanisms to balance data privacy with selective proof of authenticity. This scenario calls for more sophisticated solutions that ensure both the confidentiality of sensitive data and the verifiable authenticity of a chosen subset.

Our contribution We propose to address this challenge by introducing a new cryptographic, blockchain-based protocol allowing at the same time confidentiality and selective proof of authenticity. At the heart of our technical contribution to achieve these properties, is a novel dedicated architecture, at the intersection of companies-oriented centralised model for usage convenience and decentralised blockchain model to prevent centralised trusted authority. For a better understanding of our solution, we elaborate more from different perspectives.

- *Blockchain-based solution*: The protocol we have designed is based on a blockchain in which all transactions (*e.g.,* digital signature of a signed contract, officially sent messages) are anchored. The blockchain acts as a virtual centralised third-party storing information without tampering them, a crucial feature to provide trust. Moving towards a decentralised system, each user in a company manages its own key pair using a wallet, moving further into a completely decentralised protocol.

- *Company operating model*: In contrast with the decentralised blockchain serving all parties, a company is interested in having the control of their private data, a crucial necessity when these private data are used to derive information being stored in the blockchain, as we will see later. For this reason, we have included in our protocol an additional multi-purpose server, later called an *operator*. This operator is the cornerstone of a company within the Carmentis protocol, since it acts as a trust party by signing data, as the party paying fees to make the Carmentis blockchain network active, as a data encryptor and as a key manager to allow stake holders of the company to access the entire data, *or a subset* of these data. We elaborate more of this aspect below.

- *Fine-grained data access*: The protocol has been designed in opposition to the *all-or-nothing* paradigm in which a user either has access to the entire data, or nothing. Instead, when the user publishes (potentially private) data to the operator, following a predefined set of access rules, the operator encrypts as many as the needed subset of the data accordingly to the access rules, using a dedicated encryption key. This encryption is later shared with users having the appropriate role with respect to the access rules.

- *Privacy-preserving proof of authenticity*: The major feature of our protocol is the ability for a user to prove the authenticity of data. This feature is described in three distinct steps: First, from the entire set of data, the prover generate an information called a commitment. Second, the prover generates the proof of authenticity. Third and last, given the proof and the commitment, the proof verifier accepts or rejects the proof. The generated proof is said to be accepted with respect to the commitment on which both the prover and the verified agreed on. This crucial condition is achieved practically in our protocol thanks to the decentralised blockchain, storing the commitment without offering the possibility to tamper it. The proof of authenticity can be used not only to prove the authenticity of a data, but also a subset of them, without revealing any information on the data not in this subset. This privacy-preserving proof only reveal the required data that should be proved to be authentic and nothing more.

- *Users anonymity*: The protocol has been designed to preserve anonymity of users, meaning that one cannot recover the identity of a user by observing (externally its interaction).

Outline In Chapter 1, we present an informal description of the Carmentis protocol, introducing the major concepts behind the protocol. In Chapter 2, we formally introduce the cryptographic concepts used to construct the protocol.

Chapter 1

Overview of Carmentis protocol

In this chapter, we present an overview of the key concepts serving as the building blocks for the Carmentis protocol step-by-step, serving a formal, yet easy-to-understand, landing on the protocol.

1.1 Basic cryptographic concepts

First, we recall some concepts being at the core of the protocol. For the moment, we introduce these concepts separately, before to put all together in the following sections.

1.1.1 Secret-key encryption

The first concept we cover in this section is the secret-key encryption. Secret-key encryption, also known as symmetric encryption, is a cryptographic technique where the same key k is used for both encrypting and decrypting information. This method is widely used for secure communication because of its simplicity and efficiency. The security property of an encryption scheme defines how well the scheme protects the confidentiality and integrity of the data against adversaries. In general, encryption security is evaluated by how difficult it is for an unauthorised entity to decipher the plaintext or gain useful information without the key.

1.1.2 Digital signature

The second concept introduced in this section is digital signature, which is a cryptographic mechanism used to ensure the authenticity, integrity, and non-repudiation of a digital message or document. It is analogous to a handwritten signature or a stamped seal, but it is much more secure due to its cryptographic basis. Unforgeability of a digital signature scheme ensures that only the legitimate owner of the private signing key sk can create a valid digital signature for a given message, making it computationally infeasible for anyone else to forge a signature, even with access to the public verification key pk. Public verifiability allows anyone with the signer's public key to independently verify the authenticity of the signature, confirming that it was created by the rightful private key holder and that the message was not altered. Non-repudiability ensures that the signer cannot deny having signed the message, as the digital signature uniquely binds the signer to the message, and only the private key they control could have produced it.

1.1.3 Cryptographic hash function and Merkle-tree

The third and last concept introduces a key mechanism in our protocol, called cryptographic hash function, being a mathematical algorithm that transforms an input, or "message," into a fixed-size string of characters, which is typically a sequence of numbers and letters called the "hash". It is designed to be a one-way function, meaning it is computationally infeasible to reverse the process and retrieve the original input from the hash value. Cryptographic hash functions have several key properties: They produce a fixed-size output regardless of the input size, are deterministic (the same input always produces the same hash), and exhibit strong resistance to collisions, meaning it is computationally infeasible to find two different inputs a and b that result in the same hash value, *i.e.,* $H(a) = H(b)$.

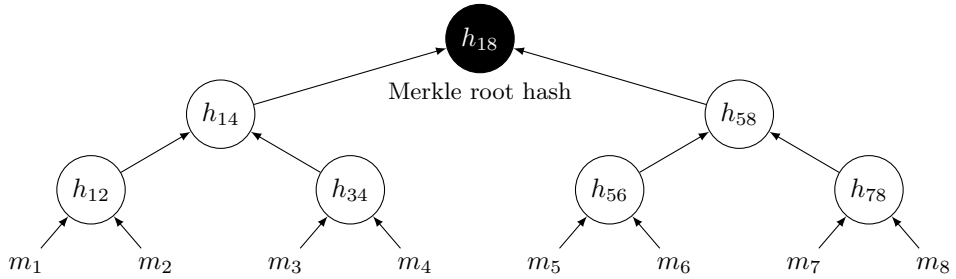


Figure 1.1: Graphical representation of a Merkle-tree. The nodes are calculated as $h_{i,i+1} = H(m_i \| m_{i+1})$ and $h_{i,j} = H(h_{i,m} \| h_{m+1,j})$, where H is a cryptographic hash function.

A Merkle tree, depicted in Figure 1.1, is a data structure that uses cryptographic hash functions to efficiently and securely verify the integrity of large sets of data. It organises data into a hierarchical tree-like structure, where each leaf node represents a single data block, and non-leaf nodes represent the hash of their respective child nodes. The root of the tree, called the *Merkle hash root*, is a single hash value that summaries the integrity of the entire dataset. Merkle trees are particularly useful in distributed systems, such as blockchain networks, where they enable efficient verification without requiring access to the entire dataset. For instance, to prove that a specific piece of data is included in the tree, one only needs the hashes along the path from the data block to the root, significantly reducing the amount of information needed for verification.

1.1.4 Key derivation function

A Key Derivation Function (KDF) is a cryptographic algorithm used to derive secure cryptographic keys from a base input, such as a password or master key. A typical usage of KDFs are to transform relatively low-entropy inputs into keys that are strong enough for use in encryption, authentication, or other cryptographic operations. But there are also interesting to derive strong key from high-entropy inputs, which is useful for instance to derive several keys from a single one.

1.2 Blockchain and consensus

First of all, let recall what is a blockchain: A blockchain consists on a chain of blocks, where each block is related with its predecessor by containing the hash of the previous block. The initial

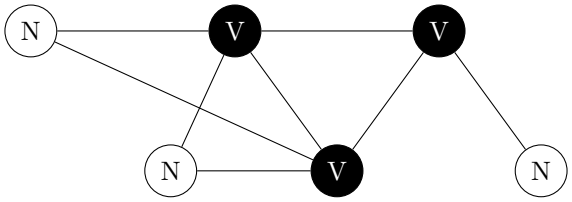
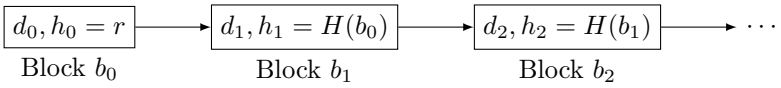


Figure 1.2: Example of network with three validators (in black) and three nodes (in white).

block, called the *genesis block*, can replace the hash of the previous block by some random r instead. Below is depicted a really simplified example of a blockchain.



A critical aspect of a blockchain is the infeasability for an attacker to tamper even a single bit without breaking the chain, thanks to the collision-resistance of the used (cryptographic) hash function. This tampering resistance is particularly appreciated in a context where integrity of some distributed data is crucial, as in the cryptocurrency setting and NFT. Indeed, the main interest of a blockchain is to mitigate the recurrent need of trust on a centralised authority but rather to distribute the trust on a large network of computers, with possibly malicious ones, to agree on a the same data *without* some centralised authority. These computers maintaining the blockchain are referred in this work as *nodes*. Note the "maintain" term, since when a block is added in the blockchain, each node updates its copy of the blockchain by extending the chain of blocks.

A block can contains any kind of information, generally referred as *transaction*. For example, a block in a blockchain used to manage cash contains transactions to transfer an amount of currency into another account. The consistency of the transaction with respect to the current state of the blockchain is unavoidable. This verification can be performed in many ways via a so-called *consensus*. In a setting where this validation is performed by a (possibly dynamic) set of well-defined validators, as in our, the validation is performed by *validator nodes*, a node having the charge to maintain the blockchain but also to verify the consistency of transactions of a given block. In Figure 1.2 we have depicted an example of a network.

1.3 Virtual blockchains

In our protocol, a single blockchain is used to anchor all the data whose the integrity has to be ensured, whatever the nature of these data and their origin. However, our protocol is designed to address possibly unrelated businesses thanks to our agnostic approach. To enhance the readability of the data being anchored in the blockchain, we have designed several new logical concepts, structuring the block of the blockchain.

1.3.1 Definition of a virtual blockchain

The first notion we have introduced is the concept of *virtual blockchain* (VB). By analogy with a virtual machine executed by a concrete machine, a virtual blockchain is maintained by the blockchain. From the blockchain perspective, a blockchain maintains virtual blockchains

by adding block, itself containing virtual block, or *micro-block*. In our protocol, seven *types* of virtual blockchains are used in the protocol, listed in Table 1.1.

Virtual blockchain name	**Instances number**	**Visibility**	**Description**
Organisation VB	n_{org}	Public	Section 1.3.2
Application VB	n_{app}	Public	Section 1.3.3
Oracle VB	n_{oracle}	Public	Section 1.3.4
Account VB	n_{acccount}	Public	Section 1.3.5
Validator nodes VB	$n_{\mathsf{validator}}$	Public	Section 1.3.6
Application-User VB	$n_{\mathsf{app}} \cdot n_{\mathsf{user}}$	Private	Section 1.3.7
Application-Ledger VB	$n_{\mathsf{appInstance}}$	Private	Section 1.3.8

Table 1.1: Listing of the virtual blockchains defined in our protocol.

As suggested in Table 1.1, a virtual blockchain can be instantiated many times if needed. For example, a single virtual blockchain is dedicated to manage a single validator node. Hence, if there are $n_{\mathsf{validator}}$ validator nodes, then there are as much as virtual blockchains. The same principle applies for all virtual blockchains.

Virtual blockchain visibility A virtual blockchain is said *public* if the data contained in the virtual blockchain can be accessed publicly. At the opposite, a virtual blockchain is said *private* if the data contained in the virtual blockchain can be accessed only by some parties. Note that whatever the visibility, the blocks of a virtual blockchain are anchored in the blockchain.

1.3.2 Organisation virtual blockchain

An *organisation*, in the context of this protocol, refers to a company. To participate in the protocol, a company must first register. This registration process results in the creation of a dedicated virtual blockchain. These virtual blockchains, which manage the operations of organisations, are referred to as *organisation virtual blockchains*.

Note that each organisation is managed by its own virtual blockchain to support the update of some fields of the organisation, such as the name or the country of the organisation. The same principle applies for all virtual blockchains.

1.3.3 Application virtual blockchain

Once registered, a company can define an *application*. It is important to note that an *application* in this context does not refer to a functional software application. Instead, it involves specifying the structure of the data that the application will transmit to the blockchain. We refer to this data structure specification as the *application definition*. These application definitions form a fundamental component of interactions within the protocol. All such definitions are stored and managed within a dedicated virtual blockchain, known as the *application virtual blockchain*.

1.3.4 Oracle virtual blockchain

An *oracle* in the context of the blockchain is a service or mechanism that provides external data to a blockchain or smart contract. Since blockchains are inherently isolated from the external world for security and integrity reasons, they cannot access external data directly. Oracles bridge this gap by acting as intermediaries that fetch, verify, and deliver real-world data to the blockchain.

Similar to an application definition, an organisation can define one ore more oracle by specifying its input and output. This oracle definition does not include additional details beyond this specification. The oracle definition is stored on a dedicated virtual blockchain known as the *oracle virtual blockchain*.

1.3.5 Account virtual blockchain

An *account* in the context of the blockchain corresponds to wallet associated with some amount of valuable tokens. As in the traditional banking system or in the context of Bitcoin, cash or tokens can be transferred from an account to another. The major difference between these two example in that in the centralised system, the bank is trusted whereas there is no trusted central authority, rather a large set of nodes maintaining the blockchain.

In our protocol, each account is managed by a dedicated virtual blockchain. In more details, the virtual blockchain starts by an initial declaration, including in particular the initial amount of the tokens, the issuer of these accounts and the public key of the user managing the account. Each time a transfer occurs, a new virtual block is added to the virtual blockchain.

1.3.6 Validator nodes virtual blockchain

Each validator in the network participating in the validation node has its own dedicated virtual blockchain, called the *validator node virtual blockchain*. Similarly to previous virtual blockchain, each validator has its own virtual blockchain to support the update of information related to the validator node.

1.3.7 Application-user virtual blockchain

For each user interacting with an application is associated a virtual blockchain. In this virtual blockchain is stored the data used by all the instances of the application definition. For instance, if a user obtains a proof of identity from an oracle, this proof can be used among all the instances of the application definition. For this reason, if there are n_{user} users and n_{app} application definitions, then we have at most $n_{\mathsf{user}} \cdot n_{\mathsf{app}}$ application-user virtual blockchains.

1.3.8 Application-ledger virtual blockchain

An application-ledger virtual blockchain represents an instance of an application definition. Each time an application definition is used by an user, a new virtual blockchain is created.

1.3.9 Description of the micro-block structure

All virtual blockchains described above are composed of micro-blocks. Whatever the virtual blockchain the micro-block belongs, it has the same structure, depicted in Figure 1.3.

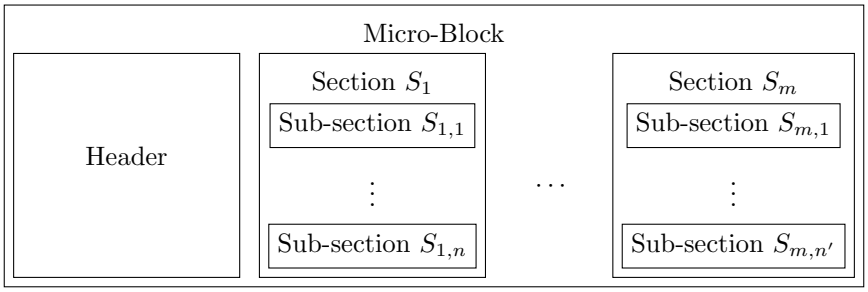


Figure 1.3: Graphical representation of micro-block structure.

Description of header The header of a micro-block always starts by the `CMTS` token indicating the used format. It is followed by the protocol version, the height (*i.e.,* the position of the micro-block in the virtual blockchain), the hash of the previous block, the block creation timestamp, the gas used to anchor the block and finally the gas price.

Definition of an internal and external scheme Before to detail the contain of a section and subsection, we have to introduce the notion of *scheme*. A scheme refers in our protocol as a data definition. For instance, an organisation definition is a scheme since it can be represented in a structured manner (for instance using JSON object). On one hand, an *internal scheme* refers to a scheme defined statically within the protocol. For example, the organisation structure do not depend on the application. On other hand, an *external application* consists on a scheme dynamically defined by the user, for instance when creating its application or oracle definition. The application definition, or more precisely its data structure, is problem-specific and is called external in this case.

Description of section A section contains the identifier of a scheme (the name of the scheme defining the structure of the subsections), which can be either internal or external. It also contains some information about the scheme. These two fields are public. Finally, it contains a list of sub-sections whose the structure is introduced below.

Description of sub-section A sub-section is the heart of the micro-block structure. We have three types of sub-section: *public*, *private* and *provable*, each having a dedicated structure, depicted in Figure 1.4. The usage of each sub-section type is directly related on sensibility of the data contained in the micro-block.

Public sub-section is the most simple sub-section, containing two fields being respectively `Type` and `Data`. The `Type` field indicates the type of the sub-section, among public, private and provable. In a public sub-section, `Type` is set as public. The `Data` field contains arbitrary data.

Private sub-section is intended to be used when the `Data` field contains sensitive data that should be only accessible by some parties. To achieve the confidentiality of the `Data` field, we rely on secret-key encryption. In addition to the fields `Type` and `Data`, a private sub-section contains three additional fields `AccessRules`, `KeyIndex` and `KeyType`, but also a more structured `Data` field. For clarity, we first focus on the last field `Data`. When an external observer sees a private sub-section in the blockchain, the `Data` field corresponds to a ciphertext, encrypting a structure containing two fields `Plaintext` and `Padding`. These fields are used during the sub-section encryption, in which one encrypts the message $\texttt{Plaintext} = m \| \texttt{Padding}$ where `Padding`

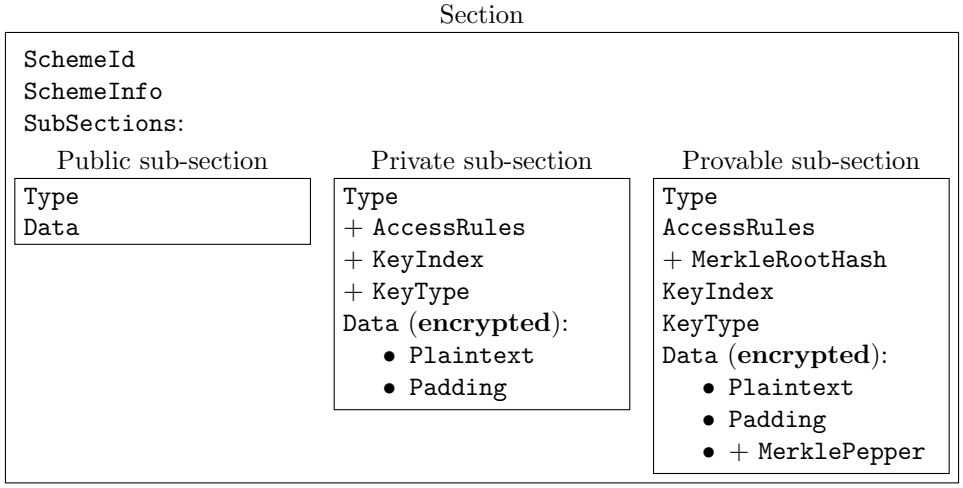


Figure 1.4: Graphical representation of the section and sub-section structures.

is used to hide the length of the message being encrypted[1]. While we will introduce the key schedule later, we mention that the secret key used to encrypt the private subsection, denoted *ssk*, is unique for each sub-section and derived from another secret. Note that the `KeyIndex` and `KeyType` fields specify the used type of key. Before to explain the meaning of the `AccessRules` field, we have to mention that the data provided by the user is encoded in a structured manner, for instance using a JSON object. The user may want to make its data fully public, fully private, or both. Indeed, the user may want to hide only a subset of properties contained in its JSON data object while making all other properties public. The `AccessRules` field defines the data properties affected by the type (public, private or provable) specified by the `Type` field in the sub-section. Note that a property can only be referenced in a single sub-section. We have exemplified the concept of access rules and visibility in Figure 1.5.

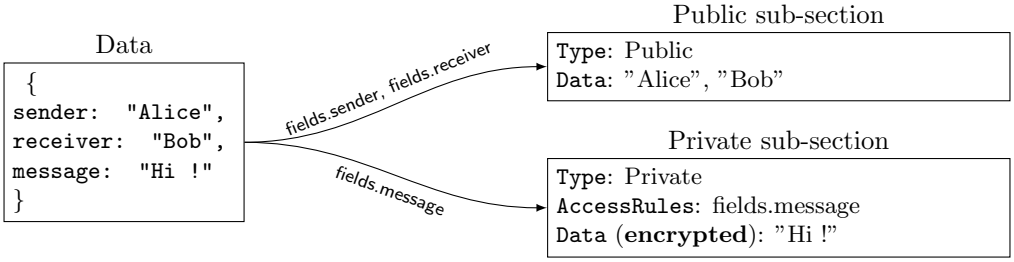


Figure 1.5: Splitting of data into sub-sections based on access rules.

Probable sub-section shares similarities with a private sub-section in the sense that both of them are used to preserve confidentiality of the `Data` field. There are two additional fields `MerkleRootHash` and `MerklePepper`. As its name suggests, the `MerkleRootHash` field contains the root hash of the Merkle-tree (see Section 1.1.3). This root hash is used by a proof verifier when verifying a proof of authenticity of (encrypted) data. The `MerklePepper` fields acts as a

[1]In contrast with other security properties being cryptographically secure, no security guarantee can be proved for message length hiding [TV11].

key to generate a proof, preventing anyone to generate a proof of authenticity. In other words, only parties having access to this particular field can generate proof of authenticity.

1.3.10 Channels

As we have seen, the notion of section and sub-sections refers as a technical structure to organise efficiently public, private and provable data in the blockchain. However, each of these structures, and more particularly the private and provable ones, are involved in a more high-level notion called *channels*. A channel can be understood as a sequence of related sub-sections within the same virtual blockchain. When a channel contains private sub-sections, we said that a the channel is *private*. In this case, secret encryption keys are used to encrypt all the data belonging to the channel. In this section, we explain how a channel is organised but also how is performed the key schedule (*i.e.,* the process to generate the encryption keys).

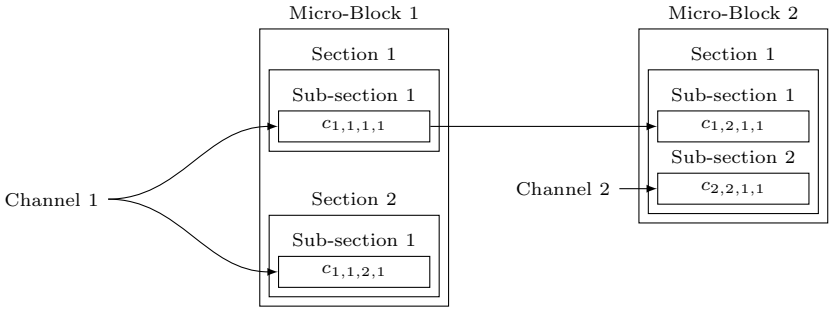


Figure 1.6: Graphical representation with two channels.

Organisation of a channel A channel corresponds to a related set of sub-sections. These sub-sections can belong to the same section but can also be distributed over multiple sections. In Figure 1.6, we have depicted two channels, whose the first channel contains three sub-sections distributed over three sections among two micro-blocks. Recall that every private sub-section contains encrypted data denoted $c_{i,hst}$ where $i \in \mathbb{N}$ is the channel index, $h \in \mathbb{N}$ is the height of the micro-block where the sub-section is located, $s \in \mathbb{N}$ is the index of the section (chosen incrementally) and $t \in \mathbb{N}$ is the index of the sub-section (chosen incrementally). From Figure 1.6, it should be clear that a section may contain sub-sections from distinct channels and that a channel may contain sub-sections distributed over sections and micro-blocks. Recall that a sub-section containing a ciphertext $c_{i,hst}$ has been encrypted using a dedicated sub-section encryption key $ssk_{i,hst}$. This encryption key has been generated via a process commonly called a *key schedule*.

Key schedule inputs Before to dive into the key schedule, we have to introduce some variables used as an input to the key schedule process. The first variable we need is the *seed* variable. Each virtual blockchain starts with a genesis bloc, and as any genesis block, the hash of the previous block do not exist, hence the hash of the previous block is replaced with a genesis identifier *genesisId* computed as $H(seed, ts)$ where *seed* is a randomly chosen element and *ts* is the timestamp where the virtual blockchain is created. The *seed* variable is uniformly chosen for each virtual blockchain and do not change over the lifetime of the virtual blockchain. The second variable we need is a private key, denoted *cck*, owned by the channel initiator. The private key, used as a crucial component in the key schedule, is kept secret by the channel initiator.

Key schedule We are now ready to describe the key schedule process. For clarity, we will refer in this section to the graphical example depicted in Figure 1.7, consisting on the key schedule for two channels distributed over two micro-blocks.

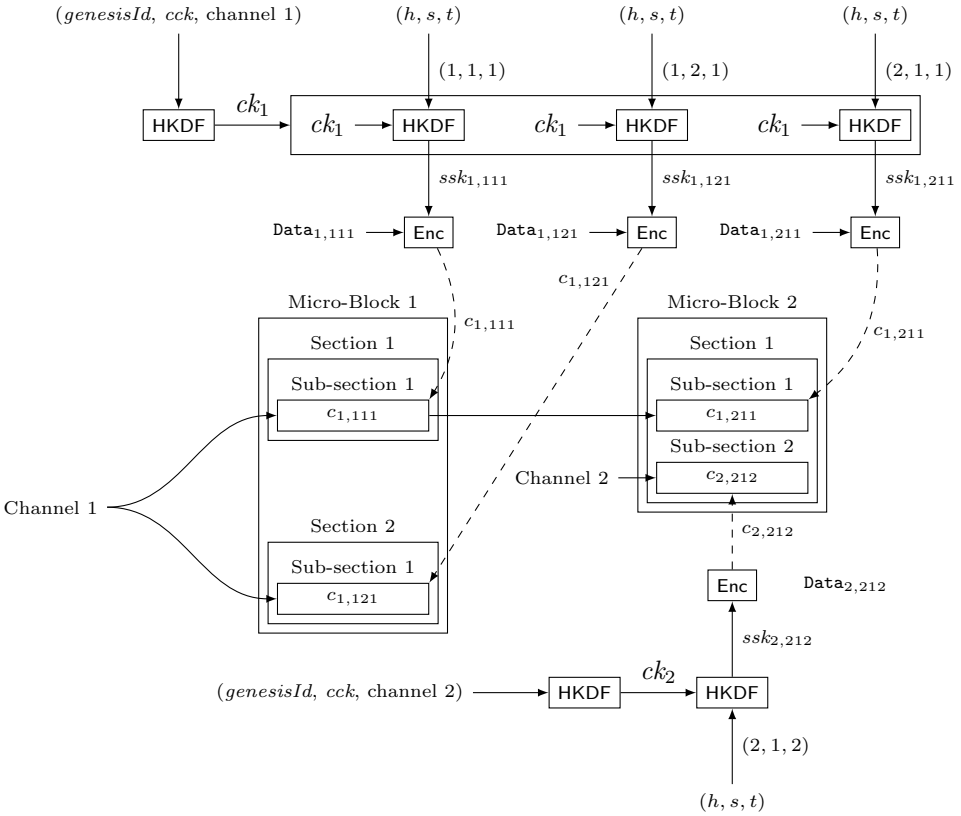


Figure 1.7: Graphical representation of a key schedule with two channels. The triplet (h, s, t) defines the index of the height of the micro-block, the index of the section and the index of the sub-section, respectively. The notation $x_{c,hst}$ should be read as x is associated to channel c at sub-section located at (h, s, t).

We start our description with the encryption key $ssk_{i,hst}$ used to encrypt the data within a sub-section. This key is derived from four inputs: (1) the height of the micro-block in which the sub-section is contained, (2) the index of the section inside the micro-block containing the sub-section, (3) the index of the sub-section inside the section and (4) a channel key, denoted $ck_{i,hst}$. More formally, the sub-section key derivation is computed as $ssk_{i,hst} \leftarrow \mathsf{HKDF}(ck_i, h\|s\|t)$ where HKDF is a (hash) key derivation function (see Section 1.1.4). Within a channel is associated a channel key ck which do not change during the lifetime of the channel. For the sake of example, observe that in Figure 1.7, the channel key ck_1 of the first channel is used to derive all the sub-section keys $ssk_{1,hst}$, used to encrypt the data $\mathtt{Data}_{1,hst}$. Therefore, the confidentiality of the encrypted data heavily holds under the confidentiality of ck. Similarly to sub-section keys, a channel key is derived, again using a key derivation function, from three inputs: (1) A public genesis identifier *genesisId* associated unique for each virtual blockchain, (2) a channel index i, and (3) a secret key *cck* kept private by the channel initiator. Formally, we have

$ck_i \leftarrow \mathsf{HKDF}(cck, genesisId \| i)$ with i the index of the channel.

Channels visibility and numbers Depending on the visibility of the contained sub-sections (*i.e.,* public, private or provable sub-sections), the channel becomes either public, private or provable. In other words, a public channel cannot contain private or provable sub-sections, a private channel cannot contain public or provable sub-sections and provable channel cannot contain public or private sub-sections. Note that the number of channels within the same virtual blockchain may depend on its visibility. By default, we have one channel for each type of channel visibility: one being public, one private and another one provable. While there are *always one public channel*, one may consider the case where there are *asymmetry of information*, handled by the channel access rules.

Channel access rules When a virtual blockchain is created, the channel initiator defines access rules for the data being anchored in the virtual blockchain. Suppose two private data d_1 and d_2 that can be accessed by users having respectively roles r_1 and r_2. Then, the channel initiator creates two distinct channels, one for each data. Hence, d_1 is encrypted in the first channel using a sub-section encryption key derived from ck_1, whereas d_2 is encrypted in the second channel using a sub-section key derived from ck_2. When a user want to access d_1 in the first channel, the channel initiator shares with the user, after completed some form of authentication, the channel key ck_1. For this reason, the number of private (but also provable) channels depends on the number of combination of roles to access the data anchored in the channel. Let denote by Data the set of variables (defined in the application definition). These access rules are formally defined as $\mathcal{L} = (\mathsf{Roles}, \mathsf{Chan}, \mathsf{Sub}, \mathsf{Perm})$ where Roles is a list of roles, Chan is a list of channels, Sub is a list of couples (r, C) allowing a role $r \in \mathsf{Roles}$ to access a set of channels $C \subseteq \mathsf{Chan}$. The last element Perm corresponds to the list of couple (c, D) where $c \in \mathsf{Chan}$ is a channel and $D \subseteq \mathsf{Data}$ is the variables visible in the channel c.

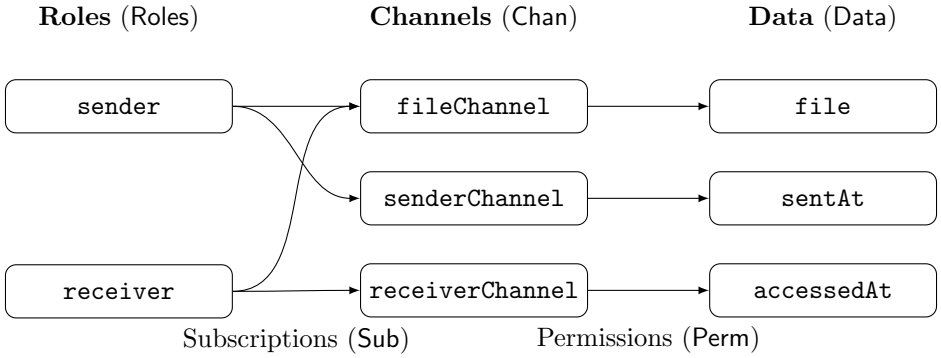


Figure 1.8: Example of access rules.

We have exemplified the access rules as follows: Suppose that a user, referred here as the sender, wants to sends a file to another user, called here the receiver. The sender wants to anchor on a virtual blockchain the fact that he has sent the file at a given date. Similarly, the receiver wants to anchor the fact that he has accessed the file at a given date. The sending and accessing dates are denoted respectively `sentAt` and `accessedAt`. The constraint is that the sender should not access the accessing date of the receiver. Conversely, the receiver cannot access the sending date, but the file, denoted `file`, is visible by both. To respect these constraints, one for each

variable, leading to three distinct channels. The first channel, containing the file and called *fileChannel*, is accessed by both the sender and the receiver. The second channel `senderChannel` can be accessed by the sender. The third channel `receiverChannel` can be accessed by the receiver. We have depicted these elements in Figure 1.8.

We now formally instantiate the access rules $\mathcal{L} = (\mathsf{Roles}, \mathsf{Chan}, \mathsf{Sub}, \mathsf{Perm})$ regarding on our example. The set of roles is composed of two roles, hence $\mathsf{Roles} = \{\texttt{sender}, \texttt{receiver}\}$. Since we have three distinct channels, we have $\mathsf{Chan} = \{\texttt{fileChannel}, \texttt{senderChannel}, \texttt{receiverChannel}\}$. The subscription set Sub exhibiting the accessible channels for each role, and the permissions set Perm exhibiting the data access permissions for each channel are defined as follows:

$$\mathsf{Sub} = \{\texttt{sender} : \{\texttt{fileChannel}, \texttt{senderChannel}\}, \texttt{receiver} : \{\texttt{fileChannel}, \texttt{receiverChannel}\}\}$$

$$\mathsf{Perm} = \{\texttt{fileChannel} : \{\texttt{file}\}, \texttt{senderChannel} : \{\texttt{sentAt}\}, \texttt{receiverChannel} : \{\texttt{accessedAt}\}\}$$

Channel initiator The holder of the private key *cck* used to generate a channel, hold by the channel initiator, is at the core of the confidentiality of the data being anchored on the virtual blockchain. It is now interesting to study *who* is the party holding the private key. To answer this question, we will now study the architecture of the protocol putting in practice all the notions we have introduced so far.

1.4 Parties description

As any cryptographic protocol, multiple parties are involved in the protocol. For the moment, we have cover all notions related to the blockchain including virtual blockchains and channels. All of these notions are orchestrated by several parties that we now describe.

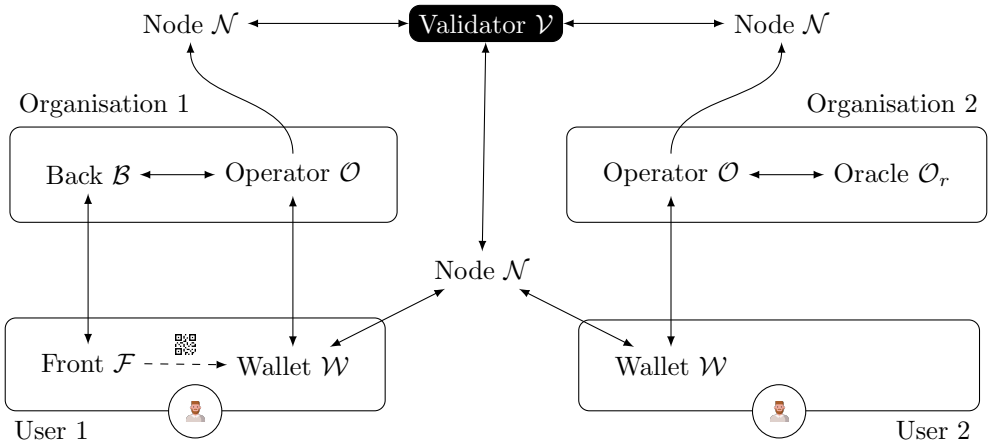


Figure 1.9: Graphical description of the parties. A directed edge between A and B means that A communicates with B. The dashed edge means a communication channel using QR code.

1.4.1 Description of the parties

The protocol contains *seven* distinct parties, depicted in Figure 1.9. Two of them are already known, namely nodes and validators, called respectively $\mathcal{N}$ and $\mathcal{V}$, in charge of maintaining the

blockchain (anchoring the virtual blockchains). Other parties are users of the blockchain. These remaining parties are grouped into two groups, namely organisational parties and user parties. In the first group are included three servers, called the *back*, the *operator* and the *oracle*, denoted respectively $\mathcal{B}$, $\mathcal{O}$ and $\mathcal{O}_r$. The group of users includes two parties, namely the *front* and the *wallet*, denoted $\mathcal{F}$ and $\mathcal{W}$ respectively.

Description of back and front We start our description by introducing the back and the front, serving the same purpose being to provide service to the user, taking the form of a web server. The back server provides web content to the user's browser referenced here as the front. It is for this reason that the back and the front are connected in Figure 1.9.

Description of the oracle The oracle in the protocol is aligned with the standard oracle definition in the blockchain terminology: An oracle is a party signing an information with its signature key. An oracle is particularly interesting when a user want to attest the ownership of an email address or more.

Description of the operator The operator consists on a server running on the organisation side. It is in charge of validate data before to be anchored in the virtual blockchain and ultimately in the sub-sections. The operator *confirms the acceptability of the data* by signing them, meaning that the operator holds the private key of a signature key pair. It is also in charge of *creating channels* within a virtual blockchain. Hence, the operator creates and holds a channel creation key *cck*, meaning that the operator is a channel initiator.

Description of the wallet The last type of parties in our protocol is the *wallet*. A wallet puts forward an interesting form of decentralisation in which an end-user manages and protects its own private key of a signature key pair. Essentially, the wallet is used by the user to approve data, later being anchored in the (virtual) blockchain. In the protocol, the wallet is limited to interact with three parties, depicted in Figure 1.9: The front by scanning QR code, and the operator and nodes by sending regular requests.

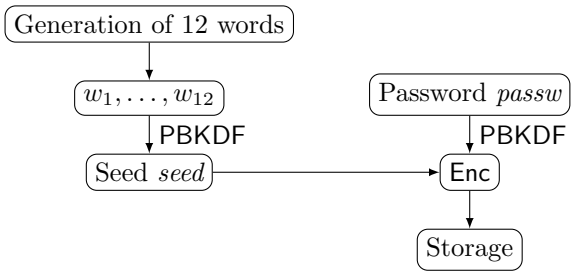


Figure 1.10: Description of the wallet setup.

1.4.2 Setup of the wallet

When a user creates a wallet, the user follows a procedure whose the general process is described in Figure 1.10. The wallet generates 12 words constituting a so-called *passphrase* from a wallet seed, denoted *seed* is derived. These derivations are done using a password-based key derivation (PBKDF). This seed is encrypted using a secret-key encryption scheme whose the secret key is

derived, again using a password-based key derivation function, whose the input is a password *passw* chosen by the user during the registration. From the seed is derived signature key pairs, depending on the need.

1.5 Protocols description

In this section, we introduce a general overview of the protocols developed on the introduced architecture.

1.5.1 Description of the public-key-based authentication protocol

The first protocol we introduce allows a user owning a given wallet, to authenticity to a service (managed by a back server) using its public key whose the private signature key is owned by the wallet.

The protocol starts with the user initiating its wallet. For clarity, we assume that the user owning the wallet has the wallet seed *seed*, from which every signature key pairs of the user are derived. The first step for the protocol is generation of a signature key pair (sk, pk) by the front $\mathcal{F}$. This public key pk is ultimately signed by the wallet, approving the front to perform action in the name of the wallet such as authenticating to a service. The public key pk is shared by the front with the wallet using the QR code. In this setting, the public key of the wallet should be known by the front $\mathcal{F}$ and the back $\mathcal{B}$.

1.5.2 Description of the oracle-based authentication protocol

The second protocol we introduce allows again the user to authenticate to a server managed by the back server. This time, the authentication is based on some information attested by an oracle, for instance using an email address.

The protocol shares similarities with the public-key-based authentication protocol. In this scenario, we assume that the wallet has some authentication material $\mathcal{I}$ as well as a signature $\sigma_{\mathcal{I}}$. The protocol establishes a secure connection between the front $\mathcal{F}$ and the wallet $\mathcal{B}$ owning the signature $\sigma_{\mathcal{I}}$. The secure connection is established via the operator $\mathcal{O}$ acting as the intermediate. When the front receives the signature from the wallet, the front verifies the authentication material $\mathcal{I}$ with respect to the signature $\sigma_{\mathcal{I}}$ and the public verification key of the oracle.

1.5.3 Description of the event approval protocol

The event protocol is initiated with the front $\mathcal{F}$ and the wallet $\mathcal{W}$ establishing a secure connection through the operator $\mathcal{O}$, via an ephemeral Diffie-Hellman, from which an encryption key is derived. This encryption key is used to encrypt data form the front to the wallet. Once received, the wallet approves the data by signing them using an ephemeral signature key derived from the seed and some other application-related information.

Chapter 2

Cryptographic background

2.1 Notations

We introduce the most common notations used in this work. More specific notations are introduced when required.

- By $a \leftarrow A()$, we denote the affectation of the output of the algorithm A to the variable a.
- By $a \leftarrow\!\!\$\ \mathcal{S}$, we denote the random sampling from the set $\mathcal{S}$, affected to the variable a.
- By λ, we denote the security parameter.
- By $[x_1, \ldots, x_n]$, we denote the list containing, in order, the elements $x_1, \ldots, x_n$.
- Given integers x and y such that $x \leq y$, by $[x, y]$, we denote the list $[x, x+1, \ldots, y]$. By $[x]$ we denote the list $[1, \ldots, x]$.
- An algorithm is written as $\mathsf{Alg}(\cdot)$, whereas a protocol between parties $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is written as $\mathsf{Proto}\langle\mathcal{A}_1(\cdot), \ldots, \mathcal{A}_n(\cdot)\rangle \rightarrow \mathcal{A}_1(\cdot), \ldots, \mathcal{A}_n(\cdot)$. Parties which do not receive an output from the protocol are sometimes omitted.
- By $[x_i]_{i\in[n]}$ we denote the list $[x_1, \ldots, x_n]$.
- By $X \stackrel{\text{P}}{=} Y$, we express the perfect indistinguishability between the two distributions X and Y. Similarly, the notation $X \stackrel{\text{S}}{\approx} Y$ expresses the statistical indistinguishability between the two distributions X and Y.

Protocol notation In this work, we use the Alice & Bob notation [BN05] whose we recall the major components:

- By A→B: m, we mean that A sends the message m to B. When B does some computation after received m, we list these computations right below.
- By A **knows** m, we mean that A knows the message m.
- By A **generates** m, we mean that A generates the message m, possibly from previously exchanged messages.

2.2 Requirements

As any cryptographic protocols, the security of our construction relies on existing cryptographic assumptions and primitives recalled below.

$\mathsf{Exp}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda)$	$\mathsf{Exp}^{\text{INT-CTXT}}_{\mathcal{A},\Pi}(\lambda)$
1 : $k \leftarrow \Pi.\mathsf{KGen}(1^\lambda)$	1 : $k \leftarrow \Pi.\mathsf{KGen}(1^\lambda)$
2 : $(m_0, h_0), (m_1, h_1) \leftarrow \mathcal{A}^{\mathsf{Enc}(k,\cdot,\cdot)}(\lambda)$	2 : $(c^*, h^*) \leftarrow \mathcal{A}^{\mathsf{Enc}(k,\cdot,\cdot)}(\lambda)$
3 : $b \leftarrow\$ \{0,1\}$	3 : $m^* \leftarrow \mathsf{Dec}(k, c^*)$
4 : $c \leftarrow \Pi.\mathsf{Enc}(k, m_b, h_b)$	4 : **return** $m^* \neq \perp \ \wedge \ (c^*, h^*) \notin \mathsf{Enc}(k,\cdot,\cdot)$
5 : $b' \leftarrow \mathcal{A}(c)$	
6 : **return** $b = b'$	

Figure 2.1: Experiments for IND-CPA-security and for INT-CTXT-security for an authenticated encryption with associated data.

Authenticated Encryption with Associated Data (AEAD) Encryption enables secure transmission of messages over untrusted communication channels by ensuring the confidentiality of the transmitted information. In the literature, two primary encryption paradigms are identified. The first is secret-key encryption, where both encryption and decryption rely on the same shared secret key. This approach is highly efficient, making it well-suited for encrypting large volumes of data. Let focus on secret-key encryption, and more particularly *authenticated encryption with associated data*, denoted AEAD. We recall the formal definition of an AEAD scheme from [Rog02]: An AEAD scheme is defined by the tuple $\Pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ over the key space $\mathfrak{K}$, the nonce space $\mathfrak{N}$, the header space $\mathfrak{H}$, the message space $\mathfrak{M}$ and the ciphertext space $\mathfrak{C}$ as follows:

- $\mathsf{KGen}(1^\lambda) \to k$: Given the security parameter λ, outputs a secret-key $k \in \mathfrak{K}$.
- $\mathsf{Enc}(k, n, m, h) \to c$: Given the secret-key $k \in \mathfrak{K}$, a nonce $n \in \mathfrak{N}$, a message $m \in \mathfrak{M}$ and some (public) header $h \in \mathfrak{H}$, outputs a ciphertext $c \in \mathfrak{C}$. For clarity, when the context make it clear, we voluntary omit the nonce by writing $\mathsf{Enc}(k, m, h) = \mathsf{Enc}(k, n, m, h)$ for a randomly generated nonce n. Note that n might be included in c without loss of security.
- $\mathsf{Dec}(k, c, h) \to m$ or $\perp$: Given the secret-key $k \in \mathfrak{K}$, a ciphertext $c \in \mathfrak{C}$ and an header $h \in \mathfrak{H}$, outputs either $m \in \mathfrak{M}$ if the couple (c, h) has been encrypted

The expectation from a security standpoint for an (authenticated) encryption scheme, intuitively, says that for any polynomial-time adversary, it preserves the confidentiality of the encrypted message. In other words, no one can distinguish between the encryption of a message, say m_0, from another message, say m_1, even if m_0 and m_1 are *chosen* by the adversary. More than choosing the challenge messages, we have to assume that the adversary has the ability to obtain the encryption of *any* message of his choice, before and after obtaining the challenge ciphertext. Under the prism of the secret-key encryption, an encryption oracle perfectly models the ability for an attacker to send a request to a running server accepting some message and returning its encryption. The formal definition of this experiment is called *Indistinguishability under Chosen-Plaintext Attack* (IND-CPA) that we have depicted in Figure 2.1. Note that compared to the initial nonce-based encryption definition of Rogaway [Rog04], we prevent the adversary to provide the used nonce since the usage of the same nonce may lead to critical flaws. For this reason, the used nonce is randomly chosen by the challenger during the experiment.

Definition 1 (Indistinguishability under Chosen-Plaintext Attack, IND-CPA**)** *Let* $\Pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *be an authenticated encryption scheme. Then,* Π *is said* IND-CPA*-secure if for every adversary* $\mathcal{A}$*, we have:*

$$\mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{A},\Pi} = 2 \cdot \left| Pr\left[\mathsf{Exp}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(1^\lambda) \to 1\right] - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda)$$

As shown in the work of Bellare [BN08], an authenticated encryption (with associated data) implies at the same time IND-CPA security explained above, and *Integrity of Ciphertext* security, denoted in short as INT-CTXT security. Intuitively, this property ensures that an adversary should not be able to produce a ciphertext c^* associated to some header h^* not produced by the encryption oracle. This property is formalised in Figure 2.1.

Definition 2 (Integrity of Ciphertext, INT-CTXT**)** *Let* $\Pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *be an authenticated encryption scheme. Then,* Π *is said* INT-CTXT*-secure if for every adversary* $\mathcal{A}$*, we have:*

$$\mathsf{Adv}^{\text{INT-CTXT}}_{\mathcal{A},\Pi} = Pr\left[\mathsf{Exp}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(1^\lambda) \to 1\right] \leq \mathsf{negl}(\lambda)$$

Signature Digital signature is a useful cryptographic primitive allowing a user to authenticate messages. In a signature, the signer of a message holds a secret signature key sk used to compute a signature of a given message m, denoted σ_m along this manuscript. The public verification key pk, as its name suggests, is public and used to authenticate the message via the signature σ_m. Since the verification of the signature relies on the public key, the verification procedure is public. Moreover, in case where the public key is certified to be associated to a specific user, say $\mathcal{U}$, then the signature σ_m proves that $\mathcal{U}$ authenticates m. Formally, a signature scheme is defined by the tuple $\mathsf{Sig} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$ defined over the key space $\mathfrak{K} \times \mathfrak{P}$, the message space $\mathfrak{M}$ and the signature space $\mathfrak{S}$:

- $\mathsf{KGen}(1^\lambda) \to (sk, \mathsf{vk})$: Given the security parameter 1^λ, outputs the secret signature key sk and the public verification key vk.
- $\mathsf{Sign}(sk, m) \to \sigma$: Given the secret signature key sk and a message m, outputs a signature σ.
- $\mathsf{Vf}(\mathsf{vk}, m, \sigma) \to b$: Given the public verification key vk, the message m and the signature σ, outputs 1 if the signature authenticates m.

The security of a digital signature scheme relies on the hardness for an adversary to produce a signature authenticating a message m', *not* signed by the user having the secret key sk. This security is formally known as *Existential-Unforgeability under Chosen-Message Attack* (EUF-CMA) security and is defined in the Figure2.2.

Definition 3 (Existential-Unforgeability under Chosen-Message Attack, EUF-CMA**)** *Let* Π *be a digital signature scheme. Then,* Π *is said* EUF-CMA*-secure if for every adversary* $\mathcal{A}$ *we have:*

$$\mathsf{Adv}^{\text{EUF-CMA}}_{\mathcal{A},\Pi} = Pr\left[\mathsf{Exp}^{\text{EUF-CMA}}_{\mathcal{A},\Pi}(\lambda) \to 1\right] \leq \mathsf{negl}(\lambda)$$

$\mathsf{Exp}^{\text{EUF-CMA}}_{\mathcal{A},\Pi}(\lambda)$	Oracle $\mathsf{OSign}(\mathcal{M}, sk, m)$
$(sk, pk) \leftarrow \Pi.\mathsf{KGen}(\lambda)$	$\sigma_m \leftarrow \Pi.\mathsf{Sign}(sk, m)$
$\mathcal{M} \leftarrow \emptyset$	**add** m **in** $\mathcal{M}$
$(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{OSign}(\mathcal{M}, sk, \cdot)}(\lambda)$	**return** σ_m
return $m^* \notin \mathcal{M} \wedge \Pi.\mathsf{Vf}(pk, m^*, \sigma^*) = \top$	

Figure 2.2: Security experiment for the EUF-CMA property.

Key Encapsulation Mechanism When two parties want to agree on a shared secret key, the most common and natural approach is to rely on the so-called Diffie-Hellman scheme. This scheme belongs to the line of research studying Key Encapsulation Mechanism (KEM). In a nutshell, a KEM scheme allows a party to generate a key pair containing a private key sk and a public key pk. Given the public key pk, one can derive two elements: A shared key k and a ciphertext c. We said that c "encapsulates" the shared key k. The party owning the key pair can efficiently agree on k by "decapsulates" the ciphertext c its private key sk, leading to k. Based on the work of Brendel *et al.* [BFG+19], we provide the formal definition of an unauthenticated KEM denoted $\Pi = (\mathsf{KGen}, \mathsf{Encaps}, \mathsf{Decaps})$ defined below:

- $\mathsf{KGen}(1^\lambda) \to (sk, pk)$: Given the unary representation of the security parameter λ, the key generation algorithm outputs a private key sk and a public key pk.
- $\mathsf{Encaps}(pk) \to (k, c)$: Given the public key pk, the key encapsulation
- $\mathsf{Decaps}(sk, c) \to k$: Given the private key sk and the ciphertext c, the decapsulation function outputs the secret key k.

The security notion associated to a KEM corresponds to the infeasability for any adversary to distinguish between a secret key k obtained via the execution of the KEM scheme and a secret key k randomly chosen from $\mathfrak{K}$. This security notion is formalised in the work of Brendel *et al.* [BFG+19] and recalled in Figure 2.3.

$\mathsf{Exp}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda)$
$(sk, pk) \leftarrow \Pi.\mathsf{KGen}(\lambda)$
$(k_0, c) \leftarrow \Pi.\mathsf{Encaps}(pk)$
$k_1 \leftarrow\!\!\$\ \mathfrak{K}$
$b \leftarrow\!\!\$\ \{0, 1\}$
$b^* \leftarrow \mathcal{A}(pk, c, k_b)$
return $b = b^*$

Figure 2.3: Security experiment for the IND-CPA property of KEM.

Definition 4 (Indistinguishability under Chosen-Plaintext Attack, IND-CPA**)** *Let* $\Pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *be a KEM scheme. Then,* Π *is said* IND-CPA*-secure if for every adversary* $\mathcal{A}$*, we have:*

$$\mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{A},\Pi} = 2 \cdot \left| Pr\left[\mathsf{Exp}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(1^\lambda) \to 1\right] - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda)$$

In the pre-quantum world, the most widely used KEM scheme is constructed based on the so-called *Diffie-Hellman key exchange* whose the security is based on the *Diffie-Hellman assumption*. The Diffie-Hellman assumption, or more commonly the Decisional Diffie-Hellman (DDH) assumption, refers on the hardness for any polynomial-time adversary to distinguish between the two distributions (g^a, g^b, g^{ab}) and (g^a, g^b, g^z) for $a, b, z \leftarrow\$ \mathbb{Z}_q$, where g is the generator of the prime-order subgroup $\mathbb{G}$. In this work, we rely on the secp256k1 curve for which the DDH assumption is assumed to be computationally hard against *classical* adversaries. The parameters of secp256k1 are recalled in Table 2.1.

p for $\mathbb{F}_p$	$2^{256} - 2^{32} - 977$
Equation	$y^2 = x^3 + 0x + 7$ $(a = 0, b = 7)$
Order q	$2^{256} - 432420386565659656852420866394968145599$
Cofactor h	1

Table 2.1: Parameters of the secp256k1 curve.

As suggested by Brendel *et al.* [BFG+19], an ephemeral Diffie-Hellman suits well within the KEM formalism.

- $\mathsf{KGen}(1^\lambda) \to (sk, pk)$: Generates a random $a \leftarrow\$ \mathbb{Z}_q$ and outputs $sk \leftarrow a$ and $pk \leftarrow g^a$.
- $\mathsf{Encaps}(pk) \to (k, c)$: Generates a random $b \leftarrow\$ \mathbb{Z}_q$ and outputs $k \leftarrow g^{ab}$ and $c \leftarrow g^b$.
- $\mathsf{Decaps}(sk, c) \to k$: Outputs $k \leftarrow g^{ba}$.

A major advantage of using the KEM notation instead of the DDH assumption is the ability to move from pre-quantum KEM to post-quantum ones without breaking the security of our scheme.

Cryptographic hash function In this work, we rely on cryptographic hash function. An hash function, modelled as $H : \{0,1\}^* \to \{0,1\}^\lambda$, takes as an input an arbitrary-sized string and outputs a λ-sized bitstring. The main interest of hash function is to obtain a short fingerprint, called a *hash*, of a possibly large data. This is particularly useful to check the integrity of a large data by comparing a short fingerprint. Indeed, the modification of a single bit should result into a completely different hash.

The critical security property of a cryptographic hash function is to provide *collision-resistance*, meaning that it should be infeasible to find two distinct messages m and m' such that $H(m) = H(m')$. By definition, such a couple of messages exists, but should be hard to obtain.

Another approach to model cryptographic hash function is to rely on the so-called *Random Oracle Model* (ROM). Introduced in [BR93], this model is an idealisation of the hash function, taking the form of an oracle, working as follows: This publicly-available oracle accepts any request containing a message m. If it is the first time that m has been seen, the oracle *randomly* chooses a value h from $\{0,1\}^\lambda$, and internally stores the couple (m, h) and returns h. In case where m has already been received during a previous request, then the oracle retrieves the couple (m, h) from its internal storage and returns h. A direct remark is that given $h \leftarrow \mathrm{ROM}(m)$, we have the guarantee that m cannot be learned, since h has been sampled at random from $\{0,1\}^\lambda$.

Password-Based Key Derivation Function Key Derivation Function (KDF) are particularly useful for obtaining pseudo-random bitstrings, used later as input keys for other cryptographic primitives, from a source key material. When, the source key material is considered to have a poor entropy such as passwords, it is interesting to rely on Password-Based Key Derivation Function (PBKDF). In a nutshell, this function is used to transpose a password with limited entropy to an l-sized pseudo-random bistrings.

We recall the formal definition from the work of Yao and Yin [YY05]. Formally, a PBKDF scheme is defined over the password space as $\mathfrak{K}$, the salt space as $\mathfrak{S}$, the number of iterations space $\mathbb{N}$ and the derived key length space $\mathbb{N}$. A PBKDF scheme is also defined via a family of hash functions $\{\mathfrak{H}_k\}_k$ from which we sample H. A PBKDF scheme is usually defined as $\mathsf{PBKDF}_H(c, p, s, l) \to k$ where:

- $c \in \mathbb{N}$ is the desired number of iterations.
- $p \in \mathfrak{P}$ is the password.
- $s \in \mathfrak{S}$ is the salt.
- $l \in \mathbb{N}$ is the length of the desired length of the derived key.
- $k \in \{0,1\}^\lambda$ is the derived key.

In our work, the pseudo-random function H is instantiated using SHA-256 and the number of iterations is fixed at 50000. The security of a password-based key derivation function is defined via a security experiment from [YY05, Definition 2], called *Strongly Secure (PB)KDF* and denoted SSPBKDF. Note that we adapted the security game using oracle instead of an iteration, and we have explicitly provided the output y. The provided security definition ensures that even given access to the pseudo-random function H, the salt s, the iteration number c, there is no polynomial-time adversary able to distinguish between $y \leftarrow F_H(p, s, c)$ and $y \leftarrow\!\$ \{0,1\}^l$.

$\mathsf{Exp}^{\text{SSPBKDF}}_{\mathcal{A},F,H,b}(\lambda)$	Oracle $\mathsf{OF}(p,(s,c),(s',c'))$
$p \leftarrow\!\$ \mathfrak{P}, s \leftarrow\!\$ \mathfrak{S}, c \leftarrow\!\$ \mathbb{N}$	If $(s,c) = (s',c')$ then abort
If $b = 0$ then $y \leftarrow F_H(p,s,c)$ else $y \leftarrow\!\$ \{0,1\}^c$	**return** $F_H(p,s',c')$
$b^* \leftarrow \mathcal{A}^{H(\cdot),\mathsf{OF}_H(p,(s,c),\cdot)}(y,s,c)$	

Figure 2.4: Strong Secure KDF definition for a password-based key derivation function F [YY05].

Definition 5 (Strong Secure KDF, SSPBKDF**)** *Let F be a password-based key derivation function. Then, F is said* SSPBKDF*-secure if for every adversary $\mathcal{A}$ we have:*

$$\mathsf{Adv}^{\text{SSPBKDF}}_{\mathcal{A},F,H} = \left| Pr\left[\mathsf{Exp}^{\text{SSPBKDF}}_{\mathcal{A},F,H,0}(\lambda) \to 1\right] - Pr\left[\mathsf{Exp}^{\text{SSPBKDF}}_{\mathcal{A},F,H,1}(\lambda) \to 1\right]\right| \leq \mathsf{negl}(\lambda)$$

Hash Key Derivation Function When dealing with a password, having a low entropy by definition, the probability to recover the password, compared to other cryptographic primitives, is not negligible. For instance, there are 2^{40} possible passwords of 10 hexa-decimals characters ($16^{10} = 2^{4*10} = 2^{40}$) which provides only 40 bits of security against brute-force attack, which is considered weak. To face this security, password-based key derivation function relies not only on the probability to guess the password but also the execution time required to compute the derived key y from a password p. Indeed, the advantage of an adversary is obtained from a probability ϵ to win the experiment, but also its execution time denoted t. Its advantage is computed as e/t. Going back to our example, a password-based key derivation function can virtually obtain a more bits of security by increasing the execution time require to obtain y. For instance, if 2^{10}

is required to derive y from p (knowing the salt and the iteration number), then the advantage to recover p by brute-force is $2^{-40}/2^{10} = 2^{-50}$, leading to 50 bits of security.

At the opposite, Hash Key Derivation Function (HKDF) considers inputs having high entropy. For this reason, an HKDF function does not have to artificially increase its execution time to obtain a better security and hence can enjoy a more direct and efficient approach.

Merkle-tree A Merkle-tree is intensively used in cryptography to prove the authenticity of a value v_i being part of a larger set of data $v_1, \ldots, v_n$, using a concise proof, say π, of size $O(\log(n))$. To obtain statistical privacy of the unrevealed data, Ben-Sasson *et al.* [BSCS16] relies on a privacy-friendly Merkle-tree, consisting on the addition of n randoms $r = r_1, \ldots, r_n$, where each random r_i is collapsed to the value v_i before to compute the leafs. Following the definition of Ben-Sasson *et al.* [BSCS16], we formally define a Merkle-tree by three algorithms $\mathrm{MT} = (\mathsf{GetRoot}, \mathsf{GetPath}, \mathsf{CheckPath})$ defined as follows:

- $\mathsf{GetRoot}(v, r) \rightarrow rt$: Given a l-sized list of values $v = v_1, \ldots, v_l$, outputs a root hash rt.
- $\mathsf{GetPath}(v, i, r) \rightarrow ap$: Given a l-sized list of values $v = v_1, \ldots, v_n$ and an index $i \leq l$, outputs an authentication path ap.
- $\mathsf{CheckPath}(rt, i, v_i, r_i, ap) \rightarrow b$: Given the root hash rt, an index i, the associated value v_i and the authentication path ap, outputs 1 if ap is a valid path for v_i with respect to the hash root rt.

The *soudness* of the Merkle-tree corresponds to the infeasability for any PPTadversary to produce an accepted authentication path for a different set of data v' and/or random r'.

$\mathsf{Exp}^{\mathrm{SOUND}}_{\mathcal{A},\Pi}(\lambda)$

1 : $(v_1, \ldots, v_n), (r_1, \ldots, r_n) \leftarrow \mathcal{A}(\lambda)$
2 : $rt \leftarrow \Pi.\mathsf{GetRoot}(v, r)$
3 : $\{(v'_j, r'_j, ap_j) | j \in [n]\} \leftarrow \mathcal{A}(rt)$
4 : **return** $\exists j \in [n], (v_j \| r_j) \neq (v'_j \| r'_j) \wedge \mathsf{CheckPath}(rt, j, v'_j, r'_j, ap_j) \rightarrow 1$

Figure 2.5: Security experiment for the soudness property of Merkle-tree.

Chapter 3

Protocol description

3.1 Notation

By $\mathcal{E}_{\mathsf{sig}}$ we denote the EUF-CMA-secure signature scheme and by $\mathcal{E}_{\mathsf{ae}}$ we denote the IND-CPA-secure and INT-CTXT-secure authenticated encryption scheme [BN08]. By $\mathcal{E}_{\mathsf{kem}}$, we denote the IND-CPA-secure key exchange mechanism (KEM).

3.2 Wallet creation

To install the wallet, the user is expected to provide a password denoted *passw*. From this password is derived a secret-key derived as $k_{\mathtt{passw}} \leftarrow \mathsf{PBKDF}(passw, 0, 256)$. This secret-key is used to encrypt the wallet. During the wallet installation, the wallet provides twelve words $passp = w_1, \ldots, w_{12}$ to the user. From these words, a seed $seed \leftarrow \mathsf{PBKDF}(passp)$ is derived. The seed (constituting the wallet) is *never* stored in clear, rather is encrypted as $c_{seed} \leftarrow \mathcal{E}_{\mathsf{ae}}.\mathsf{Enc}(k_{\mathtt{passw}}, seed)$.

Key	Known by	Used for	Lifetime
Knowledge			
passw	Wallet	Secure wallet storage	Forget when wallet closed.
passp	Wallet	Wallet recovery	Forget when wallet closed.
Private keys			
f	Front	Create secure channel	Forget when wallet handshake done.
w	Wallet	Create secure channel	Forget when wallet handshake done.
$k_{\mathtt{sess}}$	Front, Wallet	Secure channel	Single execution of the protocol.
$k_{\mathtt{pw}}$	Wallet	Secure wallet storage	Forget when wallet is closing.
$k_{\mathtt{oper}}$	Operator	Anchored data encryption	Forget when anchoring done.
$sk_{\mathcal{W}}^{\mathtt{lt}}$	Wallet	Long-term signature key	Long-term.
Public keys			
g^f	–	Create secure channel	Forget when wallet handshake done.
g^w	–	Create secure channel	Forget when wallet handshake done.

3.3 Protocols description

In Carmentis, the wallet communicates with the front using the operator as a proxy.

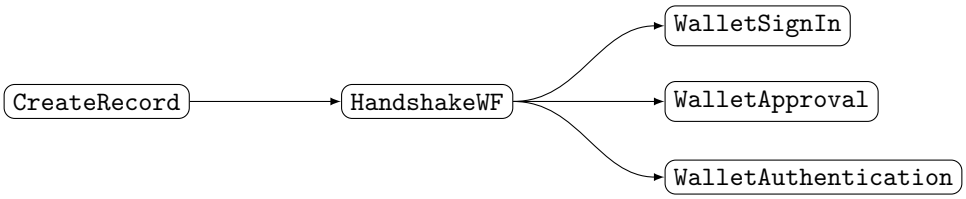


Figure 3.1: Description of the executed sub-protocols during an execution.

Description of `HandshakeWF` sub-protocol

In a common scenario, the end-user performs an action on the front, which has to be approved by the wallet. To approve the action, the wallet requires some information such as the data being anchored in the blockchain. However, the wallet can only receive a limited amount of data (via the QRCode displayed on the screen), vanishing any hope to transmit large data. To solve this issue, a front/wallet "handshake" is executed to create a secure connection between the front and the wallet, through the operator.

Initial knowledge of the parties Before the execution of the handshake, we assume that a party may have initial knowledge that we describe below:

- The wallet $\mathcal{W}$ initially has access to the seed *seed*, its long-term private signature key $sk^{\mathtt{lt}}_{\mathcal{W}}$ (which is derived from *seed*) and a read-write access to its state $\mathsf{state}_{\mathcal{W}}$.
- The front $\mathcal{F}$ initially knows some data d. During the handshake, these data are exchanged between all the parties (except the external observer and the node for which d must remains unknown). It also have access to `app` which contains the application identifier `appId` and the application version `appVer`. The front has also a read-write access to the state $\mathsf{state}_{\mathcal{F}}$.
- The back $\mathcal{B}$ initially knows some metadata `meta`, the application `app` as well as a read-write access to its state $\mathsf{state}_{\mathcal{B}}$.
- The operator $\mathcal{O}$ initially has a read-write access to its state $\mathsf{state}_{\mathcal{O}}$.

Initially, we assume that the front $\mathcal{F}$ knows the data d composed of the triplet $(d_{\mathtt{pub}}, d^{\mathtt{wfo}}_{\mathtt{priv}}, d^{\mathtt{wf}}_{\mathtt{priv}})$ where $d_{\mathtt{pub}}$ are the public data, $d^{\mathtt{wfo}}_{\mathtt{priv}}$ are the data, the identifier of the application `appId` as well as the application version `appVer`. We assume that $\mathcal{F}$ owns an unique device identifier `deviceId`, unique for a couple (website, device).

Description of the `CreateRecord` The record creation protocol is used to validate a set of data and prepare the handshake between the front and the wallet via the operator.

$\mathtt{CreateRecord}\langle\mathcal{F}(\mathrm{d}), \mathcal{B}(\mathtt{meta}), \mathcal{O}\rangle \rightarrow \mathcal{F}(\mathtt{recId}), \mathcal{B}(\mathtt{recId}, d), \mathcal{O}(\mathtt{recId}, d, \mathtt{meta})$

1. $\mathcal{F} \rightarrow \mathcal{B} : d$
   (a) Abort if d invalid.
2. $\mathcal{B} \rightarrow \mathcal{O} : \mathtt{PREPARE\_RECORD}, (d, \mathtt{meta})$ % Label added to enhance clarity

(a) Check that $\mathcal{B}$ is allowed to interact with $\mathcal{O}$. % TODO
(b) $\texttt{recId} \leftarrow\!\!\$\ \{0,1\}^{256}$
(c) store $(\texttt{recId}, d)$
(d) $\texttt{id} \leftarrow \texttt{appId-recId}$

3. $\mathcal{B} \leftarrow \mathcal{O} : (\texttt{id}, \texttt{recId})$
4. $\mathcal{F} \leftarrow \mathcal{B} : (\texttt{id}, \texttt{recId})$

Description of the `HandshakeWF` We now describe the messages exchanged during the handshake sub-protocol.

1. $\mathcal{F} \rightarrow \mathcal{O} : \texttt{MSG\_SDK\_CONNECTION}, \texttt{device}$
   (a) $r \leftarrow\!\!\$\ \{0,1\}^{256}$
   (b) $\texttt{now} \leftarrow$ Current date time
   (c) $\texttt{request} \leftarrow \texttt{device}\|\texttt{now}\|r$
   (d) $\texttt{requestId} \leftarrow H(\texttt{request})$
   (e) $\texttt{qrId} \leftarrow\!\!\$\ \{0,1\}^{256}$
   (f) store $(\texttt{requestId}, (\texttt{now}, \texttt{"PENDING"}, \texttt{request}))$
   (g) store $(\texttt{qrId}, (\texttt{requestId}, \texttt{now}))$
2. $\mathcal{F} \leftarrow \mathcal{O} : \texttt{qrId}$
   (a) $(sk_f^{\mathsf{kem}}, pk_f^{\mathsf{kem}}) \leftarrow \mathcal{E}_{\mathsf{kem}}.\mathsf{KGen}(1^\lambda)$
   (b) $\texttt{qrData} \leftarrow (\texttt{qrId}, \texttt{appId}, g^f)$

   % Wallet $\mathcal{W}$ starts the communication with the operator $\mathcal{O}$ based on the `qrData` obtained by scanning the QRCode.
3. $\mathcal{W} \leftarrow \mathcal{F} : \texttt{qrData}$ % (via QRCode scanning)
4. $\mathcal{W} \rightarrow \mathcal{O} : \texttt{MSG\_GET\_CONNECTION\_INFO}, \texttt{qrId}$
   (a) get $(\texttt{qrId}, (\texttt{requestId}, \texttt{ts}))$
   (b) get $(\texttt{requestId}, (\texttt{ts}, \texttt{status}, \texttt{device}\|\texttt{now}\|r))$
   (c) abort if $\texttt{status} \neq \texttt{"PENDING"}$
5. $\mathcal{W} \leftarrow \mathcal{O} : \texttt{device}\|\texttt{now}\|r$
   (a) $\texttt{id} \leftarrow H(\texttt{device}\|\texttt{now}\|r)$
   (b) Accept the connection.
   (c) $(k_{\mathsf{sess}}, c_{\mathsf{kem}}) \leftarrow \mathcal{E}_{\mathsf{kem}}.\mathsf{Encaps}(pk_f^{\mathsf{kem}})$
   (d) store $(\texttt{id}, k_{\mathsf{sess}})$
6. $\mathcal{W} \rightarrow \mathcal{O} : \texttt{MSG\_ACCEPT\_CONNECTION}, (\texttt{id}, c_{\mathsf{kem}})$
   (a) get $(\texttt{id}, (\texttt{ts}, \texttt{status}, \texttt{device}\|\texttt{now}\|r))$
   (b) $\texttt{status} \leftarrow \texttt{"CONNECTED"}$
7. $\mathcal{F} \leftarrow \mathcal{O} : \texttt{MSG\_WALLET\_CONNECTED}, (\texttt{id}, c_{\mathsf{kem}})$
   (a) $k_{\mathsf{sess}} \leftarrow \mathcal{E}_{\mathsf{kem}}.\mathsf{Decaps}(sk_f^{\mathsf{kem}}, c_{\mathsf{kem}})$
   (b) store $(\texttt{id}, k_{\mathsf{sess}})$

At the end of the wallet handshake sub-protocol, the wallet and the front have in common a shared secret session key k_{sess} obtained via the ephemeral Diffie-Hellman. This protocol is the cornerstone of other sub-protocols, establishing a secure connection between the front and the wallet by the intermediate of the operator.

3.3.1 Description of the WalletApproval protocol

Description of WalletApproval This protocol is a wrapper protocol describing the execution of the record creation, the handshake and the approvement of the record by the wallet. For clarity, let denote $d = (d_{\text{pub}}, d^{\text{wfo}}_{\text{priv}})$.
$\texttt{WalletApproval}\langle \mathcal{W}(seed), \mathcal{F}(d), \mathcal{B}(\texttt{meta}), \mathcal{O}, \mathcal{N}\rangle \rightarrow \mathcal{W}(d), \mathcal{F}, \mathcal{B}(d), \mathcal{O}(d)$

1. $\mathcal{W}$ **knows** *seed*, $\mathcal{F}$ **knows** d, $\mathcal{B}$ **knows** `meta` = `requestId`, `appId`, channels, actors, accessRules, approvalMsgId.
2. $\texttt{CreateRecord}\langle \mathcal{F}(d), \mathcal{B}(\texttt{meta}), \mathcal{O}\rangle \rightarrow \mathcal{F}(\texttt{recId}), \mathcal{B}(\texttt{recId}), \mathcal{O}(\texttt{recId}, d, \texttt{meta})$
3. $\texttt{HandshakeWF}\langle \mathcal{W}(seed), \mathcal{F}(\texttt{device}), \mathcal{B}, \mathcal{O}\rangle \rightarrow \mathcal{W}(k_{\text{sess}}), \mathcal{F}(k_{\text{sess}})$
4. $\mathcal{F}$ **generates** $\texttt{IV} \leftarrow\!\!\$\ \{0,1\}^{128}$, $\mathcal{F}$ **generates** $c \leftarrow \mathcal{E}_{\text{ae}}.\mathsf{Enc}(k_{\text{sess}}, \texttt{IV}, (d_{\text{pub}}, d^{\text{wfo}}_{\text{priv}}))$
5. $\mathcal{F} \rightarrow \mathcal{O}$: `MSG_REQUEST_DATA`, c
6. $\mathcal{W} \leftarrow \mathcal{O}$: `MSG_FORWARDED_REQUEST_DATA`, c
   (a) $d \leftarrow \mathcal{E}_{\text{ae}}.\mathsf{Dec}(k_{\text{sess}}, c)$
   % `genesisId` is undefined when it is the first interaction in the flow.
   % `genesisId` is added by the back.
   (b) $(sk_{\mathcal{W}}, \mathsf{vk}_{\mathcal{W}}) \leftarrow \mathcal{E}_{\text{sig}}.\mathsf{KGen}(seed, \texttt{nonce}, \texttt{appId}, \texttt{genesisId})$ % `genesisId` is obtained
7. $\mathcal{W} \rightarrow \mathcal{O}$: `MSG_ANSWER_SERVER`, (`"WALLET_HANDSHAKE"`, `recId`, $\mathsf{vk}_{\mathcal{W}}$)
   (a) get (`recId`, d)
   (b) get `genesisId`
   (c) $cck \leftarrow\!\!\$\ \{0,1\}^{256}$ % Generate channel key
   (d) Update virtual blockchain
8. $\mathcal{W} \leftarrow \mathcal{O}$: `MSG_SERVER_TO_WALLET`, (`"BLOCK_DATA"`, `recId`, `appId`, `genesisId`, d) % Confirm d
   (a) Show the data and accept (or reject).
9. $\mathcal{W} \rightarrow \mathcal{O}$: `MSG_ANSWER_SERVER`, (`"CONFIRM_RECORD"`, `recId`)
   (a) Anchor the data (anchorRecord(`recId`)) via $\mathcal{N}$ to obtain `res`
10. $\mathcal{W} \leftarrow \mathcal{O}$: `MSG_SERVER_TO_WALLET`, `res`
11. $\mathcal{W} \rightarrow \mathcal{O}$: `MSG_ANSWER_CLIENT`, `recId`
12. $\mathcal{F} \leftarrow \mathcal{O}$: `MSG_FORWARDED_ANSWER`, `recId`

3.3.2 Description of the WalletSignIn sub-protocol

In this protocol, the wallet scans a code QR from the front, in order to sign-in using its long-term signature key pair. The protocol is represented as follows:

$$\texttt{WalletSignIn}\langle \mathcal{W}(seed, \texttt{nonce}), \mathcal{F}(\texttt{device}), \mathcal{B}(\texttt{meta}), \mathcal{O}\rangle \rightarrow \mathcal{F}(sk^{\mathsf{sig}}_{\mathsf{ss}}, \sigma, pk^{\mathsf{sig}}_{w,\mathsf{ltk}})$$

We now describe the protocol formally:

1. $\mathcal{C}$ **generates** $(sk^{\mathsf{sig}}_{\mathsf{ss}}, pk^{\mathsf{sig}}_{\mathsf{ss}}) \leftarrow \mathsf{Sig}.\mathsf{KGen}(1^{\lambda})$
2. $\mathcal{W}$ **knows** *seed*, $\mathcal{F}$ **knows** $d = pk^{\mathsf{sig}}_{\mathsf{ss}}$, $\mathcal{B}$ **knows** `meta` = `requestType`, `appId`. $\mathcal{F}$ **knows** `device`.
3. $\texttt{CreateRecord}\langle \mathcal{F}(d), \mathcal{B}(\texttt{meta}), \mathcal{O}\rangle \rightarrow \mathcal{F}(\texttt{recId}), \mathcal{B}(\texttt{recId}, d), \mathcal{O}(d, \texttt{meta})$
4. $\texttt{HandshakeWF}\langle \mathcal{W}(seed), \mathcal{F}(\texttt{device}), \mathcal{B}, \mathcal{O}\rangle \rightarrow \mathcal{W}(k_{\text{sess}}), \mathcal{F}(k_{\text{sess}})$
5. $\mathcal{F}$ **generates** $\texttt{IV} \leftarrow\!\!\$\ \{0,1\}^{128}$, $\mathcal{F}$ **generates** $c \leftarrow \mathcal{E}_{\text{ae}}.\mathsf{Enc}(k_{\text{sess}}, \texttt{IV}, pk^{\mathsf{sig}}_{\mathsf{ss}})$
6. $\mathcal{F} \rightarrow \mathcal{O}$: `MSG_REQUEST_DATA`, c
7. $\mathcal{W} \leftarrow \mathcal{O}$: `MSG_FORWARDED_REQUEST_DATA`, c

(a) $(pk_{\mathsf{ss}}^{\mathsf{sig}}) \leftarrow \mathcal{E}_{\mathsf{ae}}.\mathsf{Dec}(k_{\mathtt{sess}}, c)$
(b) $(sk_{w,\mathsf{ltk}}^{\mathsf{sig}}, pk_{w,\mathsf{ltk}}^{\mathsf{sig}}) \leftarrow \mathcal{E}_{\mathsf{sig}}.\mathsf{KGen}(seed, \mathtt{nonce})$
(c) $\sigma \leftarrow \mathcal{E}_{\mathsf{sig}}.\mathsf{Sign}(sk_{w,\mathsf{ltk}}^{\mathsf{sig}}, pk_{\mathsf{ss}}^{\mathsf{sig}})$

8. $\mathcal{W} \rightarrow \mathcal{O}$: `MSG_ANSWER_CLIENT`, $(\sigma,\ pk_{w,\mathsf{ltk}}^{\mathsf{sig}})$
9. $\mathcal{F} \leftarrow \mathcal{O}$: `MSG_FORWARDED_ANSWER`, $(\sigma,\ pk_{w,\mathsf{ltk}}^{\mathsf{sig}})$

3.3.3 Description of the `WalletAuthentication` protocol

The authentication protocol is used when the user, through its wallet, wants to authenticate himself to an application using its email address. More precisely, the email address is verified by a so-called *oracle*[1] holding a private signature key used to the sign the email address. In the following, we denote the signature of the email by the oracle as $\sigma_{\mathtt{email}}$.

1. $\mathcal{W}$ **knows** *seed*, $\mathcal{W}$ **knows** $\sigma_{\mathtt{email}}$, $\mathcal{F}$ **knows** $d = \epsilon$, $\mathcal{B}$ **knows** `meta` = `requestType`, `appId`. $\mathcal{F}$ **knows** `device`.
2. $\mathtt{CreateRecord}\langle \mathcal{F}(d), \mathcal{B}(\mathtt{meta}), \mathcal{O}\rangle \rightarrow \mathcal{F}(\mathtt{recId}), \mathcal{B}(\mathtt{recId}, d), \mathcal{O}(d, \mathtt{meta})$
3. $\mathtt{HandshakeWF}\langle \mathcal{W}(seed), \mathcal{F}(\mathtt{device}), \mathcal{B}, \mathcal{O}\rangle \rightarrow \mathcal{W}(k_{\mathtt{sess}}), \mathcal{F}(k_{\mathtt{sess}})$
4. $\mathcal{F}$ **generates** $\mathtt{IV} \leftarrow\!\!\$\ \{0,1\}^{128}$, $\mathcal{F}$ **generates** $c \leftarrow \mathcal{E}_{\mathsf{ae}}.\mathsf{Enc}(k_{\mathtt{sess}}, \mathtt{IV}, d)$
5. $\mathcal{F} \rightarrow \mathcal{O}$: `MSG_REQUEST_DATA`, c
6. $\mathcal{W} \leftarrow \mathcal{O}$: `MSG_FORWARDED_REQUEST_DATA`, c
   (a) $(pk_{\mathsf{ss}}^{\mathsf{sig}}) \leftarrow \mathcal{E}_{\mathsf{ae}}.\mathsf{Dec}(k_{\mathtt{sess}}, c)$
   (b) $\epsilon \leftarrow \mathcal{E}_{\mathsf{sig}}.\mathsf{KGen}(seed, \mathtt{nonce})$
7. $\mathcal{W} \rightarrow \mathcal{O}$: `MSG_ANSWER_CLIENT`, $(\sigma_{\mathtt{email}})$
8. $\mathcal{F} \leftarrow \mathcal{O}$: `MSG_FORWARDED_ANSWER`, $(\sigma_{\mathtt{email}})$

3.4 Security arguments

In this section, we provide informal arguments on the security of the presented protocol. We stress that **these arguments cannot substitute security proofs**. We provide them to give intuitions justifying the obtained security of the protocol.

Confidentiality of data on chain The data within a private channel should be kept private from other parties outside of the channel. The secrecy of the data also holds at two conditions: The first condition is that the channel key *ck* and the sub-section key *ssk* are not revealed at any time by the user. The second condition is that the Merkle-trees should includes randoms to obtain a randomised hash. Under these two conditions, the confidentiality of the data in a private channel is believed to hold.

Anonymity of users The anonymity of the wallet owned by a user is not preserved at every step since the long-term public key of the wallet is used, for instance, for authentication. However, during the approval, the wallet do not rely on its public key but on a dedicated virtual blockchain-focused signature key pair, one key pair for one virtual blockchain. Within the same virtual blockchain, the used signature key pair is the same, meaning that only *weak* anonymity can be ensured, in the sense of a pseudonym (the freshly generated key pair) is used instead of the long-term public key. However, it is believed that two approvals performed by the same wallet

[1] This kind of oracles should not be confused with oracles given to the adversary in security experiments.

over two distinct virtual blockchains cannot be linked as coming from the same wallet. The reason is that the two key pairs are derived from a key derivation function whose the output is indistinguishable from randomly generated key pair. The anonymity holds under the fact that these two keys are not tied with an identity.

Proof unforgeability The core of the protocol is to provide a proof of integrity using Merkle-trees. The proof is required to be unforgeable, meaning that no one can create a proof of integrity for other data not anchored in the virtual blockchain. This unforgeability only holds under the unforgeability of the Merkle-tree, which intuitively holds under the collision-resistance property of the underlying cryptographic hash function.

Chapter 4

Security model & analysis

The introduced protocol involves a several parties having a different level of trust. In this section, we introduce the parties involved in the protocol and the targeted level of security.

- **Organisation**: The organisation corresponds to the entity running the *back* (of the web server) and the *operator*. Note that the front of the web server is served by the back server but is assumed to *not collude* with the organisation, which explains why the front is located at the *user* and not the organisation.
  - **Back**($\mathcal{B}$): The back server has a limited role in the security of the protocol. Indeed, in the protocol, the back server do not hold any key and is limited to provide the web page to the end-user.
  - **Operator**($\mathcal{O}$): The operator acts in the protocol as a counter-signer, attesting the (partial) correctness of the result sent by the user. By partial correctness, we mean that the verification is essentially structural (for instance, does every field has been provided?) rather than a complete verification.
- **User**: The user holds the data being anchored in the blockchain and initiates a transaction.
  - **Front**($\mathcal{F}$): The front corresponds to the web page sent by the back server. It is here that the user puts its data.

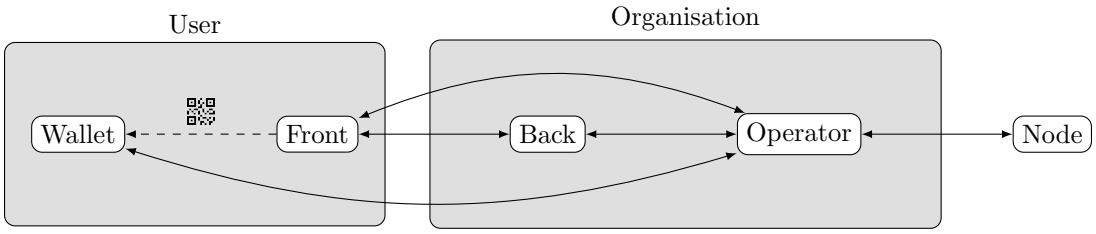


Figure 4.1: Graphical representation of the communication channel between the parties. Two parties linked with a full edge means that both parties are allowed to communicate between each other. Otherwise, no communication is assumed between these parties. The dashed edge between the wallet and the front means that the wallet can obtain information from the front (via QRCode) but the converse is not true.

  - **Wallet**($\mathcal{W}$): The wallet is owned by the user. The wallet holds a secret-seed from which all (signature) key pairs are derived. In our security model, we assume that the front can provide some information to the wallet via the usage of QRCode, as shown in Figure 4.1.

- **Node** ($\mathcal{N}$): The node plays a crucial role in the blockchain. It maintains the state of the ledger and, if configured appropriately, can participate to the validation process of the blocks to be added in the blockchain.

4.1 Security model

The security model is the intersection of a *protocol definition*, a *security property* and an *adversary*.

4.1.1 Protocol definition

- $\texttt{HandshakeWF}\langle \mathcal{W}, \mathcal{F}, \mathcal{B}, \mathcal{O} \rangle \rightarrow \mathcal{W}(k), \mathcal{F}(k)$ The inputs of the protocol for the parties are listed below:

  - The wallet $\mathcal{W}$ starts the protocol with its seed *seed*, its long term signature key $sk_{\mathcal{W}}^{\texttt{lt}}$ and its state $\mathsf{state}_{\mathcal{W}}$.

    The wallet ends the handshake protocol with a secret session key k shared with the front $\mathcal{F}$. It also outputs a state $\mathsf{state}_{\mathcal{W}}$ containing its (possibly updated) internal state.
  - The front $\mathcal{F}$ starts the handshake protocol with some data d, the application $\texttt{app} = (\texttt{appId}, \texttt{appVer})$ where $\texttt{appId}$ is the application identifier and $\texttt{appVer}$ is the application version. It is also inputted with a device parameter $\texttt{device} = (\texttt{deviceId}, \texttt{ip}, \texttt{ua})$ as well as its state $\mathsf{state}_{\mathcal{F}}$.

    At the end of the protocol, $\mathcal{F}$ outputs the session key $k_{\texttt{sess}}$ as well as its possibly updated state $\mathsf{state}_{\mathcal{F}}$.
  - The back $\mathcal{B}$ starts the handshake protocol with some metadata $\texttt{meta}$, the application $\texttt{app} = (\texttt{appId}, \texttt{appVer})$ and its state $\mathsf{state}_{\mathcal{B}}$.

    At the end of the protocol, $\mathcal{B}$ outputs the data d, the metadata $\texttt{meta}$ as well as its possibly updated state $\mathsf{state}_{\mathcal{B}}$.
  - The operator $\mathcal{O}$ starts the handshake protocol with its internal state $\mathsf{state}_{\mathcal{O}}$.

    At the end of the handshake, $\mathcal{O}$ outputs its possibly updated state $\mathsf{state}_{\mathcal{O}}$.

- $\texttt{Approve}\langle \mathcal{W}(\mathit{seed}, k), \mathcal{F}(k, d_{\texttt{pub}}, d_{\texttt{priv}}^{\texttt{wfo}}), \mathcal{B}, \mathcal{O}, \mathcal{N} \rangle$: The protocol ends with the following outputs from the parties:

  - The wallet $\mathcal{W}$ ends the protocol with the data being inputted by the user in the front, namely $d_{\texttt{pub}}$ and $d_{\texttt{priv}}^{\texttt{wfo}}$. It also terminates with a proof key k_π used later to prove the authenticity of some data.
  - The operator $\mathcal{O}$ ends the protocol with the public data $d_{\texttt{pub}}$ as well as the private data $d_{\texttt{priv}}^{\texttt{wfo}}$. It also obtains a commitment cm for the anchored data d (not only on the public data) but also an agreement $\sigma_{\mathcal{W}}$ from the wallet.
  - The node $\mathcal{N}$ acting as a bulletin board, obtains all data being anchored in the blockchain, which includes the commitment cm, the public data $d_{\texttt{pub}}$.

- $\mathsf{GenProof}(k_\pi, \mathcal{D}) \rightarrow \pi$: Given the proof key k_π and a subset $\mathcal{D}$ of the data d (containing the public and private data), this algorithm outputs a proof π.
- $\mathsf{CheckProof}(\mathcal{D}, cm, \pi) \rightarrow b$: Given the subset $\mathcal{D}$ of data d (containing the public and private data), the commitment cm and the proof π, it outputs 1 if the proof is valid, 0 otherwise.

4.1.2 Considered adversaries

Our security definition can be observed under the prism of different adversaries, each one modelling a different attack and corruption scenario. For a better understanding of the security, we provide three adversaries having an incremental capabilities. We describe these adversaries, whose the capabilities are sum-up in Table 4.1.

	Abilities				**Proof Method**	
Name	MS	MI	MB	CR	CM	FV
External observer	✓				✓	✓
External attacker	✓	✓	✓		✓	✓
System intruder	✓	✓	✓	✓		✓

Table 4.1: Description of the considered adversaries. MS, for "Message Sniffing", allows the adversary to read exchanged messages. MI, for "Message Injection", allows the adversary to inject a message at any moment in the protocol and to any party. MB, for "Message Blocking", allows the adversary to block a message transiting from a party to another. CR, for "Corruption", allows the adversary to corrupt an honest party (before the setup). CM, for "Computational Model", corresponds to a proof in the computational model. FV, for "Formal Verification", corresponds to a proof using automated verification. *Note: The proofs are currently in progress.*

External observer The first adversary we consider is called *external observer*. As its name suggests, this adversary is passive in the sense that he is limited to observe the communication between the parties and nothing else. Hence, no interaction is allowed with any of the existing parties and is only able to read data being published. While this adversary is not useful and hence do not model well a real attacker, the protocol *should* be considered secure against this passive adversary, otherwise, no security can be proven against stronger adversaries.

External attacker The second adversary we consider is called *external attacker*. This time, the adversary is allowed to interact with any party accepting outside messages. It is also able to block any communication between two parties, but also modify a transiting message when it is possible.

System intruder The third and last adversary, called *system intruder*, obtains all the capabilities of the two previous adversaries, but is also able to corrupt a party inside the protocol.

Communication model During the protocol execution, we assume that a message exchanged between two parties A and B is done via *secure* and *authenticated* channel. In particular, we assume that replay-attacks are prevented using these channels. Note that only one-sided authenticated channel is sufficient for our need. In contrast, we allow an external adversary to initiate a message with any party of its choice. We elaborate more on this aspect below.

4.2 Security properties

In this section, we introduce the security properties guaranteed by the Carmentis protocol. For clarity, we first introduce them informally.

Proof unforgeability The main goal of Carmentis is to provide a mechanism to prove the authenticity of the data being anchored in the blockchain. This can can expressed as follows: Suppose two users u_1 and u_2 and some l-sized data $d = d_1, \ldots, d_l$ owned by u_1. The user u_1 wants to prove the authenticity of a subset of d, denoted $\mathcal{D} = \{d_i | i \in \mathcal{I}\}$, to u_2. To help u_1 to prove the authenticity of $\mathcal{D}$, we allow him to compute some commitment cm on which all users will agree on, for instance using the blockchain. A trivial solution for this would be to anchor all the data in the blockchain, which might be not suitable for large data. Rather, we restrict the size of cm to be sub-linear (even better, independent) on the size of the data d.

When the user u_1 wants to prove the authenticity of $\mathcal{D}$ with respect to the commitment cm, u_1 will compute a proof π whose the size is restricted to be sub-linear in d (otherwise, it suffices to reveal d in its entirety). Denote the proof validation algorithm as $\mathsf{CheckProof}(cm, \mathcal{D}, \pi) \to b$. The *unforgeability* property ensures that it should be infeasible to compute a proof π' with respect to some data $\mathcal{D}'$ for some indexes $\mathcal{I}$ (different of $\mathcal{D}$ for at least one index $i \in \mathcal{I}$, or in other words, $d_i \neq d'_i$) such that $\mathsf{CheckProof}(cm, \mathcal{D}', \pi') \to 1$.

Data confidentiality The confidentiality of the data is expected to reach one of the three following levels:

- The first level called *public* is used when the data can be made public. For this level, no confidentiality is required.
- The second level called *system private* is used when the data cannot be made public. At this level, any exterior adversary should not be able to learn the data. At this level, every entity acting in the system managed either by the user or the organisation are allowed to observe the data.
- The third and last level called *user private* is used when the data cannot be made public and should not be visible by the organisation. In other words, only the user (which includes the wallet and the front) should be able to learn the data.

Wallet anonymity A wallet in the cryptocurrency systems is mainly designed to manage a secret signature key used for approving an action like a sign-in or a transaction, by creating a so-called signature. The *public* signature key, known by everyone as suggested in its name, is used to verify a signature computed over the signed object. An interesting property for an end-user having a wallet is to consider a notion of *anonymity*. More formally, it should be infeasible to distinguish between two users u_1 and u_2 (each one having its own wallet) interacting publicly with its wallet.

Chapter 5

Presentation of use-cases

In this chapter, we introduce two applications of our protocol called respectively *Sign* and *Access*.

5.1 Sign application

In this application, two users A and B want to sign a file f using their wallet. Before to sign the file, each user must provide a proof of identity provided by an oracle. The signature of the file is performed only after the validation of the identity with respect to the proof of identity provided by the oracle. For clarity, we divide our explanation into four steps: First, we present the initial setup including the parties and the existing virtual blockchains. Second, we introduce the procedure in which the back server anchors the file f that should be signed by A and B. One can think about f as a contract which is first written and approved by a lawyer before to be signed by the users. Third, we introduce the procedure in which a user obtains a proof of identity. Fourth, we explain the procedure in which the file is signed.

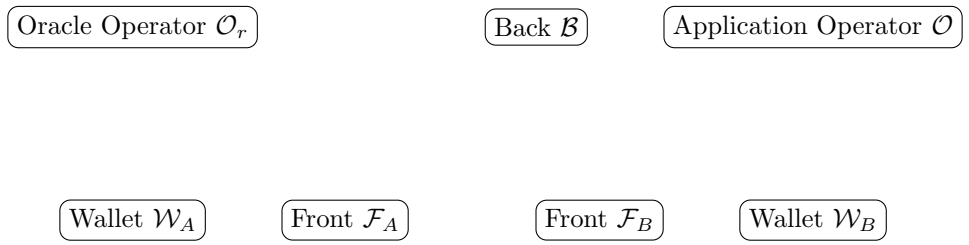


Figure 5.1: Graphical representation of parties in the sign use-case.

5.1.1 Initial setup

We start the initial setup description by introducing the parties. As a central piece of the protocol is located the back server $\mathcal{B}$ providing the file f to be signed. Together with the back server $\mathcal{B}$ is located the application operator $\mathcal{O}$. Note that for clarity we mention $\mathcal{O}$ as an application operator instead of simply operator to avoid confusion. Indeed, in the protocol, the oracle is composed of two parts: The oracle operator receiving request, acting as a proxy for the second part of the oracle being the oracle backend. For simplicity, we will only consider the oracle

operator denoted $\mathcal{O}_r$. The protocol also includes two wallets $\mathcal{W}_A$ and $\mathcal{W}_B$ and two fronts $\mathcal{F}_A$ and $\mathcal{F}_B$, one for each user A and B.

Before to explain the active steps of the protocol, it remains to explain the assumed initial structure of the virtual blockchains. First, we assume that each organisation (one to manage the oracle and another one to manage the back and application operator) have setup an organisation virtual blockchain. In addition, the organisation managing the back and application operator is assumed to have declare an appliaction definition in a new application virtual blockchain (see Section 1.3.3). Similarly, the organisation managing the oracle is required to have an oracle virtual blockchain (see Section 1.3.4) declaring the oracle definition.

5.1.2 File anchoring

The file anchoring procedure starts with back server assumed to hold (or receive) the file f. The back sends an anchoring request to the application operator $\mathcal{O}$, which then creates a new application-ledger virtual blockchain (see Section 1.3.8). This new (public) application-ledge virtual blockchain is initiated with a single micro-block containing the file f and the two identities A and B, corresponding to the set of allowed signers.

5.1.3 Authentication

In the second step, the user authenticates to the back server with a proof of identity provided by the oracle. This proof of identity is anchored in the blockchain.

The authentication process starts with a request access from $\mathcal{F}_A$ to the back for the file f, providing its public key pk_A. The back $\mathcal{B}$ asks the application operator $\mathcal{O}$ to allow A (identified by its public key pk_A) to the file f. The request consisting on pk_A is transferred from $\mathcal{O}$ to the oracle $\mathcal{O}_r$ which then starts an authentication process with A. Once the authentication process is completed, the oracle $\mathcal{O}_r$ reponds with three elements: An identity $\mathcal{I}_A$, a signature σ_A, and a merkle root hash rt_A. Once these three elements are sent by the oracle to the application operator $\mathcal{O}$, it anchors in a new application-user virtual blockchain (see Section 1.3.7) whose the first micro-block contains these three elements. Note that the application-user virtual blockchain has a larger scope that application-ledger virtual blockchain, since it is independent of the instances of the application. To use this proof of identity anchored in the application-user virtual blockchain in the current instance of the application, a new *reference block* is added to the application-ledger virtual blockchain, referencing the (public) micro-block containing the proof of identity in the application-user virtual blockchain.

5.1.4 File signature

Once the user has been authenticated, the user A asks via the front $\mathcal{F}_A$ to the back $\mathcal{B}$ to sign the file f. A request is sent to the wallet $\mathcal{W}_A$ to sign the file f, returning the signature $\sigma_{A,f}$ to the operator $\mathcal{O}$. This signature is then stored in the application-ledger virtual blockchain in a dedicated micro-block.

5.2 Access application

The access application allows a user to prove its majority to access an adult service, without revealing any information about the identity of the user accessing the service.

Overall protocol description We describe the protocol in which four parties are invovled: The proof of majority oracle $\mathcal{O}_1$, the user $\mathcal{U}$, the adult service $\mathcal{B}$ and a second oracle $\mathcal{O}_2$.

1. The first interaction is initiated by the user U to the adult service $\mathcal{B}$, sending an "hello" message.
2. The adult service responds with a random challenge s to the user.
3. The user asks the first oracle $\mathcal{O}_1$ in order to obtain a proof of identity.
4. The first oracle $\mathcal{O}_1$ responds with a proof of identity composed of two parts: A signature σ_U of a hash root rt_U, corresponding to the couple (σ_U, rt_U).
5. The user sends the couple (σ_U, rt_U) and the challenge s is sent to the second oracle $\mathcal{O}_2$.
6. The second oracle $\mathcal{O}_2$ responds with a signature σ_s of s.
7. The signature σ_s is sent back to the adult service which verifies the signature σ_s using the public key of the second oracle $\mathcal{O}_2$.

Bibliography

[BFG+19] Jacqueline Brendel, Marc Fischlin, Felix Günther, Christian Janson, and Douglas Stebila. Challenges in proving post-quantum key exchanges based on key encapsulation mechanisms. *IACR Cryptol. ePrint Arch.*, 2019:1356, 2019.

[BN05] Sébastien Briais and Uwe Nestmann. *A Formal Semantics for Protocol Narrations*, page 163–181. Springer Berlin Heidelberg, 2005.

[BN08] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4):469–491, July 2008.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, page 62–73, New York, NY, USA, 1993. Association for Computing Machinery.

[BSCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Proceedings, Part II, of the 14th International Conference on Theory of Cryptography - Volume 9986*, page 31–60, Berlin, Heidelberg, 2016. Springer-Verlag.

[Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, page 98–107, New York, NY, USA, 2002. Association for Computing Machinery.

[Rog04] Phillip Rogaway. *Nonce-Based Symmetric Encryption*, page 348–358. Springer Berlin Heidelberg, 2004.

[TV11] Cihangir Tezcan and Serge Vaudenay. *On Hiding a Plaintext Length by Preencryption*, page 345–358. Springer Berlin Heidelberg, 2011.

[YY05] Frances F. Yao and Yiqun Lisa Yin. *Design and Analysis of Password-Based Key Derivation Functions*, page 245–261. Springer Berlin Heidelberg, 2005.