HERMES
A framework for cryptographically assured access control and data security

Eugene Pilyankevich
eugene@cossacklabs.com

Ignat Korchagin
ignat@cossacklabs.com

Andrey Mnatsakanov
andrey@cossacklabs.com

ABSTRACT

This paper presents Hermes – a practical data security scheme with a reference implementation, which enables distributed sharing and collaboration, enforcing access control with the help of cryptographic methods (public key cryptography and traditional symmetric cryptography).

Keywords

Persistent storage operations (CRUD), cryptographic enforcements of access control, cryptographic access control (CAC), distributed information system, encryption scheme.

1. Introduction

Modern information systems evolve into increasingly complex structures. This paves the way for new potential security bugs, vulnerabilities, and data leaks [1]. The essential security practices that underpin modern information systems' development are the efficient compartmentalisation of components and restricting the exposure of sensitive information. However, these measures are insufficient when information systems are required to interact with third-party services and use communication channels, which cannot be fully trusted [2, 3].

Under such circumstances, only strong cryptographic solutions can provide sufficient proven security guarantees and ensure unconditional access control enforcement in distributed information systems. It is in this context that we developed Hermes – a practical cryptography-based access control and data security scheme with a reference implementation [4, 5].

Hermes enables collaboration and distributed data sharing through enforcing access control with the help of cryptographic methods (both public key cryptography and traditional symmetric cryptography). In case of an attack, when one or more of the components within the Hermes framework are compromised, Hermes will preserve the maximum possible number of security guarantees for the protected data (a denial-of-service is considered to be the worst-case possible outcome within a Hermes-powered infrastructure).

Hermes allows distributing only the necessary amount of data between the system components for their correct operation and, consequently, limits the possible damage that may be done if a component is compromised.

As the network design of Hermes can be mapped to typical client-server architecture, we can define the following major advantages of Hermes:

1. An absence of a central point of failure (sensitive data being compromised);
2. No access to both cryptographic keys and sensitive data in plain text for the server side;
3. End-to-end authenticated encryption between all the components (both server side and client side).

1.1 Problem definition and existing research

Access control and data protection are fundamental security services in the modern computing systems. In essence, an access control system filters the attempts of client(s) to execute any of the basic data access operations. This is done by enforcing a set of access rules (permissions) on protected resources, only allowing interactions authorised by the policy configured by the resource owner(s).

Implementations of access control in software are vulnerable to any compromisation of the machine that hosts it. Moreover, such enforcement mechanisms do not work when protected resources are stored by an untrusted or semi-trusted third party, which is an increasingly common practice [6].

Cryptographic enforcements of access control have been researched for over 30 years and are now a mature research topic in itself [6]. When symmetric cryptographic primitives are used, each protected resource is encrypted and only the authorised clients should have the access to the encryption key. This control mechanism is implemented via wrapping the encryption key in an asymmetric authorisation key, unique for each client. We use the term access control keys (ACKs) to denote these authorization keys in the Hermes' scope. But, unlike the majority of similar data access control systems, ACKs in Hermes are used (directly or indirectly) to cryptographically protect the data and not just to support the entity authentication in the system.

There are other modern cryptographic methodologies which provide different sets of security guarantees and serve different design goals. Prior to our research, we've studied the available practical implementations of attribute-based encryption [7], searchable encryption [8], private information retrieval [9], fuzzy identity-based encryption [10], homomorphic encryption [11] but have not found proposed solutions that would either match the real-world performance requirements or provide wide enough use-case flexibility.

1.2 Design goals and choices

Hermes uses a typical modern data-processing model, where "pieces" of data are stored in one or more (possibly remote) logical data stores (databases, files, key-value stores, etc.) and where an access control policy (ACP) for the stored data is defined.

Hermes has two main design goals:

1. Minimisation of damage from compromisation of separate components of the system;
2. Cryptographic enforcement of the implementation of the determined ACP.

The first goal is achieved through limiting the exposure of data between the components of Hermes and also through separating them from each other. The second goal is achieved through reliance on cryptographic solutions rather than on institutional and operational guidelines and correct software implementation of an ACP. Each access right granted to one or another client within an ACP is expressed through an appropriate ACK that can only be obtained by this authorised client.

This concept of cryptographic enforcement of an ACP over the data within the Hermes' scheme allows:

1. Binding an ACP implementation (collection of ACKs) to the actual data, wherever it is stored or transmitted and making it available to each component of Hermes;
2. Storing an ACP implementation in an arbitrary environment (centralised or distributed) where it is available to legitimate parties upon request.

Hermes explicitly defines two basic functions in relation to persistent storage – READ and UPDATE. The remaining functions of the four persistent storage operations (CRUD) [12] – CREATE and DELETE – are derived from READ and UPDATE and certain implementation constraints (described in more detail in [5]). It is also possible to *grant* (in other words – to *delegate*) and to *revoke* permissions to these functions.

In summary, the ability to READ is implemented as an ability to decrypt data, while the ability to UPDATE is implemented as an ability to decrypt data and calculate the message authentication code for this data. The possession of a READ ACK also implies an ability to *grant/revoke* it. The same holds true for UPDATE. In general, possession of certain permission implies an ability to grant it.

2. Preliminaries and definitions

2.1 Components

Hermes consists of four components, three of which form the server infrastructure of Hermes and the last one is a part of the client implementation. Note that a specific implementation of Hermes' scheme may consider and contain a particular set of features of the server-side infrastructure. That's why these client-server relations are conditional. The components of Hermes are:

1. The *Client* – an active entity related to the client side of Hermes. It produces and consumes the data. The *Client* can be represented by real clients who interact via some UI or by system processes that are communicating with Hermes' infrastructure via pre-defined application programming interfaces (APIs).
2. The *Data store* – a logical unit for storing and distributing protected data related to the server-side environment of Hermes. All the protected data in the Hermes' infrastructure is divided into "pieces" (records). In Hermes, there are no restrictions on how to divide the data into records. The access control policy in Hermes is set on a per-record basis. The *Data store* is responsible for authorising UPDATE operations on the records (described below).
3. The *Keystore* – a logical unit for storing and distributing an ACP (expressed as a collection of protected ACKs), related to the server-side environment of Hermes. The *Keystore* never denies key delivery to an authorised client and always fetches the client's key if it's available. The *Keystore* has no knowledge about the plain-text values of READ/UPDATE ACKs.
4. The *Credential store* – a logical unit for storing long-term Diffie-Hellman public keys that represent Hermes entities.

There are two types of cryptographic keys used within the Hermes' infrastructure:

1. Access control key (symmetric key K);
2. Long-term (static) Diffie-Hellman key pair (public key pk and private key sk).

Note that each entity within the Hermes' infrastructure possesses its own long-term Diffie-Hellman key pair and should securely store this private key by itself.

The following sections describe the security functions of Hermes' and the cryptographic enforcement of CRUD operations in detail.

2.2 Cryptographic context of Hermes

Hermes uses three high-level security functions for providing security for basic CRUD operations:

1. Data protection;
2. Access control policy management (creation, distribution, revocation);
3. Authorization of READ/UPDATE operations.

These security functions are implemented with a help of four cryptographic schemes: Symmetric encryption, Access control key wrapping, Message authentication codes, and Secure communication sessions. These are described in the following section.

2.3 Cryptographic schemes of Hermes

2.3.1 Symmetric encryption

The cryptographic scheme of symmetric encryption is a pair of algorithms SYM = (E, D).

1. The algorithm E (encryption algorithm) takes a key *K*, a plaintext *R*, and returns the ciphertext $R^K = \mathrm{E}_K(R)$.
2. The algorithm D (decryption algorithm) takes a key *K*, a purported ciphertext R^K and returns a value $\mathrm{D}_K(R^K)$. Consequently, $R = \mathrm{D}_K(\mathrm{E}_K(R))$.

The security of a symmetric encryption scheme is defined similarly to the definition from [13]. The preferable algorithms for the symmetric encryption include the well-known block cipher AES [14] or some of the stream ciphers from the ChaCha family [15].

2.3.2 Access control key wrapping

Access control key wrapping scheme is a pair of algorithms *WRAP* = (W, U).

1. The algorithm W (wrapping algorithm) takes a public key (from the long-term Diffie-Hellman key pair) *pk*, an ACK (a regular symmetric key *K*) and returns a wrapped ACK: $K^{pk} = \mathrm{W}_{pk}(\mathrm{K})$.
2. The algorithm U (unwrapping algorithm) takes a private key (from the long-term Diffie-Hellman key pair) *sk*, a wrapped ACK K^{pk}, and returns an unwrapped ACK: $K = \mathrm{U}_{sk}(K^{pk})$.

The security of access control key wrapping scheme is defined similarly to a common security definition of Diffie-Hellman-based DHETM scheme [16]. The preferable candidates for the ACK wrapping scheme include DHIES [13] or DHETM schemes.

2.3.3 Message authentication code

A message authentication code (MAC) is a single algorithm T (MAC generation algorithm) that takes a key K, and a message R, and returns a string $UT_{\mathrm{R}} = \mathrm{T}_K(R)$.

In the Hermes' terminology, this string is called an *Update Tag* (*UT*). Note that we use "implicit" verification of the *UT* described in "Authorization of READ/UPDATE operations" section below.

The security of T is defined similarly to the definition in [13]. The preferable candidate for MAC includes HMAC [5].

2.3.4 Secure session

Secure session scheme is used for establishing secure communication between two entities. Let's denote those entities as A (where A possesses a private key sk_A and a public key pk_A that form A's long-term Diffie-Hellman key pair) and B (where B possesses a private key sk_B and a public key pk_B that form B's long-term Diffie-Hellman key pair).

Secure session includes three phases:

1. Mutual authentication,
2. Key establishment, and
3. Secure data transmission.

During the *first* phase, A authenticates B and B authenticates A. This stage works under a standard assumption that long-term public keys pk_A and pk_B are exchanged via a third-party trusted Certificate Authority (CA). CA has verified that A indeed possesses sk_A that corresponds to the pk_A. The same is true for B [18]. In other words, a typical public-key infrastructure (PKI) [19] is used during this phase. In the Hermes' terminology, the *Credential store* server-side component may be considered as a CA.

During the *second* phase, A and B establish a common session encryption key using Diffie-Hellman based protocol KE. Actually, A uses KE that takes sk_A and pk_B and returns a shared secret $SS = \mathrm{KE}(sk_A, pk_B)$, while B uses KE that takes sk_B and pk_A and returns the same shared secret $SS = \mathrm{KE}(sk_A, pk_B)$. The security of KE is defined in [18]. The preferable candidates for the KE protocol include KEA, Unified Model, or MQV (all with Key Confirmation) schemes [18].

During the *third* phase, both A and B use a symmetric encryption scheme (defined above) to protect the data transmitted over the communication channel.

Note that all the communications between the components of Hermes are protected with a help of the Secure session scheme. The terms "requests", "receives", and "sends" (used in the following sections) imply that a mutually authenticated channel is established between the components and all the further transmitted data is encrypted.

2.4 Definitions of the high-level security functions of Hermes

2.4.1 Data protection

Data protection in Hermes acts as an access control-enforcing mechanism. All the sensitive data is encrypted using symmetric encryption (a traditional stream or block cipher can be used). If an entity possesses no appropriate symmetric cryptographic key(s), it will not be able to interpret/process the data. To make it possible to apply different ACPs to different parts of the data, it is logically segmented into pieces – records. There are no restrictions (imposed by Hermes) on how to divide the data into records. Let's denote an arbitrary record as R to be able to define the plain-text data as a set of records:

$$\text{DATA} = (R_1, R_2, ..., R_n).$$

Having *n* records $R_1, R_2, ..., R_n$ and n keys $K_1, K_2, ..., K_n$, we can define the encrypted data as follows:

$$\text{ENCRYPTED\quad DATA} = (\mathrm{E}_{K_1}(R_1), \mathrm{E}_{K_2}(R_2), ..., \mathrm{E}_{K_n}(R_n)).$$

Having *n* records and the corresponding keys, we can define the decrypted data as follows:

$$\text{DECRYPTED\quad DATA} = (\mathrm{D}_{K_1}(\mathrm{E}_{K_1}(R_1)), \mathrm{D}_{K_2}(\mathrm{E}_{K_2}(R_2)), ..., \mathrm{D}_{K_n}(\mathrm{E}_{K_n}(R_n))).$$

Note that such segmented state is not natural for the data. The division of data into sets of records is conditional (all the records remain concatenated) – a blob of data can be interpreted as single record or a collection of records. This is achieved through generating a number of data structures that control distribution of symmetric keys.

The intention of such data segmentation is to provide flexible protection with the help of data access compartmentalisation where the goal is to maximally reduce the size of the smallest possible data fragment protected with a single key. While the specific nature of how data is segmented is an issue of implementation, defining a record to be the smallest element of data protected by a single key enables Hermes to offer highly granular access control.

By appropriately (securely) distributing the keys $K_1, K_2, ..., K_n$ to a set of entities (via the *Keystore*), it is possible to create an arbitrary ACP that allows those entities to access only limited parts of the data with per-record granularity.

Note that as we are relying on cryptographic keys to limit the access to different parts of the data, we need to ensure that these keys do not have statistical interdependencies. That's why each key $K_1, K_2, ..., K_n$ is generated independently from each other.

2.4.2 Access control policy management

As mentioned above, an ACP is expressed as a set of protected ACKs stored in the *Keystore* related to a set of encrypted records stored in the *Data store*. Let's consider a partial ACP related to a single record.

ACP Creation

In its simplest form, an ACP starts with some client (*Alice*) who wishes to secure and store a record *R*. *Alice* will generate two ACKs, a READ key K_{R_read} and an UPDATE key K_{R_update}. *Alice* encrypts *R* using K_{R_read} and generates a MAC for R using K_{R_update}. She then sends the encrypted record and the MAC to the *Data store*.

Alice now will establish a minimal ACP that enables her to subsequently READ and UPDATE the record *R* using the symmetric encryption and ACK wrapping schemes. Let's suppose a client *Alice* (who has a long-term Diffie-Hellman key pair (pk_A, sk_A)) has sent a record *R* (actually, an encrypted record $\mathrm{E}_{K_{R\_read}}(R)$ and an *Update Tag* $UT_R = \mathrm{T}_{K_{R\_update}}(R)$) to the *Data store*.

Now *Alice* should create an initial ACP for the record *R* by wrapping K_{R_read} and K_{R_update} that she has previously generated:
$K_{R\_read}^{pk_A} = \mathrm{W}_{pk_A}(\mathrm{K}_{R\_read})$, $K_{R\_update}^{pk_A} = \mathrm{W}_{pk_A}(\mathrm{K}_{R\_update})$ and posting $K_{R\_read}^{pk_A}$, $K_{R\_update}^{pk_A}$ to the *Keystore*.

Only *Alice* is able to READ the record *R* (only Alice is able to unwrap $K_{R\_update}^{pk_A}$ that she received from the *Keystore* using her private key sk_A and, consequently, decrypt the encrypted record $\mathrm{E}_{K_{R\_read}}(R)$ that *Alice* received from the *Data store*) and to UPDATE record *R* (only *Alice* is able to unwrap $K_{R\_update}^{pk_A}$ she received from the *Keystore* using sk_A and, consequently, calculate a correct *Update Tag* UT_R) that will be validated by the *Data store* during the processing of an UPDATE transaction.

ACP Distribution

We can define the ACP distribution mechanism that is very similar to ACP creation (as it uses the same cryptographic schemes). Again, let's suppose *Alice* (having her long-term Diffie-Hellman key pair (pk_A, sk_A)) has sent a record *R* (actually, an encrypted record $\mathrm{E}_{K_{R\_read}}(R)$ and an *Update Tag* $\mathrm{T}_{K_{R\_update}}(R)$) to the *Data store*, then has created an initial ACP for the record *R* (actually *Alice* has sent a wrapped ACKs $K_{R\_read}^{pk_A}$ and $K_{R\_update}^{pk_A}$ to the *Keystore*) and now *Alice* wants to grant READ access to the record *R* to another client – *Bob* (who has his long-term Diffie-Hellman key pair (pk_B, sk_B)).

To do this, *Alice* should securely distribute K_{R_read} to *Bob*. Actually, *Alice* should get $K_{R\_read}^{pk_A}$ from the *Keystore*, unwrap it: $K_{R\_read} = U_{sk_A}(K_{R\_read}^{pk_A})$, and wrap it using *Bob*'s public key pk_B:
$K_{R\_read}^{pk_B} = \mathrm{W}_{pk_B}(\mathrm{K}_{R\_read})$ and finally send $K_{R\_read}^{pk_B}$ to the *Keystore*.

Now the ACP for the record *R* defines READ/UPDATE access for *Alice* ($K_{R\_read}^{pk_A}$, $K_{R\_update}^{pk_A}$) and READ access for *Bob* ($K_{R\_read}^{pk_B}$).

Note that granting the UPDATE access to *Bob* is almost similar, but here it is K_{R_update} instead of K_{R_read} that should be wrapped with *Bob*'s public key pk_B and posted to the *Keystore*.

ACP Revocation

We can define the ACP revocation mechanism as follows. Let's suppose *Alice* (who has a long-term Diffie-Hellman key pair (pk_A, sk_A)) has previously granted the READ and UPDATE access rights to the posted record *R* to *Bob* (who has a long-term Diffie-Hellman key pair (pk_B, sk_B)) and *Eve* (who has a long-term Diffie-Hellman key pair (pk_E, sk_E)), and now *Alice* wants to revoke the UPDATE access rights from *Bob*.

The *Keystore* stores the ACP (to the record *R*) which is described in the table below.

	Alice	*Bob*	*Eve*
Access to record *R*	READ/UPDATE	READ/UPDATE	READ/UPDATE

This ACP is represented via ACKs in the table below.

	Alice	*Bob*	*Eve*
Access to record *R*	$K_{R\_read}^{pk_A}$, $K_{R\_update}^{pk_A}$	$K_{R\_read}^{pk_B}$, $K_{R\_update}^{pk_B}$	$K_{R\_read}^{pk_E}$, $K_{R\_update}^{pk_E}$

After revocation, the ACP should be changed to:

	Alice	*Bob*	*Eve*
Access to record *R*	READ/UPDATE	READ	READ/UPDATE

This is represented in terms of ACKs as follows:

	Alice	*Bob*	*Eve*
Access to record *R*	$K_{R\_read}^{pk_A}$, $K_{R\_update\_new}^{pk_A}$, $K_{R\_update}^{pk_A}$	$K_{R\_read}^{pk_B}$, $K_{R\_update}^{pk_B}$	$K_{R\_read}^{pk_E}$, $K_{R\_update\_new}^{pk_E}$, $K_{R\_update}^{pk_E}$

One can see that access revocation (of both READ and UPDATE permissions) involves generating a new value for the corresponding ACK, updating the *Data store* content using the new ACK and then redistributing the new ACK to the clients who should retain access.

Actually, to revoke *Bob's* UPDATE permission to *R*, *Alice* should restore the existing ACP to *R* (getting all the UPDATE ACKs associated with *R* from the *Keystore*). Then *Alice* generates a new UPDATE ACK $K_{R\_update\_new}$, performs an authorized updating transaction with *R* (see section 3.3 for

more details), wraps $K_{R\_update\_new}$ twice: $K^{pk_A}_{R\_read\_new} = \mathrm{W}_{pk_A}(\mathrm{K}_{R\_read\_new})$,
$K^{pk_E}_{R\_read\_new} = \mathrm{W}_{pk_E}(\mathrm{K}_{R\_read\_new})$, and finally sends $K^{pk_A}_{R\_update\_new}, K^{pk_E}_{R\_update\_new}$ to the *Keystore*.
Now *Bob* is unable to unwrap $K_{R\_update\_new}$ (and is also unable to calculate UT_R^{new}) and, consequently, *Bob* is unable to perform UPDATE.

Note that revocation of READ permissions is similar, but contains some additional steps: *Alice* revokes UPDATE permission as described above and then additionally generates a new READ ACK $K_{R\_read\_new}$, re-encrypt *R*: $R^{K_{R\_read\_new}} = \mathrm{E}_{K_{R\_read\_new}}(R)$, performs an authorized updating transaction with *R* (see section 3.3 for more details), wraps $K_{R\_read\_new}$ twice: $K^{pk_A}_{R\_read\_new} = \mathrm{W}_{pk_A}(K_{R\_read\_new})$,
$K^{pk_E}_{R\_read\_new} = \mathrm{W}_{pk_E}(K_{R\_read\_new})$, and finally sends $K^{pk_A}_{R\_read\_new}$, $K^{pk_E}_{R\_read\_new}$ to the *Keystore*.

Implementation note:

While the access control objectives of the system are achieved by generating the new set of ACKs, *UTs* and encrypted records, a practical implementation (of the *Keystore*) should have means of disposing of the previously valid keys $K^{pk_A}_{R\_update}$, $K^{pk_B}_{R\_update}$, $K^{pk_E}_{R\_update}$ that became redundant for both performance and security reasons (see [5]).

Security notes:

1. If *Bob* obtains READ permissions to the record, he will be able to transfer this READ permission to a third client. *Alice* won't be able to prevent this. The same applies to READ/UPDATE permissions, which can be propagated (granted) by clients who possess these permissions to other clients, without limitations. Traitor tracing questions [20] are out of scope of Hermes.
2. UPDATE access to the record can't be performed without having READ permissions for the same record. That's why granting/revoking of UPDATE access rights to *R* should be accompanied by verification of the client's possession of READ access rights to *R*.
3. If *Alice* revokes *Bob*'s READ access rights to a record *R* without changing the content of *R*, it's worth remembering that *Bob* had had a previous opportunity to access to the sensitive data of *R* and could have remembered/copied/transferred it. Revocation of READ permission would only make sense if *Alice* modifies *R* and wants *Bob* to have no knowledge of performed modifications.

2.5 Authorisation of READ/UPDATE operations

Authorization of READ/UPDATE operations is implemented with a help of symmetric encryption and MAC schemes. As it was mentioned above, the possession of READ ACK to a record *R* gives a client the ability to decrypt the encrypted record *R* and read its contents.

As with READ, possession of UPDATE ACK to a record *R* gives a client the ability to calculate the correct *UT* and to prove the permission to perform UPDATE on the record *R*.

All the records $R_1, ..., R_n$ are stored in the *Data store* with appropriate *UTs*: $UT_1, ..., UT_n$.

The *Update Tags* are never fetched back from the *Data store* as a part of the data query. Rather it is the *Data store* that uses them to verify the client's possession of an appropriate UPDATE ACK and, consequently, the permission to perform UPDATE.

Let's suppose *Alice* (who has a long-term Diffie-Hellman key pair (pk_A, sk_A)) has previously sent the record *R* (encrypted record $\mathrm{E}_{K_{A\_read}}(R)$ and the *Update Tag* $UT_R = \mathrm{T}_{K_{R\_update}}(R)$) to the *Data store*, defined initial ACP to *R* (posted $K_{R\_read}^{pk_A}$, $K_{R\_update}^{pk_A}$ to the *Keystore),* and wants to perform READ (in phase 1) and UPDATE (in phase 2) operations. Actually, *Alice* does the following:

Phase 1. READ authorisation:

1. Requests and receives $K_{R\_read}^{pk_A}$ from the *Keystore* and unwraps it: $K_{R\_read} = U_{sk_A}(K_{R\_read}^{pk_A})$;
2. Requests and receives $\mathrm{E}_{K_{R\_read}}(R)$ from the *Data store* and decrypts it: $R = \mathrm{D}_{K_{R\_read}}(\mathrm{E}_{K_{R\_read}}(R))$. Now Alice is able to READ the record *R*.

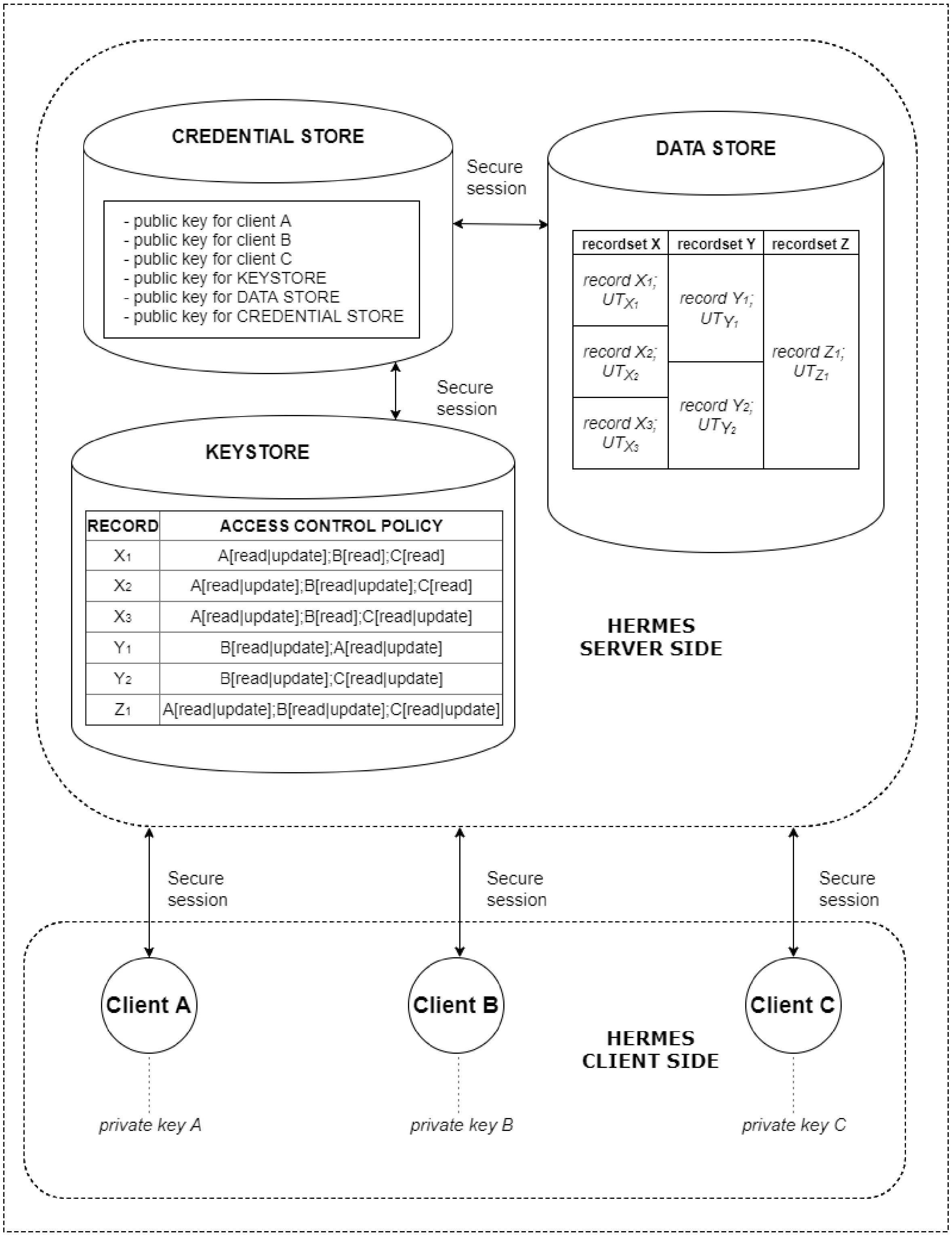
Phase 2. UPDATE authorisation:

1. Performs **Phase 1**;
2. Requests and receives $K_{R\_update}^{pk_A}$ from the *Keystore* and unwraps it: $K_{R\_update} = U_{sk_A}(K_{R\_update}^{pk_A})$;
3. Updates *R*: $R \to R_{updated}$;
4. Calculates MAC for *R* and $R_{updated}: UT_R = \mathrm{T}_{K_{R\_update}}(R)$, $UT_{R_{updated}} = \mathrm{T}_{K_{R\_update}}(R_{updated})$;
5. Encrypts $R_{updated}$: $R_{updated}^{K_{R\_read}} = E_{K_{R\_read}}(R_{updated})$;
6. Sends UT_R, $R_{updated}^{K_{R\_read}}$ and $UT_{R_{updated}}$ to the *Data store*;
7. The *Data store* compares the UT_R sent by Alice with the stored UT_R. If they match, the *Data store* authorizes Alice's UPDATE operation; otherwise the *Data store* ignores the requested UPDATE operation.

Note that the stored *UTs* are never exposed by the *Data store* and can only be generated with the knowledge of a proper UPDATE ACK. This validation mechanism allows the *Data store* to perform the UPDATE authorization without the knowledge of the UPDATE ACK value.

2.6 The functional scheme of Hermes

The following illustration demonstrates a functional scheme of a Hermes infrastructure with three clients (A, B, C) and defined access control policy to the recordsets X, Y, Z and corresponding records).



RECORD	ACCESS CONTROL POLICY
X_1	A[read|update];B[read];C[read]
X_2	A[read|update];B[read|update];C[read]
X_3	A[read|update];B[read];C[read|update]
Y_1	B[read|update];A[read|update]
Y_2	B[read|update];C[read|update]
Z_1	A[read|update];B[read|update];C[read|update]

Figure 1: The functional scheme of Hermes.

One can see that the *Keystore* keeps all the information about the current ACP to all data (encrypted records: *X1*, *X2*, *X3*, *Y1*, *Y2*, *Z1* with their *Update Tags*: UT_{X_1}, UT_{X_2}, UT_{X_3}, UT_{Y_1}, UT_{Y_2}, UT_{Z_1}) in the *Data store*.
The following table demonstrates what is actually stored in the *Keystore*, according to the notation above:

RECORD	**ACCESS CONTROL POLICY**
X_1	A [read | update]; B[read]; C[read].
X_2	A[read | update]; B[read | update]; C[read].
X_3	A[read | update]; B[read]; C[read | update].
Y_1	B[read | update]; A[read | update].
Y_2	B[read | update]; C[read | update].
Z_1	A[read | update]; B[read | update]; C[read | update].

This table equals the following table:

RECORD	**ACCESS CONTROL POLICY**
X_1	$K^{pk_A}_{X_1\_read}$, $K^{pk_A}_{X_1\_update}$; $K^{pk_B}_{X_1\_read}$; $K^{pk_C}_{X_1\_read}$.
X_2	$K^{pk_A}_{X_2\_read}$, $K^{pk_A}_{X_2\_update}$; $K^{pk_B}_{X_2\_read}$, $K^{pk_B}_{X_2\_update}$; $K^{pk_C}_{X_2\_read}$.
X_3	$K^{pk_A}_{X_3\_read}$, $K^{pk_A}_{X_3\_update}$; $K^{pk_B}_{X_3\_read}$; $K^{pk_C}_{X_3\_read}$, $K^{pk_C}_{X_3\_update}$.
Y_1	$K^{pk_B}_{Y_1\_read}$, $K^{pk_B}_{Y_1\_update}$; $K^{pk_A}_{Y_1\_read}$, $K^{pk_A}_{Y_1\_update}$.
Y_2	$K^{pk_B}_{Y_2\_read}$, $K^{pk_B}_{Y_2\_update}$; $K^{pk_C}_{Y_2\_read}$; $K^{pk_C}_{Y_2\_update}$.
Z_1	$K^{pk_A}_{Z_1\_read}$, $K^{pk_A}_{Z_1\_update}$; $K^{pk_B}_{Z_1\_read}$, $K^{pk_B}_{Z_1\_update}$; $K^{pk_C}_{Z_1\_read}$, $K^{pk_C}_{Z_1\_update}$.

To clarify the notation, let's remember that, for example, $K^{pk_A}_{Z_1\_read}$ means that this ACK can be used by client *A* to perform READ operation on the record *Z1* (*Z1* is encrypted on this ACK: ${Z_1}^{K_{Z_1\_read}} = \mathrm{E}_{K_{Z_1\_read}}(Z_1)$, while the ACK is itself wrapped: $K^{pk_A}_{Z_1\_read} = \mathrm{W}_{pk_A}(\mathrm{K}_{Z_1\_read})$).

3 CRUD implementation

Using the security functions of Hermes outlined above, we can implement all the four functions of persistent storage.

The READ function is explicitly defined by data protection and READ authorisation; the UPDATE function is explicitly defined by data protection and UPDATE authorisation; the CREATE and DELETE functions are not directly supported by Hermes, but can be defined implicitly.

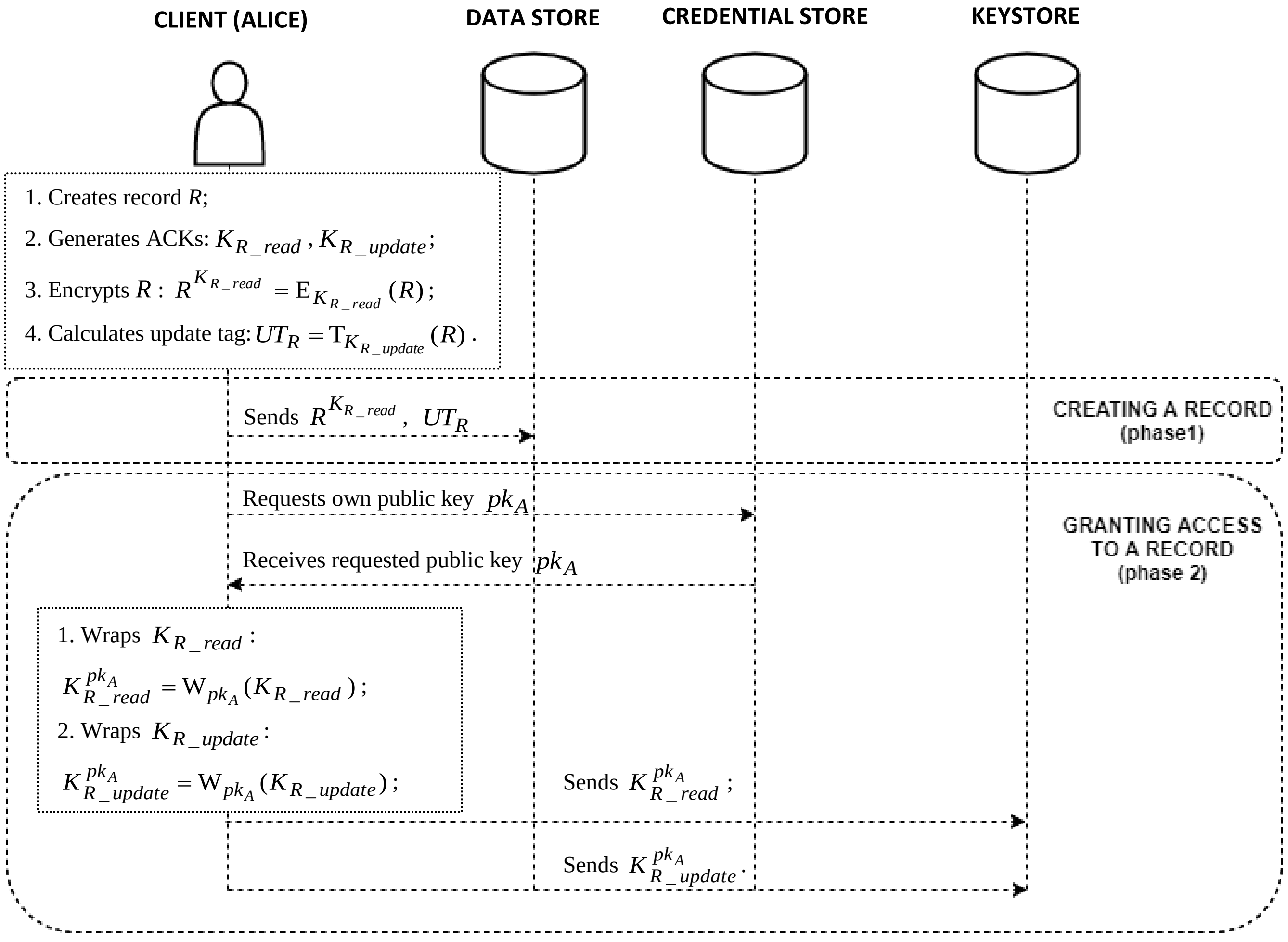
The CREATE function is defined by operation of the READ and UPDATE functions plus the ACP management (creation and distribution) security function, while the DELETE function is defined by operation of the UPDATE function with a nullable record. Granting rights to CREATE is out of scope of Hermes methodology. However, several approaches are suggested in [5].

3.1 CREATE

The process of creation of a record *R* consists of two phases:

1. *R* is created, protected, sent, and stored in the *Data store*;
2. The access control policy for the record *R* (which is now stored in the *Data store*) is created and it is sent to the *Keystore*.

The overall process of creation of the record *R* and access control policy distribution for it is outlined in the following illustration:



CLIENT (ALICE)
DATA STORE
CREDENTIAL STORE
KEYSTORE
1. Creates record R;
2. Generates ACKs: K_{R_read}, K_{R_update};
3. Encrypts R: $R^{K_{R\_read}} = \mathrm{E}_{K_{R\_read}}(R)$;
4. Calculates update tag: $UT_R = \mathrm{T}_{K_{R\_update}}(R)$.
Sends $R^{K_{R\_read}}$, UT_R
CREATING A RECORD (phase1)
Requests own public key pk_A
Receives requested public key pk_A
GRANTING ACCESS TO A RECORD (phase 2)
1. Wraps K_{R_read}:
$K_{R\_read}^{pk_A} = \mathrm{W}_{pk_A}(K_{R\_read})$;
2. Wraps K_{R_update}:
$K_{R\_update}^{pk_A} = \mathrm{W}_{pk_A}(K_{R\_update})$;
Sends $K_{R\_read}^{pk_A}$;
Sends $K_{R\_update}^{pk_A}$.

Figure 2: CREATE.

Note that *Alice* may create an empty record (null) and grant READ/UPDATE permissions to this record to *Bob*. *Bob* then can perform UPDATE on that record. This way, *Alice* only acts as a creator of the ACP, while *Bob* acts as data provider (see [5] for more details).

3.2 READ

The process of performing READ on the record *R* consists of two phases:

1. Getting and unwrapping the READ ACK to *R* from the *Keystore*;
2. Getting and decrypting the encrypted *R* form the *Data store*.

The overall process of reading of the record R is outlined in the following illustration:

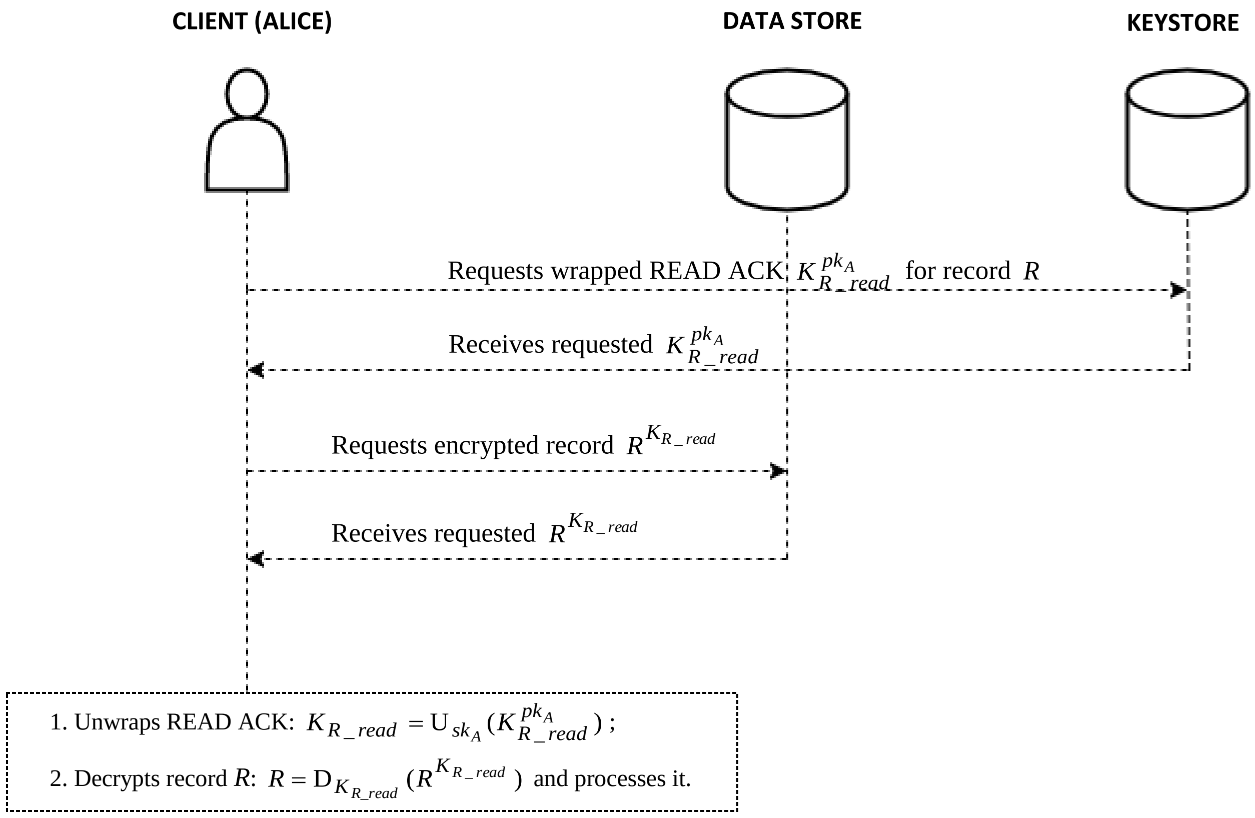

CLIENT (ALICE)
DATA STORE
KEYSTORE
Requests wrapped READ ACK $K_{R\_read}^{pk_A}$ for record R
Receives requested $K_{R\_read}^{pk_A}$
Requests encrypted record $R^{K_{R\_read}}$
Receives requested $R^{K_{R\_read}}$
1. Unwraps READ ACK: $K_{R\_read} = \mathrm{U}_{sk_A}(K_{R\_read}^{pk_A})$;
2. Decrypts record *R*: $R = \mathrm{D}_{K_{R\_read}}(R^{K_{R\_read}})$ and processes it.

Figure 3: READ.

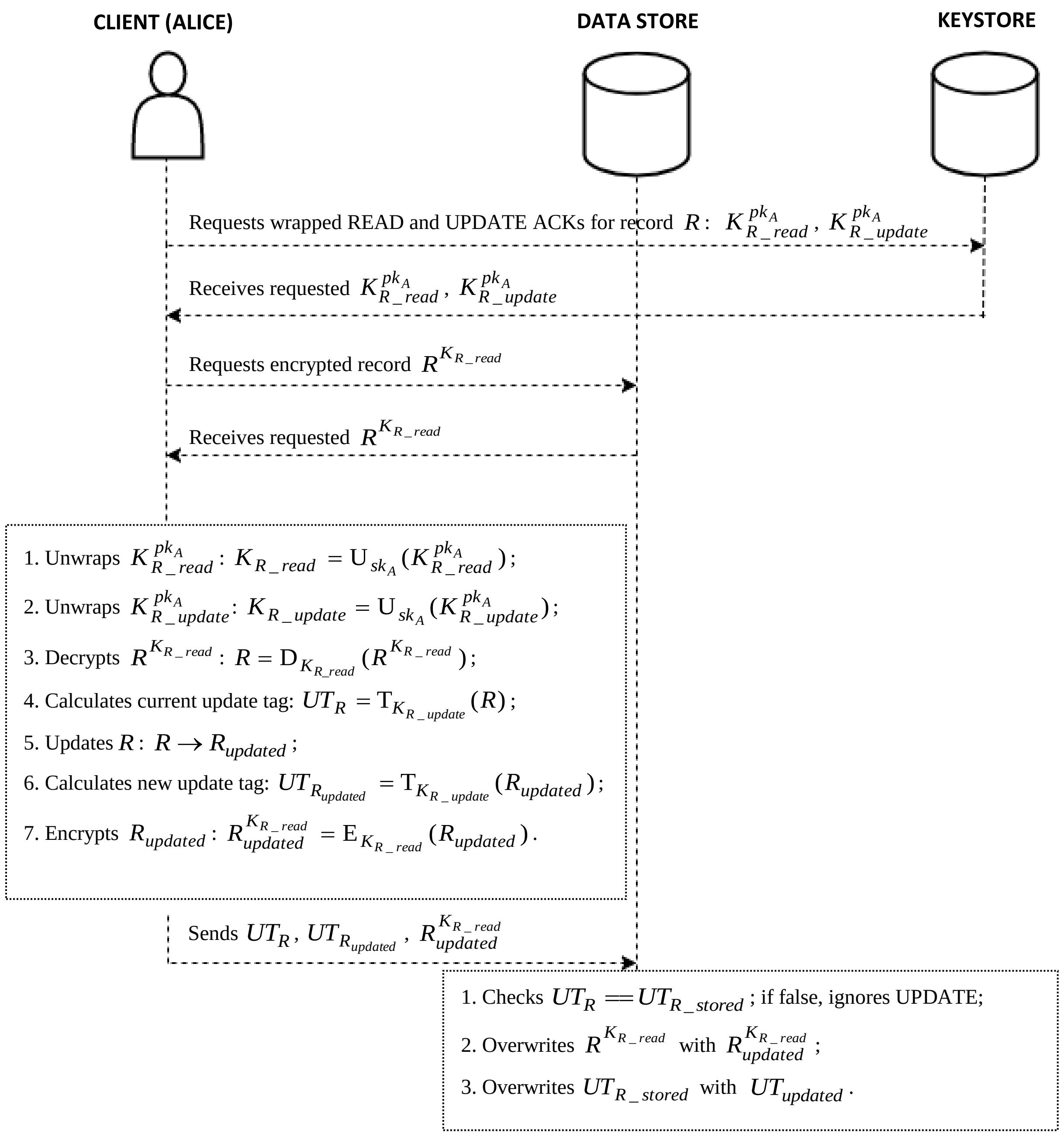
Note that *Alice* doesn't receive (nor she is permitted to receive) the update tag UT_R, which is considered to be secret information with respect to other clients with READ permissions to *R* (see [5] for more details).

3.3 UPDATE

The process of performing UPDATE on the record *R* (in other words – updating the record *R*) consists of three phases:

1. Getting and unwrapping the READ and UPDATE ACKs to the record *R* from the *Keystore*;
2. Getting and decrypting the encrypted record *R* from the *Data store*;
3. Updating the record *R* and authorising the updates.

The overall process of updating the record *R* is outlined in the following illustration:



CLIENT (ALICE)
DATA STORE
KEYSTORE
Requests wrapped READ and UPDATE ACKs for record R: $K_{R\_read}^{pk_A}$, $K_{R\_update}^{pk_A}$
Receives requested $K_{R\_read}^{pk_A}$, $K_{R\_update}^{pk_A}$
Requests encrypted record $R^{K_{R\_read}}$
Receives requested $R^{K_{R\_read}}$
1. Unwraps $K_{R\_read}^{pk_A}$: $K_{R\_read} = \mathrm{U}_{sk_A}(K_{R\_read}^{pk_A})$;
2. Unwraps $K_{R\_update}^{pk_A}$: $K_{R\_update} = \mathrm{U}_{sk_A}(K_{R\_update}^{pk_A})$;
3. Decrypts $R^{K_{R\_read}}$: $R = \mathrm{D}_{K_{R\_read}}(R^{K_{R\_read}})$;
4. Calculates current update tag: $UT_R = \mathrm{T}_{K_{R\_update}}(R)$;
5. Updates R: $R \to R_{updated}$;
6. Calculates new update tag: $UT_{R_{updated}} = \mathrm{T}_{K_{R\_update}}(R_{updated})$;
7. Encrypts $R_{updated}$: $R_{updated}^{K_{R\_read}} = \mathrm{E}_{K_{R\_read}}(R_{updated})$.
Sends UT_R, $UT_{R_{updated}}$, $R_{updated}^{K_{R\_read}}$
1. Checks $UT_R == UT_{R\_stored}$; if false, ignores UPDATE;
2. Overwrites $R^{K_{R\_read}}$ with $R_{updated}^{K_{R\_read}}$;
3. Overwrites UT_{R_stored} with $UT_{updated}$.

Figure 4: UPDATE.

Upon receiving the data, the *Data store* verifies that *Alice* possesses the valid UPDATE ACK by checking if the equation $UT_R == UT_{R\_stored}$ holds (if it doesn't hold, the *Data store* ignores the requested UPDATE) and overwrites the stored encrypted record and *Update Tag* with new values. This way the *Data store* never processes the records in plaintext.

3.4 DELETE

Deleting records is a particular simplified case of performing UPDATE on the records we want to delete. In DELETE, the updated record becomes NULL, so there is no need for the updater to encrypt the updated record and put it into the *Data store*.

The overall process of DELETING a record *R* is outlined in the following illustration:

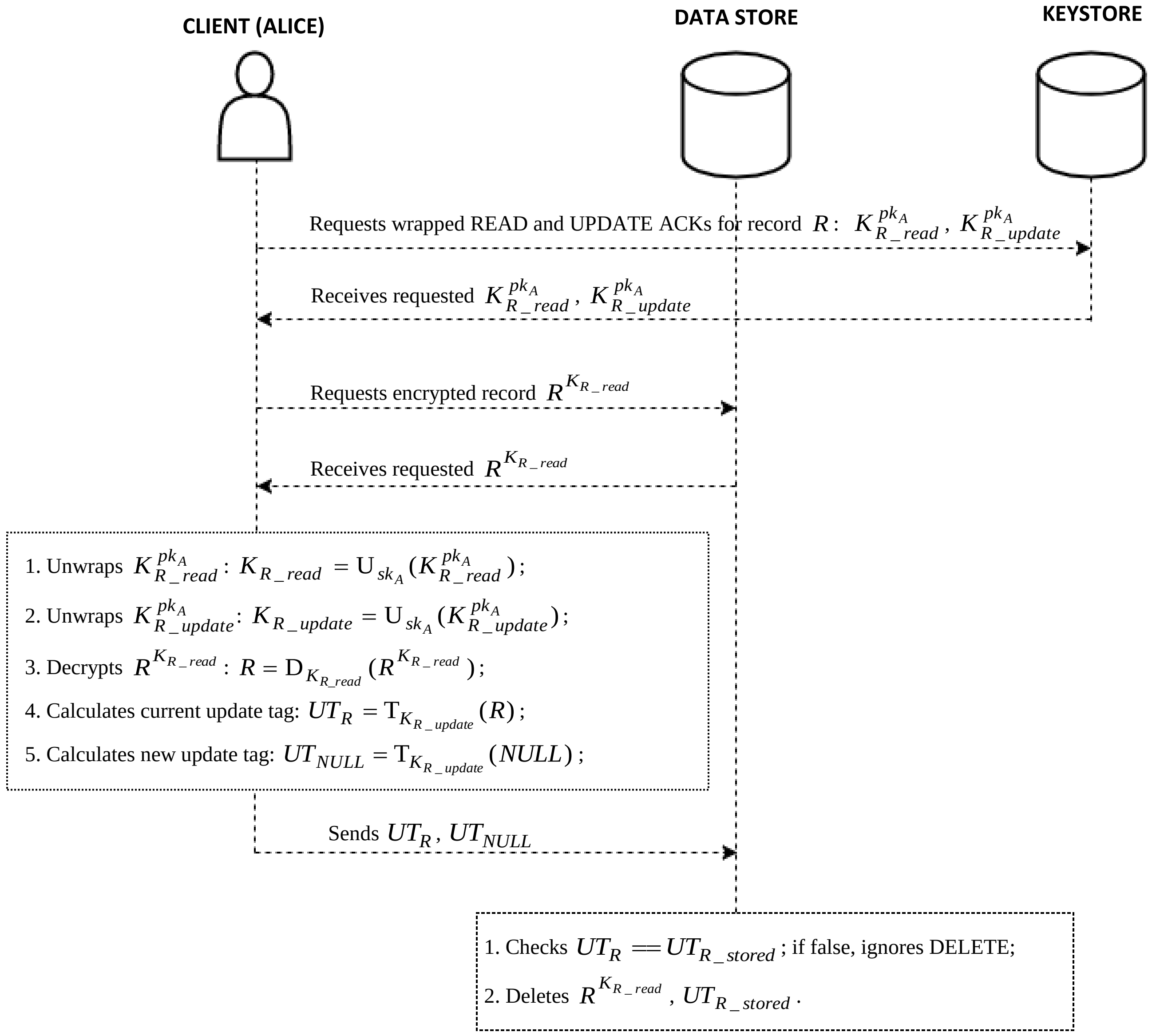

CLIENT (ALICE)
DATA STORE
KEYSTORE
Requests wrapped READ and UPDATE ACKs for record R : $K_{R\_read}^{pk_A}$, $K_{R\_update}^{pk_A}$
Receives requested $K_{R\_read}^{pk_A}$, $K_{R\_update}^{pk_A}$
Requests encrypted record $R^{K_{R\_read}}$
Receives requested $R^{K_{R\_read}}$
1. Unwraps $K_{R\_read}^{pk_A}$: $K_{R\_read} = \mathrm{U}_{sk_A}(K_{R\_read}^{pk_A})$;
2. Unwraps $K_{R\_update}^{pk_A}$: $K_{R\_update} = \mathrm{U}_{sk_A}(K_{R\_update}^{pk_A})$;
3. Decrypts $R^{K_{R\_read}}$: $R = \mathrm{D}_{K_{R\_read}}(R^{K_{R\_read}})$;
4. Calculates current update tag: $UT_R = \mathrm{T}_{K_{R\_update}}(R)$;
5. Calculates new update tag: $UT_{NULL} = \mathrm{T}_{K_{R\_update}}(NULL)$;
Sends UT_R , UT_{NULL}
1. Checks $UT_R == UT_{R\_stored}$; if false, ignores DELETE;
2. Deletes $R^{K_{R\_read}}$, UT_{R_stored} .

Figure 5: DELETE.

Security note:

The illustration (**Figure 4**) in the section 3.3 considers the *UT* to be sensitive information. This may at first seem unusual as the *UT* is simply a MAC. But in this context, the *UT* is actually a token that authenticates a secure operation. This approach enables the operation of the *Data store* to be restricted solely to the store and retrieval of encrypted data. While leaking the *UT* does not reveal any information about the record itself, this does potentially give an attacker the opportunity to replace the record data with garbage data (or to blindly delete the record). It is the role of higher-level application software to detect missing or corrupted data and that reliable backup procedures are in place for critical records.

4 Security considerations

4.1 Threat model and security assumptions

Hermes is designed to operate under a restrictive threat model in which:

1. An attacker (external attacker or malicious server) may compromise one or more components such that:
   1.1 partial or complete ciphertext leakage may occur,
   1.2 partial or complete ACP model leakage may occur;
2. A trusted client may "behave" dishonestly;
3. An external passive/active attacker may be present in the communication channels.

This model allows us to make some basic security assumptions about the operational process within Hermes and assess them as part of the overall security evaluation.

Assumption 1.	Hermes clients are trusted entities.
Assumption 2.	Encrypted data and ACP to these data circulate only within the Hermes infrastructure (and even if ciphertext is leaked, this will not compromise the whole system). Sensitive data appears as plain-text only in the client's context.
Assumption 3.	Basic execution environments of all Hermes' components (hardware or operating systems) are trustworthy. Each component strictly follows the set of actions defined and allowed by Hermes.
Assumption 4.	All the Hermes' data storage components support backup and logging mechanism. They also operate correctly and without failures.
Assumption 5.	All the communication between Hermes' components is performed via authenticated and encrypted channels.
Assumption 6.	All the cryptographic primitives used are well-studied (have proven security) by experts and are properly implemented.

Practical extensions to these assumptions (temporary hardware failures, data corruption, denial of service of some components, appropriateness of crypto implementation to side-channel risks of certain platforms etc.) are assessed in the implementation considerations [5].

4.2 Trusted clients

Within Hermes, clients are trusted to:

1. Properly operate with the data according to the defined ACP;
2. Further distribute access rights to the data they've obtained access to, to other clients (a client with certain access rights may only distribute access rights of the same level or lower – i.e. a client with permission to READ and UPDATE may further distribute permissions to READ and UPDATE, or just to READ to another client).

4.3 Security analysis

Hermes separates the important cryptographic operations from the network-facing code. This greatly reduces the potential attack surface, minimises the damage from discovered zero-day vulnerabilities [21], and simplifies the security audit of the system.

In our security analysis we consider the four key Hermes components (*Client*, *Data store*, *Keystore*, *Credential store*) together with their communication from the standpoint of the degree of compromise of the system and its type.

We use the term "compromisation" to indicate the most severe and complete form of breakage of an entity's defenses (as in "adversary gains total control").

The table below demonstrates 5 levels of compromisation, ordered by the increase in the enemy's capabilities and the negative consequences for the system respectively.

Level	**Compromised entities**	**Adversary capabilities**	**The worst security consequences**
1	Passive access to all plaintext communication channels	Can read wrapped ACKs transmitted over channel	Ciphertext-only attack (COA) on ACK wrapping[1]
		Can read encrypted sensitive data transmitted over channel	COA on data protection[2]
		Can read *UTs* of encrypted records transmitted over channel	COA on MAC[3]

2	Active access to all plaintext communication channels	All capabilities of 1	COA on ACK wrapping, data protection and MAC
		Can block all the communication channels[4]	DoS[5]
3	*Keystore*	Can read wrapped ACKs	COA on ACK wrapping
		Can delete wrapped ACKs	DoS
		Can write garbage instead of ACKs	Unauthorised access revocation / DoS
		Can disable *Keystore*	DoS
4	*Credential store*	Can read public credentials	-
		Can delete public credentials	DoS
		Can write forged public credentials	Impersonation
		Can disable *Credential store*	DoS
5	*Data store*	Can read encrypted data and *UTs*	COA on data protection and MAC
		Can delete encrypted sensitive data	DoS
		Can write garbage instead of data	DoS
		Can force all *UT* verifications to succeed/fail	Unauthorised UPDATE, DELETE, access revocation / DoS
		Can disable *Data store*	DoS
6	*Keystore*, *Data store*	All capabilities of 3, 5	COA on ACK wrapping, DoS, unauthorised access revocation / DoS, COA on data protection and MAC, unauthorised UPDATE / DELETE

7	*Keystore*, *Credential store*	All capabilities of 3, 4	COA on ACK wrapping, DoS, unauthorised access revocation / DoS, impersonation
8	*Credential store*, *Data store*	All capabilities of 4, 5	DoS, impersonation, COA on data protection and MAC, unauthorised UPDATE, DELETE
9	*Keystore*, *Credential store*, *Data store*	All capabilities of 3, 4, 5	COA on ACK wrapping, DoS, unauthorised access revocation / DoS, impersonation, COA on data protection and MAC, unauthorised UPDATE, DELETE

[1] COA on ACK wrapping – Ciphertext-only attack on the key wrapping algorithm.
[2] COA on data protection – Ciphertext-only attack on the encryption algorithm.
[3] COA on MAC – Ciphertext-only attack on the message authentication code.
[4] Block communication channels – Man-in-the-middle adversary is able to prevent sending / receiving of data (e.g. deleting or modifying it) for the system clients/services.
[5] Full DoS – Complete system's denial of service, with (optional) full or partial data loss.

Practical attacker types and non-cryptosystem mitigation for many attacks are assessed in the implementation [5].

4.4 Security guarantees

We would assert that Hermes offers the following security guarantees:

1. Compromisation of a single entity in the system causes only limited damage;
2. All the sensitive information appears in plain text only within the client's context;
3. Data is protected in granular form (per-record);
4. All communications are protected with end-to-end encryption and authentication;
5. *Data store* imports/stores/exports only veritable protected data;
6. *Keystore* imports/stores/exports only veritable wrapped ACKs;
7. *Credential store* imports/stores/exports only veritable public credentials;
8. Each data record is protected with a unique key;
9. Each data record has legitimate access control policy.

The following table demonstrates the security guarantees safeguarded by Hermes in the case of a system compromisation.

Compromisation level	**Compromised entities**	**Number of safeguarded guarantees**
1	Passive access to all communication channels	1, 2, 3, 4, 5, 6, 7, 8, 9
2	Active access to all communication channels	1, 2, 3, 5, 6, 7, 8, 9
3	*Keystore*	1, 2, 3, 4, 5, 7, 8
4	*Credential store*	1, 2, 3, 4, 5, 6, 8, 9
5	*Data store*	1, 2, 4, 6, 7, 9
6	*Keystore*, *Data store*	1, 2, 4, 7
7	*Keystore*, *Credential store*	1, 2, 3, 4, 5, 8
8	*Credential store*, *Data store*	1, 2, 4, 6, 9
9	*Keystore*, *Credential store*, *Data store*	1, 2, 4

In summary, even in the most extreme case of compromisation where all the server side components are compromised, several security guarantees are preserved and damage is limited.

An adversary can potentially perform a denial-of-service attack on the complex infrastructure and ciphertext-only attacks on ACKs wrapping, data protection, and MAC in response to the considered security assumptions.

Some damage can also be done in the following cases:

1. Unauthorised access revocation (see the second attack in the next section – *Unauthorised revocation of UPDATE permissions*);
2. Impersonation – if the *Credential store* gets broken;
3. Unauthorised UPDATE (only for a legitimate client with READ permissions, who broke the *Data store*),
4. DELETE, access revocation / DoS – in a case when the *Data store* gets broken.

4.5 Potential attacks

During the security analysis, we've identified the most probable compromisations of the base assumptions that lead to actual risks. It is worth noting that for an active adversary performing such attack would be very hard, since the breakage points are popular, well studied cryptographic mechanisms.

4.5.1 Unauthorised UPDATE

Conditions:

1. A malicious client *Eve* who has READ permission to the record *R* (meaning that *Eve* is able to unwrap the wrapped READ ACK $K_{R\_read}^{pk_E}$);
2. An active access to the communication channel between an honest client *Bob* who has READ/UPDATE permissions (*Bob* is able to unwrap both wrapped READ and wrapped UPDATE ACKs $K_{R\_read}^{pk_B}$, $K_{R\_update}^{pk_B}$).

This attack violates security *Assumption 5*.

Description:
Eve is present in the communication channel and waits for *Bob* to perform the UPDATE operation on the record *R*. When *Bob* sends the UT_R, $R_{updated}^{K_{R\_read}}$, $UT_{R_{updated}}$ (see details in the *"CRUD implementation"* section above), *Eve* forges an encrypted record: $R_{updated}^{K_{R\_read}} \to R_{forged}^{K_{R\_read}}$. The *Data store* compares the UT_R received from *Bob* with the stored UT_R (which are equal because *Bob* has legal UPDATE permission) and authorizes UPDATE operation with forged (by *Eve*) updated record $R_{forged}^{K_{R\_read}}$.

Consequences:
An unauthorized UPDATE of a single record by a malicious client who has READ permissions to that record.

4.5.2 Attack on the Keystore (unauthorised rescinding of UPDATE permissions)

Conditions:

1. A malicious client with READ permissions (*Eve*);
2. A failure (accidental or intentional) in the normal operation of the *Keystore* (violation of *Assumption 4*).

Description:
If *Eve* manages to cause an intentional (or accidental) failure in the correct functioning of the *Keystore* (such that the *Keystore* contains a copy of a wrapped UPDATE ACK incorrectly assigned to Eve), *Eve* will be able to use it to rescind the READ/UPDATE permissions from the legal authorized clients or/and grant the permission to rescind the READ/UPDATE permissions from the legal authorized clients to other clients who possess READ permissions.

This attack may take place because a minimal implementation of the *Keystore* uses the simple existence of a READ or UPDATE ACK to validate that a client has these permissions when granting them to another client. Similarly, the *Keystore* cannot verify if the ACK acquired through a grant READ/UPDATE is valid. This presents opportunities for attacks whereby a client who gains an invalid ACK (or maliciously uses an existing one) can propagate invalid ACKs, replacing the existing valid ones.

Consequences:

A possibility of an unauthorised revocation of READ/UPDATE permissions from all the clients with READ/UPDATE access rights. The worst possible outcome of such attack is a denial of service by corruption of the access control policy (the *Data store* is not compromised in the course of this attack).

Mitigation:

Backup/recovery of the *Keystore*'s content, transactional logging, non-volatile writes (versioned writes). Additional mechanisms for further validation of the GRANT of access rights procedure is considered an area for further work. It is further discussed in [5].

5 Implementation considerations and further work

5.1 Implementing Hermes-based security tools

We believe that Hermes is a flexible and scalable scheme, which can be applied in a wide range of use-cases. In a real-world implementation of a Hermes-based security system, the location (remote or local) of the components may vary depending on the architecture and transport infrastructure of the protected information system. Wherein, the security properties (guarantees) of a protected system may expand with a help of an additional (organisational or technical) security methods.

We also strongly recommend reading the implementation-related document on Hermes [5] for better understanding of the way Hermes operates in an actual practical setting. In that document you can find a lot of useful information i.e. performance analysis, detailed specifications, and suggestions that can be useful for a practical introduction of Hermes into existing information systems.

5.2 Reference implementation

To provide a brief insight into the possible implementations of Hermes security system, a highly abstract reference implementation 'hermes-core' is released as open-source software [4].

6 Conclusion

Hermes is a novel practical cryptographic scheme intended to effectively solve real-life issues with common security requirements. Hermes provides end-to-end encryption between entities, cryptographic enforcement of access control policy distribution related to data compared with existing operational and algorithmic methods, processing sensitive data on the server side only in encrypted form.

Hermes can mitigate a wide range of threats and withstand full or partial compromise of its separate components, while still preserving the security of the sensitive data.

7 Acknowledgments

The authors would like to note that the shaping up and completion of the present scientific paper describing Hermes would be impossible without the individual contributions and collective work of the following persons:

- Anastasiia Voitova, *anastasi@cossacklabs.com*;
- Artem Storozhuk, *storojs72@gmail.com*;
- Dmytro Kornieiev, *dmitry@cossacklabs.com*;
- Karen Sawrey (aka Alona Ivanova), *karen@cossacklabs.com*.

8 References

[1] Growing Networks: Detours, Stunts and Spillovers / M. Aanestad, O. Hanseth / http://heim.ifi.uio.no/~oleha/Publications/iris24.pdf.
[2] Risk management guide for information technology systems / NIST Technical report SP 800-30 / http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-30.pdf.
[3] Internet of Things / F. Xia, L. T. Yang, L. Wang, A. Vinel / https://pdfs.semanticscholar.org/930c/4981e87584afa7e6f1f4977323e365aae097.pdf.
[4] Cossack Labs GitHub repository / https://github.com/cossacklabs/hermes-core.
[5] Cossack Labs website resources – Implementing Hermes-based Security Systems / https://www.cossacklabs.com/hermes/implementing-hermes-based-systems/.
[6] Cryptographic Enforcement of Information Flow Policies without Public Information / J. Crampton, N. Farley, G. Gutin, M. Jones, B. Poetterng / https://arxiv.org/pdf/1410.5567v2.pdf.
[7] Ciphertext-Policy Attribute-Based Encryption: An Expressive, Efficient, and Provably Secure Realization / B. Waters / https://eprint.iacr.org/2008/290.pdf.
[8] Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions / R. Curtmola, J. Garay, S. Kamara, R. Ostrovsky / https://eprint.iacr.org/2006/210.pdf.
[9] Private Information Retreival / B. Chor, O. Goldreich, E. Kushilevitz, M. Sudan / http://madhu.seas.harvard.edu/papers/1995/pir-journ.pdf.
[10] Fuzzy Identity-Based Encryption / A. Sahai, B. Waters / https://eprint.iacr.org/2004/086.pdf.
[11] A Guide to Fully Homomorphic Encryption / F. Armknecht, C. Boyd, C. Carr, K. Gjosteen, A. Jaschke, C. A. Reuter, M. Strand / https://eprint.iacr.org/2015/1192.pdf.
[12] The CRUD Security Matrix: A Technique for Documenting Access Rights / D. L. Lunsford, M. R. Collins / http://ocean.otr.usm.edu/~w300778/is-doctor/pubpdf/sc2008.pdf.
[13] DHIES: An encryption scheme based on Diffie-Hellman Problem / M. Abdalla, M. Bellare, P. Rogaway / http://web.cs.ucdavis.edu/~rogaway/papers/dhies.pdf.
[14] Announcing the ADVANCED ENCRYPTION STANDARD (AES) / FIPS publication 197 / https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf.
[15] The ChaCha family of stream ciphers / D. J. Bernstein / https://cr.yp.to/chacha.html.
[16] Authenticated Encryption in the Public-Key Setting: Security Notions and Analyses / J. H. An / https://eprint.iacr.org/2001/079.ps.
[17] Keying Hash Functions for Message Authentication / M. Bellare, R. Canetti, H. Krawczyk / https://cseweb.ucsd.edu/~mihir/papers/kmd5.pdf.

[18] Authenticated Diffie-Hellman Key Agreement Protocols / S. Blake-Wilson, A. Menezes / https://link.springer.com/content/pdf/10.1007/3-540-48892-8_26.pdf.
[19] Public Key Infrastructure Implementation and Design / S. Choudhury, K. Bhatnagar, W. Haque / https://www.e-reading.club/bookreader.php/142115/Choudhury_-_Public_Key_Infrastructure_implementation_and_design.pdf.
[20] Traitor Tracing with Constant Size Ciphertext / D. Boneh, M. Naor / http://crypto.stanford.edu/~dabo/papers/const-tt.pdf.
[21] Why Your Encrypted Database Is Not Secure / P. Grubbs, T. Ristenpart, V. Shmatikov / https://eprint.iacr.org/2017/468.pdf.