Universally Composable Key-Management (full version)

Steve Kremer[1], Robert Künnemann[2], and Graham Steel[2]

[1] LORIA & INRIA Nancy – Grand-Est, France
[2] INRIA Paris – Rocquencourt, France

Abstract. We present the first universally composable key-management functionality, formalized in the GNUC framework by Hofheinz and Shoup. It allows the enforcement of a wide range of security policies and can be extended by diverse key usage operations with no need to repeat the security proof. We illustrate its use by proving an implementation of a security token secure with respect to arbitrary key-usage operations and explore a proof technique that allows the storage of cryptographic keys externally, a novel development in simulation-based security frameworks.

Table of Contents

1 Introduction

Security critical applications often store keys on dedicated hardware security modules (HSM) or key-management servers to separate highly sensitive cryptographic operations from more vulnerable parts of the network. Access to such devices is given to protocol parties by the means of *Security APIs*. Examples of such APIs are the RSA PKCS#11 standard [1], IBM's CCA [2] and the trusted platform module (TPM) [3] API. Building on the work of Longley and Rigby [4] and Bond and Anderson [5] on API attacks, several recent papers have investigated the security of APIs on the logical level adapting symbolic techniques for protocol analysis [6–8], finding many new attacks. More recent work has tried to define appropriate security notions for APIs in terms of cryptographic games [9, 10]. This approach has two major disadvantages: first, it is not clear how the security notion will compose with other protocols implemented by the API. Second, it is difficult to see whether a definition covers the attack model completely, since the game may be tailored to a specific API. Since security APIs are foremost used as building blocks in other protocols, composability is crucial. In this work, we adapt the more general approach to API security of Kremer et al. [10] to a framework that allows for composition.

Composability can be proven in frameworks for simulation-based security, such as GNUC [11], a deviation of the Universal Composability (UC) framework [12]. The requirements of a protocol are formalized by abstraction: an *ideal functionality* computes the protocol's inputs and outputs securely, while a 'secure' protocol is one that emulates the ideal functionality. Simulation-based security naturally models the composition of the API with other protocols, so that proofs of security can be performed in a modular fashion. We decided to use the GNUC model because it avoids shortcomings of the original UC framework which have been pointed out over the years. Moreover, the GNUC framework is well structured and well documented resulting in more rigorous and readable security proofs.

Contributions. We present, to the best of our knowledge, the first composable definition of secure key-management in the form of a key-management functionality $\mathcal{F}_{\mathrm{KM}}$. It assures that keys are transferred correctly from one Security token to another, that the global security policy is respected (even though the keys are distributed on several tokens) and that operations which use keys are computed correctly. The latter is achieved by describing operations unrelated to key-management by so-called key-usage functionalities. $\mathcal{F}_{\mathrm{KM}}$ is parametric in the policy and the set of key-usage functionalities, which can be arbitrary. This facilitates revision of security devices, because changes to operations that are not part of the key-management or the addition of new functions do not affect the emulation proof. In order to achieve this extensibility, we investigate what exactly a "key" means in simulation-based security. Common functionalities in such settings do not allow two parties to share the same key, in fact, they do not have a concept of keys, but a concept of "the owner of a functionality" instead. The actual key is kept in the internal state of a functionality, used for computation, but never output. Dealing with key-management, we need the capability to export and import keys and we propose an abstraction of the concept of keys, that we call *credentials*. The owner of a credential can not only compute a cryptographic operation, but he can also delegate this capacity by transmitting the credential. We think this concept is of independent interest, and as a further contribution, subsequently introduce a general proof method that allows the substitution of credentials by actual keys when instantiating a functionality. Some aspects of the ideal functionality $\mathcal{F}_{\mathrm{crypto}}$ by Küsters et al. [13] are similar to our key-management functionality in that they both provide cryptographic primitives to a number of users and enjoy composability. However, the $\mathcal{F}_{\mathrm{crypto}}$ approach aims at abstracting a specified set of cryptographic operations on client machines to make the analysis of protocols in the simulation-based security models easier, and addresses neither key-management nor policies.

Limitations. Our key-management functionality is tightly coupled with the employment of a deterministic, symmetric authenticated encryption scheme that is secure against key-dependant messages for key export and import. While deterministic, symmetric authenticated encryption is indeed typically used to transfer keys (see, e. g., RFC 3394), it restricts the analysis to security devices providing this kind of encryption. We have not yet covered asymmetric encryption of keys in $\mathcal{F}_{\mathrm{KM}}$ (but we cover asymmetric encryption of user-supplied data), although $\mathcal{F}_{\mathrm{KM}}$ could be extended to support this. Second, adaptive corruption of parties, or of keys that produce an encryption, provokes the well-known commitment problem [14], so we place limitations on the types of corruptions that the environment may produce.

2 Background: GNUC

The GNUC ("GNUC is Not UC") framework was recently proposed by Hofheinz and Shoup [11] as an attempt to address several known shortcomings in UC. In particular, in UC the notion of a poly-time protocol implies that the interface of a protocol has to contain enough input padding to give sub-protocols of the implementation enough running time, hence the definition of an interface that is supposed to be abstract depends on the complexity of its implementation. Moreover, the proof of the composition theorem is flawed due to an inadequate formulation of the composition operation [11], though here the authors remark that, "none of the objections we raise point to gaps in security proofs of existing protocols. Rather, they seem artifacts of the concrete technical formulation of the underlying framework".These shortcomings are also addressed to a greater or lesser extent by other altenative frameworks [15–17]: we chose GNUC because it is similar in spirit to the original UC yet rigorous and well documented. We now give a short introduction to GNUC and refer the reader to [11] for additional details.

2.1 Preliminaries

Let Σ be some fixed, finite alphabet of symbols. We note η the security parameter.

Definition 1 (probabilistic polynomial-time). *We say that a probabilistic program A runs in polynomial-time, if the probability that A's runtime on an input of length n is bounded by a polynomial in n is 1. If so, we say such a program is PPT.*

Definition 2 (Computationally indistinguishable). *Let $X := \{X_\eta\}_\eta$ and $Y := \{Y_\eta\}_\eta$ be two families of random variables, where each random variable takes values in a set $\Sigma^* \cup \{\bot\}$. We say that X and Y are* computationally

indistinguishable*, written $X \approx Y$, if for every PPT program D that takes as input a string over Σ we have that*

$$|\Pr[D(x) = 1|\ x \leftarrow X_\eta] - \Pr[D(y) = 1|\ y \leftarrow Y_\eta]|$$

is negligible in η.

2.2 Machines and interaction

In GNUC a protocol π is modeled as a library of programs, that is, a function from protocol names to code. This code will be executed by interactive Turing machines. There are two distinguished machines, the environment and the adversary, that π does not define code for. All other machines are called *protocol machines*. Protocol machines can be divided into two subclasses: *regular* and *ideal*protocol machines. They come to life when they are called by the environment and are addressed using machine ids. A machine id <pid,sid> contains two parts: the party id pid, which is of the form <reg,*basePID*> for regular protocol machines and <ideal> for ideal protocol machines, and the session id sid. Session ids are structured as pathnames of the form $< \alpha_1, \ldots, \alpha_k >$. The last component α_k must be of the particular form $protName, sp$. When the environment sends the first message to a protocol machine, a machine running the code defined by the protocol name $protName$ is created. The code will often make decisions based on the session parameter sp and the party id. A machine M, identified by its machine id $< pid, < \alpha_1, \ldots, \alpha_k >>$, can call a subroutine, i.e., a machine with the machine id $< pid, < \alpha_1, \ldots, \alpha_k, \alpha_{k+1} >>$. We then say that M is the *caller* with respect to this machine. Two protocol machines, regular or ideal, are *peers* if they have the same session id. Programs have to declare which other programs they will call as subroutines, defining a static call graph which must be acyclic and have a program r with in-degree 0 – then we say that the protocol is rooted at r.

GNUC imposes the following communication constraints on a regular protocol machine M: it can only send messages to the adversary, to its ideal peer (i. e., a machine with party id <ideal> and the same session id), its subroutines and its caller. As a consequence, regular protocol machines cannot talk directly to regular peers . They can communicate via the adversary, which models an insecure network, or via the ideal peer. This ideal peer is a party that can communicate directly with all regular protocol parties and the adversary.

The code of the machines is described by a sequence of steps similarly to [11, § 12]. Each step is defined by a block of the form

```
name [conditions]: P
```

name is the label identifying the step. The logical expression [conditions] is a *guard* that must be satisfied to trigger a step. We omit the guard when it is true. A step name in the guard expression evaluates to true if the corresponding step has been triggered at some previous point. P is the code (whose semantics we expect to be clear) to be executed whenever the guard evaluates to true. In particular P may contain *accept-clauses* that describe the form of the message that can be input. The accept clause, too, might have logical conditions that must be satisfied in order to continue the execution of the step. Any message not triggering any step is processed by sending an error message to A.

2.3 Defining security via ideal functionalities

As in other universal composability frameworks, the security of a protocol is specified by a so-called *ideal functionality*, which acts as a third party and is trusted by all participants. Formally, an ideal functionality is a protocol that defines just one protocol name, say r. The behavior defined for this protocol name depends on the type of machine: all regular protocol machines act as "dummy parties" and forward messages received by their caller (which might be the environment) to their ideal peer. The ideal protocol machine interacts with the regular parties and the adversary: using the inputs of the parties, the ideal functionality defines a secure way of computing anything the protocol shall compute, explicitly computing the data that is allowed to leak to the attacker.

Example 1. For instance, an authenticated channel is specified as a functionality that takes a message from Alice and sends it to the attacker, exposing its content to the network, but only accepting a message from the attacker (the network) if it is the same message Alice sent in the first place. We will later come back to the following formulation, which is very similar to the one presented in [11, §12.1.1]:

```
ready−sender: accept <ready> from P;
   send <sender-ready> to A
ready−receiver[¬ready-sender]: accept <ready> from Q;
   send <ready-receiver-early> to A
ready−receiver[ready-sender]: accept <ready> from Q;
   send <receiver-ready> to A;
send [ready-receiver]: accept <send, x> from P;
   x̄ ← x; send <send, x> to A
done [send]: accept <done> from A;
   send <done> to P
deliver [send]: accept <deliver, x> from A where x = x;
   send <deliver, x̄> to Q
```

Listing 1: $\mathcal{F}_{ach}$ with session parameters $< P_{pid}, Q_{pid}, label >$. Note that in this example, every step can only be executed once.

We see that a functionality is completely defined by the code run on the ideal protocol machine.

Now we can define a second protocol, which is rooted at r, and does not necessarily define any behaviour for the ideal party, but for the regular protocol machines. The role of the environment Z is to distinguish whether it is interacting with the ideal system (dummy users interacting with an ideal functionality) or the real system (users executing a protocol). We say that a protocol π *emulates* a functionality $\mathcal{F}$ if for all attackers interacting with π, there exists an attacker, the simulator Sim, interacting with $\mathcal{F}$, such that no environment can distinguish between interacting with the attacker and the real protocol π, or the simulation of this attack (generated by Sim) and $\mathcal{F}$. It is actually not necessary to quantify over all possible adversaries: the most powerful adversary is the so-called dummy attacker A_D that merely acts as a relay forwarding all messages between the environment and the protocol [11, Theorem 5].

Let Z be a program defining an environment, i. e., a program that satisfies the communication constraints that apply to the environment (e. g., it sends messages only to regular protocol machines or to the adversary). Let A be a program that satisfies the constraints that apply to the adversary (e. g., it sends messages only to protocol machines (ideal or regular) it previously received a message from). The protocol π together with A and Z defines a structured system of interactive Turing machines (formally defined in [11, § 4]) denoted $[\pi, A, Z]$. The execution of the system on external input 1^η is a randomized process that terminates if Z decides to stop running the protocol and output a string in Σ^*. The random variable $\mathrm{Exec}[\pi, A, Z](\eta)$ describes the output of Z at the end of this process (or $\mathrm{Exec}[\pi, A, Z](\eta) = \perp$ if it does not terminate). Let $\mathrm{Exec}[\pi, A, Z]$ denote the family of random variables $\{\mathrm{Exec}[\pi, A, Z](\eta)\}_{\eta=1}^{\infty}$. An environment Z is well-behaved if the data-flow from Z to the regular protocol participants and the adversary is limited by a polynomial in the security parameter η. We say that Z *is rooted at* r, if it only invokes machines with the same session identifier referring to the protocol name r. We do not define the notion of a *poly-time protocol* and a bounded adversary here due to space constraints and refer the reader to the definition in [11, § 6].

Definition 3 (emulation w.r.t. the dummy adversary). *Let π and π' be poly-time protocols rooted at r. We say that π' emulates π if there exists an adversary Sim that is bounded for π, such that for every well-behaved environment Z rooted at r, we have*

$$\mathrm{Exec}[\pi, Sim, Z] \approx \mathrm{Exec}[\pi', A_D, Z].$$

where $\approx$ is the usual notion of computational indistinguishability.

3 An ideal key management functionality and its implementation

In this section we motivate and define our ideal functionality for key management. We explain first its architecture, then our concept of key usage functionalities which cover all the usual cryptographic operations we might want to perform with our managed keys. We then describe our notion of security policies for key management, and finally give an implementation of such a functionality.

3.1 Architecture

An ideal functionality should provide the required operations in a way that makes security obvious. This means its design must be as simple as possible in order for this security to be clear. However, there are subtle issues in such designs: obtaining a satisfactory formulation of digital signature took years because of repeated revisions caused by subtle flaws making the functionality unrealizable. The functionality we define will at some point need to preserve authenticity in a similar way to this signature functionality, but in a multi-session setting. So we must expect a key-management functionality to be *at least* as complex. Nonetheless we aim to keep it as simple as possible, and so justify the inclusion of each feature by discussing what minimum functionality we expect from a key-management system.

Policies. The goal of key-management is to preserve some kind of *policy* on a global level. Our policies express two kinds of requirements: usage policies of the form "key A can only be used for tasks X and Y", and dependency policies of the form "the security of key A may depend on the security of keys B and C". The need for the first is obvious. The need for the second arises because almost all non-trivial key management systems allow keys to encrypt other keys, or derive keys by, e. g., encrypting an identifier with a master key. Typically, the policy defines *roles* for keys, i. e., groups of tasks that can be performed by a key, and *security levels*, which define a hierarchy between keys.The difficulty lies in enforcing this policy globally when key-management involves a number of distributed security tokens that can communicate only via an untrusted network. Our ideal key-management functionality considers a distributed set of security tokens as a single trusted third party. It makes sure that every use of a key is compliant with the (global) policy. Therefore, if a set of well-designed security tokens with a sound local policy emulates the ideal key-management functionality, they can never reach a state where a key is used for an operation that is contrary to the policy. This implies that, in general, the key should be kept secret from the user, as the user cannot be forced to comply with the policy. Thus, keys are only accessed via an interface that executes only operations on the key permitted by the policy. The functionality associates some meta-data, an *attribute*, to each key. This attribute defines the key's role, and thus its uses. Existing industrial standards [1] and recent academic proposals [9, 10] are similar in this respect.

Sharing Secrets. A key created on one security token is *a priori* only available to users that have access to this token (since it is hidden from the user). Many cryptographic protocols require that the participants share some key, so in order to be able to run a protocol between two users of different security tokens, we need to be able to "transfer" keys between devices without revealing them. There are several ways to do this but we will opt for the simplest, key-wrapping (the encryption of one key by another). While it is possible to define key-management with a more conceptual view of "transferring keys" and allow the implementation to decide for an option, we think that since key-wrapping is relevant in practice (it is defined in RFC 3394), the choice for this option allows us to define the key-management in a more comprehensible way. We leave the definition of a notion more general in this regard for future work.

Secure Setup. The use of key-wrapping requires some initial shared secret values to be available before keys can be transferred. We model the setup in the following way: a subset of users, $Room$, is assumed to be in a secure environment during a limited setup-phase. Afterwards, the only secure channel is between a user U_i, and his security token ST_i. The intruder can access all other channels, and corrupt any party at any time, as well as corrupt keys, i. e., learn the value of the key stored inside the security token. This models the real world situation where tokens can be initialised securely but then may be lost or subject to, e. g., side channel attacks once deployed in the field.

Operations required. These requirements give a set of operations that key-management demands: `new` (create keys), `attr_change` (alter their attributes), `wrap` and `unwrap` (our chosen method of transferring keys), `corrupt` (corruption of keys) and `share`/`finish_setup` (modelling a setup phase in a secure environment).We argue that a reasonable definition of secure key-management has to provide at least those operations. Furthermore, the users need a way to access the keys stored in the security tokens, so there is a set of operations for each type of key. A signature key, for example, allows the operations *sign* and *verify*. This allows the following classification: the first group of operations defines *key-management*, the second *key-usage*. While key-management operations, for example `wrap`, might operate on two keys of possibly different types, key-usage operations are restricted to calling an operation on a single key and user-supplied data. This is coherent with global policies as mentioned above: the form "key A can be used for task X" expresses key-usage, the form "the security of key A depends on keys B and C" expresses a constraint on the key-management.

3.2 Key-usage (KU) functionalities

We now define an abstract notion of a functionality making use of a key which we call a key usage (KU) functionality. This will allow us to define our ideal key management functionality $\mathcal{F}_{\mathrm{KM}}$ in a way that is general with respect to a large class of cryptographic functionalities.For every KU operation, $\mathcal{F}_{\mathrm{KM}}$ calls the corresponding KU functionality, receives the response and outputs it to the user. We define $\mathcal{F}_{\mathrm{KM}}$ for arbitrary KU operations, and consider a security token secure, with respect to the implemented KU functionalities, if it emulates the ideal functionality $\mathcal{F}_{\mathrm{KM}}$ parametrized by those KU functionalities. This allows us to provide an implementation for secure key-management independent of which KU functionalities are used.

Credentials. Many existing functionalities, e. g., [12], bind the roles of the parties, e. g., signer and verifier, to a machine IDencoded in the session parameters. In implementations, however, the privilege to perform an operation is linked to the knowledge of a key rather than a machine ID. While for most applications this is not really a restriction, it is for *key*-management. The privilege to perform an operation of a KU functionality must be transferable as some piece of information, which however cannot be the actual key: a signing functionality, for example, that exposes its keys to the environment is not realizable, since the environment could then generate dishonest signatures itself. Our solution is to generate a key, but only send out a *credential*, which is a hard-to-guess pointer that refers to this key. We actually use the key generation algorithm to generate credentials.

Our approach imposes assumptions on the KU functionalities, as they need to be implementable in a key-manageable way.

Definition 4 (key-manageable implementation). *A key-manageable implementation $\hat{I}$ is defined by (i) a set of commands Cmds that can be partitioned into private and public commands, as well as key-generation, i. e., $\mathcal{C} = \mathcal{C}^{priv} \uplus \mathcal{C}^{pub} \uplus \{\mathtt{new}\}$, and (ii) a set of PPT algorithms implementing those commands, $\{impl_C\}_{C\in\mathcal{C}}$, such that for the key-generation algorithm $impl_{\mathtt{new}}$ it holds that*

- *for all k, $\Pr[k' = k | (k', p) \leftarrow impl_{\mathtt{new}}(1^\eta)]$ is negligible in η, and,*
- *$\Pr[|k_1| \neq |k_2| | (k_1, p_1) \leftarrow impl_{\mathtt{new}}(1^\eta); (k_2, p_2) \leftarrow impl_{\mathtt{new}}(1^\eta)]$ is negligible in η.*

$\hat{I}$ is a protocol in the sense of [11, §5], i. e., a run-time library that defines only one protocol name. The session parameter encodes a machine id P. When called on this machine, the code below is executed. If called on any other machine no message is accepted. From now on in our code we follow the convention that the response to a query $(\mathrm{Command}, \mathtt{sid}, \ldots)$ *is always of the form* $(\mathrm{Command}^\bullet, \mathtt{sid}, \ldots)$*, or* $\perp$.

`new`: accept <`new`> from parentId;
  $(key, public) \leftarrow impl_{\mathtt{new}}(1^\eta)$; $(credential,$ <`ignore`>$) \leftarrow impl_{\mathtt{new}}(1^\eta)$;
  $L \leftarrow L \cup \{(credential, key)\}$; send <$\mathtt{new}^\bullet, credential, public$> to parentId
`command`: accept <C, $credential$, m> from parentId;
  if $(credential, key) \in L$ for some key send <$C^\bullet, impl_C(key, m)$> to parentId
`public_command`: accept <C, $public$, m> from parentId;
  send <$C^\bullet, impl_C(public, m)$> to parentId
`corrupt`: accept <$corrupt$, $credential$> from parentId;
  if $(credential, key) \in L$ for some key send <`corrupt`$^\bullet$, key> to parentId
`inject`: accept <`inject`, k> from parentId;
  $(c,$ <`ignore`>$) \leftarrow impl_{\mathtt{new}}(1^\eta)$; $L \leftarrow L \cup \{(c, k)\}$; send <`inject`$^\bullet$, c> to parentId

The definition requires that each command C can be implemented by an algorithm $impl_C$. If C is private $impl_C$ takes the key as an argument. Otherwise it only takes public data (typically the public part of some key, and some user data) as arguments. In other words, an implementation $\hat{I}$ emulating $\mathcal{F}$ is, once a key is created, stateless w.r.t. queries concerning this key.

Definition 5 (key-manageable functionality). *A poly-time functionality $\mathcal{F}$ (to be precise, an ideal protocol [11, § 8.2]) is* key-manageable *iff it is poly-time, and there is a set of commands $\mathcal{C}$ and implementations, i. e., PPT algorithms $\mathtt{Impl}_{\mathcal{F}} = \{impl_C\}_{C\in\mathcal{C}}$, defining a key-manageable implementation $\hat{I}$ (also poly-time) which emulates $\mathcal{F}$.*

3.3 Policies

Since all credentials on different security tokens in the network are abstracted to a central storage, $\mathcal{F}_{\mathrm{KM}}$ can implement a global policy. Every credential in $\mathcal{F}_{\mathrm{KM}}$ is associated to an attribute from a set of attributes A and to the KU functionality it belongs to (which we will call its type). Keys that are used for key-wrapping are marked with the type `KW`.

Definition 6 (Policy). *Given the KU functionalities $\mathcal{F}_i$, $i \in \{1, \ldots, l\}$ and corresponding sets of commands $\mathcal{C}_i$, a* policy *is a quaternary relation* $\Pi \subset \{\mathcal{F}_1, \ldots, \mathcal{F}_l, \mathtt{KW}\} \times \cup_{i\in\{1,\ldots,l\}} \mathcal{C}_i^{priv} \cup \{\mathtt{new}, \mathtt{wrap}, \mathtt{unwrap}, \mathtt{attribute\_change}\} \times A \times A$.

$\mathcal{F}_{\mathrm{KM}}$ is parametrized by a policy Π. If $(\mathcal{F}, C, a, a') \in \Pi$ and if

- $C = \mathtt{new}$, then $\mathcal{F}_{\mathrm{KM}}$ allows the creation of a new key for the functionality $\mathcal{F}$ with attribute a.
- $\mathcal{F} = \mathcal{F}_i$ and $C \in \mathcal{C}^{priv}$, then $\mathcal{F}_{\mathrm{KM}}$ will permit sending the command C to $\mathcal{F}$, if the key is of type $\mathcal{F}$ and has the attribute a.
- $\mathcal{F} = \mathtt{KW}$ and $C = \mathtt{wrap}$, then $\mathcal{F}_{\mathrm{KM}}$ allows the wrapping of a key with attribute a' using a wrapping key with attribute a.
- $\mathcal{F} = \mathtt{KW}$ and $C = \mathtt{unwrap}$, then $\mathcal{F}_{\mathrm{KM}}$ allows to unwrapping a wrap with attribute a' using a wrapping key with attribute a.
- if $C = \mathtt{attribute\_change}$, then $\mathcal{F}_{\mathrm{KM}}$ allows the changing of a key's attribute from a to a'.

Note that a' is only relevant for the commands `wrap`, `unwrap` and `attribute_change`. Because of the last command, a key can have different attributes set for different users of $\mathcal{F}_{\mathrm{KM}}$, corresponding to different security tokens in the real word.

Example 2. To illustrate the definition of policy consider the case of a single KU functionality for encryption $\mathcal{F}_{\mathrm{enc}}$. The set of attributes A is $\{0, 1\}$: intuitively a key with attribute 1 is allowed for wrapping and a key with attribute 0 for encryption. The following table describes a policy that allows wrapping keys to wrap encryption keys, but not other wrapping keys, and allows encryption keys to perform encryption on user-data, but nothing else – even decryption is disallowed. The policy Π consists of the following 4-tuples ($\mathcal{F}$,Cmd,attr_1,attr_2) defined in Figure 1.

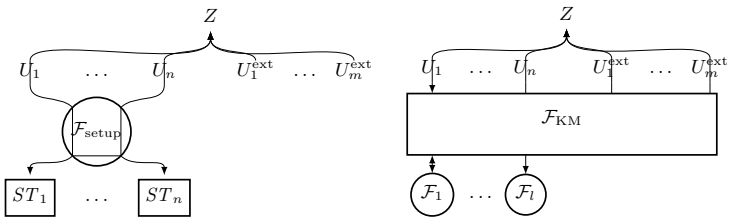
$\mathcal{F}$	Cmd	attr_1	attr_2
KW	new	1	*
$\mathcal{F}_{\mathrm{enc}}$	new	0	*
KW	attribute_change	1	1
$\mathcal{F}_{\mathrm{enc}}$	attribute_change	0	0
KW	wrap	1	0
KW	unwrap	1	0
$\mathcal{F}_{\mathrm{enc}}$	enc	0	*

Fig. 1: Security policy

3.4 The key-management functionality and reference implementation

We are now in a position to give a full definition of $\mathcal{F}_{\mathrm{KM}}$ together with an implementation. We give a description of $\mathcal{F}_{\mathrm{KM}}$ in the Listings 3 to 8. At the same time, to illustrate our definition and demonstrate its use, we present the implementation of a Security API showing that it is possible to implement a Security API for key-management that is independent of the KU functions it provides. For book-keeping purposes $\mathcal{F}_{\mathrm{KM}}$ maintains a set $\mathcal{K}_{\mathrm{cor}}$ of corrupted keys and a wrapping graph $\mathcal{W}$ whose vertices are the credentials. An edge (c_1, c_2) is created whenever (the key corresponding to) c_1 is used to wrap (the key corresponding to) c_2. In Section 4, we show that this implementation is a realization of $\mathcal{F}_{\mathrm{KM}}$. We emphasize that extending $\mathcal{F}_{\mathrm{KM}}$ and the implementation by a new KU functionality does not require a new proof.

Structure. $\mathcal{F}_{\mathrm{KM}}$ acts as a proxy service to the KU functionalities. It is possible to create keys, which means that $\mathcal{F}_{\mathrm{KM}}$ asks the KU functionality for the credentials and stores them, but outputs only a *handle* referring to the key. This handle can be the position of the key in memory, or a running number – we just assume that there is a way to draw them such that they are unique. When a command $C \in \mathcal{C}_i^{priv}$ is called with a handle and a message, $\mathcal{F}_{\mathrm{KM}}$ substitutes the handle with the associated credential, and forwards the output to $\mathcal{F}_i$. The response from $\mathcal{F}_i$ is forwarded unaltered. All queries are checked against the policy. The environment may corrupt parties connected to security tokens, as well as individual keys.



(a) Distributed security tokens in the network

(b) An idealized functionality $\mathcal{F}_{\text{KM}}$ in the same network

Fig. 2: Distributed security tokens in the network (left-hand side) and idealized functionality $\mathcal{F}_{\text{KM}}$ in the same network (right-hand side).

Definition 7 (Parameters to a security token network). *We summarize the parameters of a security token Network as two tuples, ($\mathcal{U}$,$\mathcal{U}^{\text{ext}}$, $\mathcal{ST}$, $Room$) and ($\overline{\mathcal{F}}$, $\overline{\mathcal{C}}$, Π). The first tuple defines* network parameters*:*

- $\mathcal{U} = \{U_1, \ldots, U_n\}$ *are the party IDs of the users connected to a security token*
- $\mathcal{U}^{\text{ext}} = \{U_1^{\text{ext}}, \ldots, U_m^{\text{ext}}\}$ *are the party IDs of external users, i. e., users that do not have access to a security token.*
- $\mathcal{ST} = \{ST_1, \ldots, ST_n\}$ *are the party IDs of the security tokens accessed by* $U_1, \ldots, U_n$.
- $Room \subset \mathcal{U}$.

The second tuple defines key-usage parameters*:*

- $\overline{\mathcal{F}} = \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$, *and*
- $\overline{\mathcal{C}} = \{\mathcal{C}_1, \ldots, \mathcal{C}_l\}$ *are key-manageable functionalities with corresponding sets of commands. Note that* $\texttt{KW} \notin \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$, *and that each* $\mathcal{C}_i \in \overline{\mathcal{C}}$ *is partitioned into the private* $\mathcal{C}_i^{priv}$ *and public commands* $\mathcal{C}_i^{pub}$, *as well as the singleton set consisting of* `new`.
- Π *is a policy for* $\overline{\mathcal{F}}$ *(cf. Definition 6) and a membership test on* Π *can be performed efficiently.*

Network setup. Figure 2 shows the network of distributed users and security tokens on the left, and their abstraction $\mathcal{F}_{\text{KM}}$ on the right. There are two kinds of users: $U_1, \ldots, U_n =: \mathcal{U}$, each of whom has access to exactly one security token ST_i, and external users $U_1^{\text{ext}}, \ldots, U_m^{\text{ext}} =: \mathcal{U}^{\text{ext}}$, who cannot access any security token. The security token ST_i can only be controlled via the user U_i. The functionality $\mathcal{F}_{\text{setup}}$ in the real world captures our setup assumptions, which need to be achieved using physical means. Among other things, $\mathcal{F}_{\text{setup}}$ assures a secure channel between each pair (U_i, ST_i). The necessity of this channel follows from the fact that *a*) GNUC forbids direct communication between two regular protocol machines (indirect communication via A is used to model an insecure channel) and *b*) $U_1, \ldots, U_n$ can be corrupted by the environment, while $ST_1, \ldots, ST_n$ are incorruptible.

The session id `sid` is of the form <α_1,...,α_{k-1},<`prot-fkm`, *sp*>>, where the session parameter *sp* is some encoding of the network parameters $\mathcal{U}$,$\mathcal{U}^{\text{ext}}$, $\mathcal{ST}$, $Room$. The code itself is parametric in the KU parameters $\overline{\mathcal{F}}$, $\overline{\mathcal{C}}$, Π. When we refer to $\mathcal{F}_{\text{KM}}$ as a network identity, we mean the machine id <`ideal`,`sid`>.

Similarly, each security token $ST_i \in \{ST_1, \ldots, ST_n\}$ is addressed via the machine id <ST_i,`sid`>. We will abuse notation by identifying the machine id with ST_i, whenever the session id is clear from the context. The session parameter within `sid` encodes the network parameters $\mathcal{U}$,$\mathcal{U}^{\text{ext}}$, $\mathcal{ST}$, $Room$. ST_i makes subroutine calls to the functionality $\mathcal{F}_{\text{setup}}$ which subsumes our setup assumptions. $\mathcal{F}_{\text{setup}}$ provides two things: 1. a secure channel between each pair U_i and ST_i, 2. a secure channel between some pairs ST_i and ST_j during the *setup phase* (see below). ST_i receives commands from a machine $U_i \in \mathcal{U}$, which is defined in Definition 10, and relays arbitrary commands sent by the environment via $\mathcal{F}_{\text{setup}}$. The environment cannot talk directly to ST_i, but the attacker can send queries on behalf of any corrupted user, given that the user has been corrupted previously (by the environment).

Setup phase. The setup is implemented by the functionality $\mathcal{F}_{\text{setup}}$, defined in Listing A.3 in Appendix A. All users in $Room$ are allowed to share keys during the setup phase, i. e., the implementation is allowed to use secure channels to transport keys during this phase, but not later.This secure channel *between two security tokens* ST is only

used during the setup phase. Once the setup phase is finished, the expression `setup_finished` evaluates to true and the functionality enters the run phase. During the run phase, $\mathcal{F}_{\text{setup}}$ provides only a secure channel between a user U_i, which takes commands from the environment, and his security token ST_i.When we say that ST_i calls $\mathcal{F}_{\text{setup}}$, we mean that it sends a message to the machine id $\langle ST_i, \langle \texttt{sid}, \langle \texttt{prot-fsetup}, \langle \mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}, \mathit{Room} \rangle \rangle \rangle \rangle$.

Implementation. The implementation ST is inspired by [10] and is parametric on the KU parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ and the implementation functions $\overline{\texttt{Impl}} := \{\texttt{Impl}_{\mathcal{F}}\}_{\mathcal{F} \in \overline{\mathcal{F}}}$. It is composable in the following sense: if a device performs the key-management according to our implementation, it does not matter how many, and which functionalities it enables access to, as long as those functionalities provide the amount of security the designer aims to achieve (cf. Corollary 1). In Section 5, we show how to instantiate those KU functionalities to fully instantiate a "secure" security token, and how $\mathcal{F}_{\text{KM}}$ facilitates analysis of this configuration.

From now on, when we say that $\mathcal{F}_{\text{KM}}$ *calls* $\mathcal{F}$, we mean that it sends a message to a regular peer that calls $\mathcal{F}$ as a sub-protocol and relays the answers. Formally, $\mathcal{F}_{\text{KM}}$ sends a message to the machine id $F = \langle\langle \texttt{reg}, \mathcal{F} \rangle, \texttt{sid} \rangle$, who in turn addresses $\langle\langle \texttt{reg}, \mathcal{F} \rangle, \langle \texttt{sid}, \langle \mathcal{F}, F \rangle \rangle \rangle$ as a dummy party. This is necessary since Condition C6 in [11, §4.5] disallows ideal parties from making sub-routine calls. Note first that, for unambiguity, we use the code $\mathcal{F}$ as the party id for this user. Note secondly that $\mathcal{F}$ uses the session parameter F to identify F as the only machine id it accepts messages from.

Executing commands in $\mathcal{C}^{priv}$. If the policy Π permits execution of a command $C \in \mathcal{C}^{priv}$, $\mathcal{F}_{\text{KM}}$ calls the corresponding functionality as a sub-protocol, substituting the handle by the corresponding credential. Similarly, ST_i uses the corresponding key to compute the output of the implementation function $impl_C$ of the command C (Listings 3.4 and 2).

`command[finish_setup]`: accept $\langle C \in \mathcal{C}_i^{priv}, h, m \rangle$ from $U \in \mathcal{U}$;
if Store $[U, h] = \langle \mathcal{F}_i, a, c \rangle$ and $\langle \mathcal{F}_i, C, a, \star \rangle \in \Pi$ and $\mathcal{F}_i \neq \texttt{KW}$
  call $\mathcal{F}_i$ with $\langle C, c, m \rangle$; accept $\langle C^{\bullet}, r \rangle$ from $\mathcal{F}_i$; send $\langle C^{\bullet}, r \rangle$ to U

`command[finish_setup]`: accept $\langle C \in \mathcal{C}_{i'}^{priv}, h, m \rangle$ from $\mathcal{F}_{\text{setup}}$;
if Store $[U_i, h] = \langle \mathcal{F}_{i'}, a, k \rangle$ and $\langle \mathcal{F}_{i'}, C, a, \star \rangle \in \Pi$ and $\mathcal{F}_{i'} \neq \texttt{KW}$
  send $\langle C^{\bullet}, impl_C(k, m) \rangle$ to $\mathcal{F}_{\text{setup}}$

Listing 2: Executing command C on a handle h with data m ($\mathcal{F}_{\text{KM}}$ above, ST_i below).

Creating keys. A user can create keys of type $\mathcal{F}$ and attribute a using the command $\langle \texttt{new}, \mathcal{F}, a \rangle$. In $\mathcal{F}_{\text{KM}}$, the functionality $\mathcal{F}$ is asked for a new credential and some public information. The credential is stored with the meta-data at a freshly chosen position h in the store. Similarly, ST stores an actual key, instead of a credential. Both $\mathcal{F}_{\text{KM}}$ and ST output the handle h and the public information given by $\mathcal{F}$, or produced by the key-generation algorithm. $\mathcal{F}_{\text{KM}}$ treats wrapping keys differently: it calls the key-generation function for `KW`. It is possible to change the attributes of a key in future, if the policy permits (Listing 6).

`new[ready]`: accept $\langle \texttt{new}, F, a \rangle$ from $U \in \mathcal{U}$;
if $\langle F, \texttt{new}, a, \star \rangle \in \Pi$
  if $F = \texttt{KW}$ then $(c, \mathit{public}) \leftarrow impl_{\texttt{new}}^{KW}(1^{\eta})$
  else call F with $\langle \texttt{new} \rangle$; accept $\langle \texttt{new}^{\bullet}, c, \mathit{public} \rangle$ from F
  if $c \in \mathcal{K} \cup \mathcal{K}_{\text{cor}}$ then send $\langle \texttt{error} \rangle$ to A
  else create h; Store $[U, h] \leftarrow \langle \texttt{F}, \texttt{a}, \texttt{c} \rangle$; $\mathcal{K} := \mathcal{K} \cup \{c\}$; send $\langle \texttt{new}^{\bullet}, h, \mathit{public} \rangle$ to U

`new[ready]`: accept $\langle \texttt{new}, F, a \rangle$ from $\mathcal{F}_{\text{setup}}$;
  if $\langle F, \texttt{new}, a, \star \rangle \in \Pi$
    $(k, \mathit{public}) \leftarrow impl_{\texttt{new}}^{F}(1^{\eta})$; create h; Store $[U_i, h] \leftarrow \langle F, a, k \rangle$;
    send $\langle \texttt{new}^{\bullet}, h, \mathit{public} \rangle$ to $\mathcal{F}_{\text{setup}}$

Listing 3: Creating keys of type $\mathcal{F}$, and attribute a ($\mathcal{F}_{\text{KM}}$ above, ST_i below).

Wrapping and Unwrapping. The commands that are important for key-management are handled by $\mathcal{F}_{\text{KM}}$ itself. To transfer a key from one security token to another in the real world, the environment instructs, for instance, U_1 to ask for a key to be *wrapped* (see Figure 4). A wrapping of a key is the encryption of a key with another key, the

wrapping key. The wrapping key must of course be on both security tokens prior to that. U_1 will receive the wrap from ST_1 and forward it to the environment, which in turn instructs U_2 to unwrap the data it just received from U_1. The implementation ST_i just verifies if the wrapping confirms the policy, and then produces a wrapping of c_2 under c_1, with additionally authenticated information: the type and the attribute of the key, plus a user-chosen identifier that is bound to a wrapping in order to identify which key was wrapped. This could, e. g., be a key digest provided by the KU functionality the key belongs to.

`wrap[finish_setup]`: accept <`wrap`, h_1, h_2, id> from $U \in \mathcal{U}$;
if Store`[`U, h_1`]`=<`KW`, a_1, c_1> and Store`[`U, h_2`]`=<F_2, a_2, c_2> and <`KW`, `wrap`, a_1, a_2>$\in \Pi$
  if $\exists w.$<c_2, <F_2, a_2, id>,w>$\in$encs`[`c_1`]`
    send <`wrap`$^\bullet$, w> to U
  else
    $\mathcal{W} \leftarrow \mathcal{W} \cup \{(c_1, c_2)\}$;
    if $c_1 \in \mathcal{K}_{\text{cor}}$
     for all c_3 reachable from c_2 in $\mathcal{W}$ `corrupt` c_3;
     $w \leftarrow \mathsf{wrap}^{<F_2, a_2, id>}(c_1,$ key`[`c_2`]`$)$
    else
      $w \leftarrow \mathsf{wrap}^{<F_2, a_2, id>}(c_1, \$^{|c_2|})$
    encs`[`c_1`]` $\leftarrow$ encs`[`c_1`]` $\cup$< c_2, <F_2, a_2, id>,w>; send <`wrap`$^\bullet$, w> to U

`wrap[finish_setup]`: accept <`wrap`, h_1, h_2, id> from $\mathcal{F}_{\text{setup}}$;
  if Store`[`U_i, h_1`]`=<`KW`, a_1, k_1> and Store`[`U_i, h_2`]`=<F_2, a_2, k_2>
      and <`KW`, `wrap`, a_1, a_2>$\in \Pi$
    $w \leftarrow \mathsf{wrap}^{<F_2, a_2, id>}(k_1, k_2)$; send <`wrap`$^\bullet$, w> to $\mathcal{F}_{\text{setup}}$

Listing 4: Wrapping key h_2 under key h_1 with additional information id ($\mathcal{F}_{\text{KM}}$ above, ST_i below).

When a wrapped key is unwrapped using an uncorrupted key, $\mathcal{F}_{\text{KM}}$ checks if the wrapping was produced before, using the same identifier. Furthermore, $\mathcal{F}_{\text{KM}}$ checks if the given attribute and types are correct. If this is the case, it creates another entry in `Store`, i. e., a new handle h' for the user U pointing to the *correct* credentials, type and attribute type of the key. This way, $\mathcal{F}_{\text{KM}}$ can guarantee the consistency of its database for uncorrupted keys, see the following Theorem 1. If the key used to unwrap is corrupted, this guarantee cannot be given, but the resulting entry in the store is marked corrupted. It is possible to inject keys by unwrapping a key that was wrapped *outside the device*. Such keys could be generated dishonestly by the adversary, that is, not using their respective key-generation function.

`unwrap[finish_setup]`: accept <`unwrap`, h_1, w, a_2, F_2, id> from $U \in \mathcal{U}$;
if Store`[U,`h_1`]`=<`KW`, a_1, c_1> and <`KW`, `unwrap`, a_1, a_2>$\in \Pi$,$F_2 \in \overline{\mathcal{F}}$
  if $c_1 \in \mathcal{K}_{\text{cor}}$
    $c_2 \leftarrow \mathsf{unwrap}^{<F_2, a_2, id>}(c_1, w)$;
    if $c_2 \neq \perp$ and $c_2 \notin \mathcal{K}$
      if $F_2 = KW$
        create h_2; Store`[`U, h_2`]` $\leftarrow$ <F_2, a_2, c_2>; key`[`c_2`]`=c_2; $\mathcal{K}_{\text{cor}} \leftarrow \mathcal{K}_{\text{cor}} \cup \{c_2\}$
      else
        call F_2 with <`inject`, c_2>; accept <`inject`$^\bullet$, c'>;
        if $c' \notin \mathcal{K} \cup \mathcal{K}_{\text{cor}}$
          create h_2;
          Store`[`U, h_2`]` $\leftarrow$ <F_2, a_2, c'>; key`[`c'`]`=c_2; $\mathcal{K}_{\text{cor}} \leftarrow \mathcal{K}_{\text{cor}} \cup \{c'\}$;
      send <`unwrap`$^\bullet$, h> to U
    else if $c_2 \neq \perp \wedge c_2 \in \mathcal{K} \wedge c_2 \in \mathcal{K}_{\text{cor}}$
      create h_2; Store`[`U, h_2`]` $\leftarrow$ <F_2, a_2, c_2>; send <`unwrap`$^\bullet$, h> to U
    else // $(c_2 = \perp \vee c_2 \in \mathcal{K} \setminus \mathcal{K}_{\text{cor}})$
      send <`error`> to A
  else if ($c_1 \notin \mathcal{K}_{\text{cor}}$ and $\exists! c_2.$<c_2, <F_2, a_2, id>,w>$\in$encs`[`c_1`]`)
    create h_2; Store`[`U, h_2`]` $\leftarrow$ <F_2, a_2, c_2>; send <`unwrap`$^\bullet$, h> to U

`unwrap[finish_setup]`: accept <`unwrap`, h_1, w, a_2, F_2, id> from $\mathcal{F}_{\text{setup}}$

if Store $[U_i, h_1]$=<KW, a_1, k_1> and $F_2 \in \overline{\mathcal{F}}$ and <KW, unwrap, a_1, a_2>$\in \Pi$
and $k_2 = \mathsf{unwrap}^{<F_2, a_2, id>}(k_1, w) \neq \perp$
create h_2; Store $[U, h_2] \leftarrow$ <F_2, a_2, k_2>; send <unwrap$^\bullet$, h> to $\mathcal{F}_{\text{setup}}$

Listing 5: Unwrapping w created with attribute a_2, F_2 and id using the key h_1. The symbol $\exists!$ in $\exists!x.p(x)$ means there is exactly one x such that $p(x)$ ($\mathcal{F}_{\text{KM}}$ above, ST_i below).

There is an improvement that became apparent during the proof of emulation (2 below). Namely, when unwrapping with a corrupted key, $\mathcal{F}_{\text{KM}}$ checks the attribute that is going to be assigned to the (imported) key against the policy, instead of just accepting that a corrupted wrapping-key might just import any wrapping the attacker generated. This prevents, for example, a corrupted wrapping-key of low security from *creating* a high-security wrapping-key by unwrapping dishonestly produced wrappings. This detail in the definition of $\mathcal{F}_{\text{KM}}$ enforces a stronger implementation than the one in [10]: ST validates the attribute given with a wrapping, enforcing that it is at least sound according to the policy, instead of blindly trusting the authenticity of the wrapping mechanism. Hence our implementation is more robust.

Changing attributes of keys . : The attributes associated with a key with handle h can be updated using the command <attr_change, h, a'>.

attr_change[finish_setup]: accept <attr_change, h, a'> from $U \in \mathcal{U}$;
if Store $[U, h]$=<F, a, c> and<F, attr_change, a, a'>$\in \Pi$
Store $[U, h]$=<F, a', c>; send <attr_change$^\bullet$> to U

attr_change[finish_setup]: accept <attr_change, h, a'> from $\mathcal{F}_{\text{setup}}$;
if Store $[U_i, h]$=<F, a, k> and <F, attr_change, a, a'>$\in \Pi$
Store $[U_i, h]$=<F, a', k>; send <attr_change$^\bullet$> to $\mathcal{F}_{\text{setup}}$

Listing 6: Changing the attribute of h to a' ($\mathcal{F}_{\text{KM}}$ above, ST_i below).

Corruption. Since keys might be used to wrap other keys, we would like to know how the loss of a key to the adversary affects the security of other keys. When an environment "corrupts a key" in $\mathcal{F}_{\text{KM}}$, the adversary learns the credentials to access the functionalities. Since corruption can occur indirectly, via the wrapping command, too, we factored this out into Listing 7. ST implements this corruption by outputting the actual key to the adversary.

procedure for corrupting a credential c:

$\mathcal{K}_{\text{cor}} \leftarrow \mathcal{K}_{\text{cor}} \cup \{c\}$
for any Store $[U, h]$=<F, a, c>
if $F =$ KW
key$[c] \leftarrow c$; send <corrupt$^\bullet$, h, c> to A
else
call F with <corrupt, c>; accept <corrupt$^\bullet$, k> from F
key$[c] \leftarrow k$; send <corrupt$^\bullet$, h, k> to A

Listing 7: Corruption procedure used in steps corrupt and wrap

corrupt[finish_setup]: accept <corrupt, h> from $U \in \mathcal{U}$;
if Store $[U, h]$=<F, a, c>
for all c' reachable from c in $\mathcal{W}$ corrupt c'

corrupt[finish_setup]: accept <corrupt, h> from $\mathcal{F}_{\text{setup}}$;
if Store $[U_i, h]$=<F, a, k> send <corrupt$^\bullet$, h, k> to A

Listing 8: Corrupting h ($\mathcal{F}_{\text{KM}}$ above, ST_i below).

Public key operations. Some cryptographic operations (e. g., digital signatures) allow users without access to a security token to perform certain operations (e. g., signature verification). Those commands do not require knowledge of the credential (in $\mathcal{F}_{\text{KM}}$), or the secret part of the key (in ST). They can be computed using publicly available

information. In the case where participants in a high-level protocol make use of, e. g., signature verification, but nothing else, the protocol can be implemented without requiring those parties to have their own security tokens. Note that $\mathcal{F}_{\mathrm{KM}}$ relays this call to the underlying KU functionality unaltered, and independent of its store and policy (see Figure 9).

```
public_command: accept <C, public, m> from U ∈ 𝒰 ∪ 𝒰^ext;
  if C ∈ C_{i,pub}
    call F_i with <C, public, m>; accept <C•, r> from F_i; send <C•, r> to U
```

Listing 9: Computing the public commands C using $public$ and m ($\mathcal{F}_{\mathrm{KM}}$, note that ST_i does not implement this step).

The implementation ST_i does not implement this step, since U_i, U_i^{ext} compute $impl_C(public, m)$ themselves.

Before we give the formal definition of $\mathcal{F}_{\mathrm{KM}}$, note that $\mathcal{F}_{\mathrm{KM}}$ is not an ideal protocol in the sense of [11, § 8.2], since not every regular protocol machine runs the dummy party protocol – the party <reg, $\mathcal{F}_i$> relays the communication with the KU functionalities.

Definition 8 **($\mathcal{F}_{\mathrm{KM}}$).** *Given the KU parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$, let the ideal protocols $\mathcal{F}_{p+1}, \ldots, \mathcal{F}_l$ be rooted at* prot-$\mathcal{F}_{p+1}$,..., prot-$\mathcal{F}_l$. *In addition to those protocols names, $\mathcal{F}_{\mathrm{KM}}$ defines the protocol name* prot-fkm. *For* prot-fkm, *the protocol defines the following behaviour: a regular protocol machine with machine id <<reg, $\mathcal{F}_i$>, sid> for $\mathcal{F}_i \in \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$ runs the following code:*

```
ready: accept <ready> from parentId
  send <ready> to <ideal, sid> (= F_KM)
relay_to: accept <m> from <ideal, sid> (= F_KM)
  send <m> to <<reg, F_i>,<sid, <prot-F_i, <>>> (= F_i)
relay_from: accept <m> from <<reg, F_i>,<sid, <prot-F_i, <>>>
  send <m> to <ideal, sid> (= F_KM)
```

The ideal party runs the logic for $\mathcal{F}_{\mathrm{KM}}$ described in Listings 3 to 8.

Remark 1: Credentials for different KU functionalities are distinct. It is nonetheless possible to encrypt and decrypt arbitrary credentials using <wrap> and <unwrap>. Suppose a designer wants to prove a Security API secure which uses shared keys for different operations. One way or another, she would need to prove that those roles do not interfere. For this case, we suggest providing a functionality that combines the two KU functionalities, and proving that the implementation of the two operations combined emulates the combined functionality. It is possible to assign different attributes to keys of the same KU functionality, and thus restrict their use to certain commands, effectively providing different roles for credentials to the same KU functionality. This can be done by specifying two attributes for the two roles and defining a policy that restricts which operation is permitted for a key of each attribute.

Remark 2: Many commonly used functionalities are not *caller-independent*, often the access to critical functions is restricted to a network party that is encoded in the session identifier. However, we think that it is possible to construct caller-independent functionalities for many functionalities, if the implementation relies on keys but is otherwise stateless. A general technique for transforming such functionalities into key-manageable functionalities that preserves existing proofs is work in progress.

Remark 3: Constraint C6 in [11, §8.2] requires each regular machine to send a message to $\mathcal{F}_{\mathrm{setup}}$ before it can address it. The initialization procedure and the parts of the definition of $\mathcal{F}_{\mathrm{KM}}$, ST and $\mathcal{F}_{\mathrm{setup}}$ that perform this procedure are explained in detail in Appendix A.

Properties. In order to identify some properties we get from the design of $\mathcal{F}_{\mathrm{KM}}$, we introduce the notion of an attribute policy graph:

Definition 9. *We define a family of* attribute policy graphs *$(\mathcal{A}_{\Pi,\mathcal{F}})$, one for each KU functionality $\mathcal{F}$ and one for key-wrapping (in which case $\mathcal{F} =$* KW*) as follows:*

- *a is a node in $\mathcal{A}_{\Pi,\mathcal{F}}$ if $(\mathcal{F}, C, a, a') \in \Pi$ for some C, a'.*
- *a is additionally marked* new *if $(\mathcal{F},$* new*$, a, a') \in \Pi$.*
- *An edge (a, a') is in $\mathcal{A}_{\Pi,\mathcal{F}}$ whenever $(\mathcal{F},$* attribute_change*$, a, a') \in \Pi$.*

Example 3. For the policy Π described in Example 2, the attribute policy graph $\mathcal{A}_{\Pi,\mathtt{KW}}$ contains one node 1 connected to itself and marked `new`. Similarly, the attribute policy graph $\mathcal{A}_{\Pi,\mathcal{F}_{\mathrm{enc}}}$ contains one node 0 connected to itself and marked `new`.

The following theorem shows that, first, the set of attributes an uncorrupted key can have in $\mathcal{F}_{\mathrm{KM}}$ is determined by the attribute policy graph, second, there are exactly three ways to corrupt a key, and third, KU-functionalities receive the `corrupt` message only if a key is corrupted. The formal proof of these claims can be found in Appendix C.

Theorem 1 (Properties of $\mathcal{F}_{\mathrm{KM}}$). *Every instance of $\mathcal{F}_{\mathrm{KM}}$ with parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ and session parameters $\mathcal{U}, \mathcal{U}^{\mathrm{ext}}$, $\mathcal{ST}$, Room has the following properties:*

(1) At any step of an execution of $[\mathcal{F}_{\mathrm{KM}}, A_D, Z]$, the following holds for $\mathcal{F}_{\mathrm{KM}}$: for all $\mathtt{Store}[U, h] =< \mathcal{F}, a, c >$ such that $c \notin \mathcal{K}_{\mathrm{cor}}$, there is a node a' marked `new` in the attribute policy graph $\mathcal{A}_{\Pi,\mathcal{F}}$ such that a is reachable from a' in $\mathcal{A}_{\Pi,\mathcal{F}}$ and there was a step `new` where $\mathtt{Store}[U', h'] =< \mathcal{F}, a', c >$ was added.

(2) At any step of an execution of $[\mathcal{F}_{\mathrm{KM}}, A_D, Z]$, the following holds for $\mathcal{F}_{\mathrm{KM}}$: all $c \in \mathcal{K}_{\mathrm{cor}}$ were either

- *(a)* directly corrupted: *there was a `corrupt` triggered by a query $<$ `corrupt`, `h` $>$ from U while `Store[U,h]`$=<$ $\mathcal{F}, a, c >$, or indirectly, that is,*
- *(b)* corrupted via wrapping: *there is $c' \in \mathcal{K}_{\mathrm{cor}}$ such that at some point the `wrap` step was triggered by a message `<wrap,`h'`,h,`id`>` from U while `Store[`U, h'`]`$=<$ `KW`$, a', c' >$, `Store[U,h]`$=< \mathcal{F}, a, c >$, or*
- *(c)* corrupted via unwrapping (injected): *there is $c' \in \mathcal{K}_{\mathrm{cor}}$ such that at some point the `unwrap` step was triggered by a message `<unwrap,`$h', w, a, F,$ id`>` from U while `Store[U,`h'`]`$=<$ `KW`$, a', c' >$ and $c = \mathsf{unwrap}_{c'}^{<F,a,id>}(w)$ for some a, F and id.*

(3) At any step of an execution of $[\mathcal{F}_{\mathrm{KM}}, A_D, Z]$, the following holds: whenever an ideal machine $\mathcal{F}_i =$ `<ideal,<sid,<`$\mathcal{F}_i$`,`F`>>>`, $F =$ `<<reg,`$\mathcal{F}$ `>,<sid>>`, accepts the message `<corrupt,`c`>` for some c such that $\mathcal{F}_{\mathrm{KM}}$ in session `sid` has an entry `Store[`U`,` h`]= <`$\mathcal{F}_i$`,`a`,`c`>`, then $c \in \mathcal{K}_{\mathrm{cor}}$ in $\mathcal{F}_{\mathrm{KM}}$.

4 Proof overview

We show that, for arbitrary KU parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$, the network $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$, consisting of the set of users $\mathcal{U}$ connected to security tokens $\mathcal{ST}$, the set of external users $\mathcal{U}^{\mathrm{ext}}$ and the functionality $\mathcal{F}_{\mathrm{setup}}$, emulates the key-management functionality $\mathcal{F}_{\mathrm{KM}}$. We will only give a proof sketch here, the complete proof can be found in the full version [18].

Let $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$ (in the following: π) denote the network consisting of the programs $\pi(\mathtt{prot} - \mathtt{fkm})$ and $\pi(\mathtt{prot} - \mathtt{fsetup})$. $\pi(\mathtt{prot} - \mathtt{fkm})$ defines the behaviours for users in $\mathcal{U}, \mathcal{U}^{\mathrm{ext}}$ and $\mathcal{ST}$. Parties in $\mathcal{U} \cup \mathcal{U}^{\mathrm{ext}}$ will act according to the convention on machine corruption defined in [11, § 8.1], while parties in $\mathcal{ST}$ will ignore corruption requests (security tokens are assumed to be incorruptible). $\pi(\mathtt{prot} - \mathtt{fkm})$ is *totally regular*, that is, for other machines, in particular ideal machines, it responds to any message with an error message to the adversary. The protocol π is a $\mathcal{F}_{\mathrm{setup}}$-hybrid protocol.

The proof that π implements $\mathcal{F}_{\mathrm{KM}}$ proceeds in several steps: making use of the composition theorem, the last functionality $\mathcal{F}_l$ in $\mathcal{F}_{\mathrm{KM}}$ can be substituted by its key-manageable implementation $\hat{I}_L$. Then, $\mathcal{F}_{\mathrm{KM}}$ can simulate $\hat{I}$ instead of calling it. Let $\mathcal{F}_{\mathrm{KM}}^{\{\mathcal{F}_l/\hat{I}_l\}}$ be the resulting functionality. In the next step, calls to this simulation are substituted by calls to the functions used in $\hat{I}$, $impl_C$ for each $C \in \mathcal{C}_l$. The resulting, partially implemented functionality $\mathcal{F}_{\mathrm{KM}}^{\{\mathcal{F}_l/\mathtt{Impl}_{\mathcal{F}_l}\}}$ saves keys rather than credentials (for $\mathcal{F}_l$). We repeat the previous steps until $\mathcal{F}_{\mathrm{KM}}$ does not call any KU functionalities anymore, i. e., we have $\mathcal{F}_{\mathrm{KM}}^{\{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_n/\mathtt{Impl}_{\mathcal{F}_n}\}}$. Then we show that the network of distributed token π emulates the monolithic block $\mathcal{F}_{\mathrm{KM}}^{\{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_n/\mathtt{Impl}_{\mathcal{F}_n}\}}$ that does not call KU functionalities anymore, using a reduction to the security of the key-wrapping scheme. This last step requires the restriction of the set of environments to those which guarantee that keys are not corrupted after they have been used to wrap. The notion of a *guaranteeing environment*, and the predicate *corrupt-before-wrap* are formally defined in Appendix D.

The first four steps are the subject of Lemma 1, the last step is Lemma 2:

Lemma 1. *Let $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ be KU parameters such that all $F \in \overline{\mathcal{F}}$ are key-manageable. Let $\mathtt{Impl}_{\mathcal{F}_i}$ be the functions defining the key-manageable implementation $\hat{I}_i$ of $\mathcal{F}_i$. Then $\mathcal{F}_{\mathrm{KM}}^{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_l/\mathtt{Impl}_{\mathcal{F}_l}}$ emulates $\mathcal{F}_{\mathrm{KM}}$. Furthermore, it is poly-time.*

Lemma 2. *For any KU parameter $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ and set of sets of PPT algorithms $\overline{\texttt{Impl}}$, let $\mathcal{F}_{\mathrm{KM}}^{\texttt{impl}} := \mathcal{F}_{\mathrm{KM}}^{\mathcal{F}_1/\texttt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_l/\texttt{Impl}_{\mathcal{F}_l}}$ be the partial implementation of $\mathcal{F}_{\mathrm{KM}}$ with respect to all KU functionalities in $\overline{\mathcal{F}}$. If $KW = (impl_{\texttt{new}}^{\texttt{KW}}, \mathsf{wrap}, \mathsf{unwrap})$ is a secure and correct key-wrapping scheme (Definition 12) then $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ emulates $\mathcal{F}_{\mathrm{KM}}^{\texttt{impl}}$ for environments that guarantee* corrupt-before-wrap.

The main result follows from the transitivity of emulation and Lemmas 1 and 2:

Corollary 1. *Let $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ be KU parameters such that all $F \in \overline{\mathcal{F}}$ are key-manageable. Let $\texttt{Impl}_{\mathcal{F}_i}$ be the functions defining the key-manageable implementation $\hat{I}_i$ of $\mathcal{F}_i$. If $KW = (impl_{\texttt{new}}^{\texttt{KW}}, \mathsf{wrap}, \mathsf{unwrap})$ is a secure and correct key-wrapping scheme, then $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ emulates $\mathcal{F}_{\mathrm{KM}}$ for environments that guarantee* corrupt-before-wrap.

5 Realizing key-usage functionalities for a static key-hierarchy

To demonstrate the use of Corollary 1, we equip the security token with the functionalities $\mathcal{F}_1 = \mathcal{F}_{\mathrm{Rand}}$ and $\mathcal{F}_2 = \mathcal{F}_{\mathrm{SIG}}$ described below. The resulting security token $ST^{\mathcal{F}_{\mathrm{Rand}},\mathcal{F}_{\mathrm{SIG}}}$ is able to encrypt keys and random values and sign user-supplied data. It is not able to sign keys, as this task is part of the key-management. The first functionality, $\mathcal{F}_{\mathrm{Rand}}$, is an unusual functionality, but demonstrates what can be done within the design of $\mathcal{F}_{\mathrm{KM}}$, as well as it's limitations: It models how random values can be stored as keys, with tests of equality and corruption, which means here that the adversary learns the value of the random value. Since our framework requires a strict division between key-management and usage, they can be transmitted (using wrap) and compared, but not appear elsewhere, since other KU functionalities shall not use them. We define $\mathcal{F}_{\mathrm{Rand}}$ as follows:

`new`: accept <`new`> from parentId (=:p);
  $c \leftarrow \{0,1\}^\eta$; $L \leftarrow L \cup \{(c,0)\}$; send <`new`$^\bullet$, c, > to p
`command`: accept <`equal`, c, n> from p;
  if $(c,k) \in L$ for some k
    if $k \notin \mathcal{K}_{\mathrm{cor}}$ send <`equal`$^\bullet$, `false`> to p
    else if $n = k$ send <`equal`$^\bullet$, `true`> to p
`corrupt`: accept <`corrupt`, c> from p;
  if $(c,0) \in L$
    $k \leftarrow \{0,1\}^\eta$; $L \leftarrow (L \setminus \{(c,0)\}) \cup \{(c,k)\}$; $\mathcal{K}_{\mathrm{cor}} = \mathcal{K}_{\mathrm{cor}} \cup \{k\}$;
    send <`corrupt`$^\bullet$, k> to A
`inject`: accept <`inject`, n> from P;
  (c,<`ignore`>) $\leftarrow \mathsf{KG}(1^\eta)$; $\mathcal{K}_{\mathrm{cor}} \leftarrow \mathcal{K}_{\mathrm{cor}} \cup \{n\}$; $L \leftarrow L \cup \{(c,n)\}$;
  send <`inject`$^\bullet$, c> to parentId

(for `new`:) $impl_{\texttt{new}}$ on input 1^η
  $n \leftarrow \{0,1\}^\eta$; output $(n, \backslash\_)$
(for `command`:) $impl_{\texttt{equal}}$ on input n, n'
  output $n = n'$

Due to space restrictions, the signature functionality $\mathcal{F}_{\mathrm{SIG}}$ is presented in Appendix E. In the following, we will consider $\mathcal{F}_{\mathrm{KM}}$ for the parameters $\overline{\mathcal{F}} = \{\mathcal{F}_{\mathrm{Rand}}, \mathcal{F}_{\mathrm{SIG}}\}$, $\overline{\mathcal{C}} = \{\{\texttt{equal}\}, \{\texttt{sign}, \texttt{verify}\}\}$ and a static key-hierarchy Π, which is defined as the relation that consists of all 4-tuples ($\mathcal{F}$,Cmd,attr_1,attr_2) such that the conditions in one of the lines in the following table holds. Note that we omit the "=" sign when we mean equality and "*" denotes that no condition has to hold for the variable.

$\mathcal{F}$	Cmd	attr_1	attr_2
KW	`new`	> 0	*
$\neq$ KW	`new`	0	*
*	`attribute_change`	a	a
KW	`wrap`	> 0	$\mathrm{attr}_1 > \mathrm{attr}_2$
KW	`unwrap`	> 0	$\mathrm{attr}_1 > \mathrm{attr}_2$
$\mathcal{F}_i$	$C \in \mathcal{C}^{priv}$	0	*

(where $a \in \mathbb{N}$)

Theorem 1 allows immediately to conclude some useful properties on this instantiation of $\mathcal{F}_{\mathrm{KM}}$: from *(1)* we conclude that all keys with $c \notin \mathcal{K}_{\mathrm{cor}}$ have the attribute they were created with. This also means that the same credential has the same attribute, no matter which user accesses it. From *(2)*, we can see that for each corrupted credential $c \in \mathcal{K}_{\mathrm{cor}}$, there was either a query $<\texttt{corrupt},\texttt{h}>$, where $\texttt{Store}[U,h]=<\mathcal{F},a,c>$, or there exists $\texttt{Store}[U,h']=<\texttt{KW},a',c'>$, $\texttt{Store}[U,h]=<\mathcal{F},a,c>$ and a query $<\texttt{wrap},h',h,id>$ was emitted, for $c' \in \mathcal{K}_{\mathrm{cor}}$, or an unwrap query $<\texttt{unwrap},h',w,a,F,id>$ for a $c \in \mathcal{K}_{\mathrm{cor}}$ was emitted. By the definition of the strict key-hierarchy policy, in the latter two case we have that $a' > a$. It follows that, for any credential c for $\mathcal{F}$, such that $\texttt{Store}[U,h]=<\mathcal{F},a,c>$ for some U, h and a, $c \notin \mathcal{K}_{\mathrm{cor}}$, as long as every corruption query $<\texttt{corrupt},\texttt{h}^*>$ at U was addressed to a different key of lower or equal rank key, i. e., $\texttt{Store}[U,h^*]=<\texttt{KW},a^*,c^*>$, $c^* \neq c$ and $a^* \leq a$. By *(3)*, those credentials have not been corrupted in their respective functionality, i. e., it has never received a message $<\texttt{corrupt},c>$.

6 Conclusions and outlook

We have presented a provably secure framework for key management in the GNUC model. In further work, we are currently developing a technique for transforming functionalities that use keys but are not key-manageable into key-manageable functionalities in the sense of Definition 4. This way, existing proofs could be used to develop a secure implementation of cryptographic primitives in a plug-and-play manner. Investigating the restrictions of this approach could teach us more about the modelling of keys in simulation-based security.

References

1. RSA Security Inc.: PKCS #11: Cryptographic Token Interface Standard v2.20. (June 2004)
2. IBM: CCA Basic Services Reference and Guide. (October 2006) Available online at http://www-03.ibm.com/security/cryptocards/pdfs/bs327.pdf.
3. Trusted Computing Group: TPM Specification version 1.2. Parts 1–3, revision 103. http://www.trustedcomputinggroup.org/resources/tpm_main_specification (2007)
4. Longley, D., Rigby, S.: An automatic search for security flaws in key management schemes. Computers and Security **11**(1) (March 1992) 75–89
5. Bond, M., Anderson, R.: API level attacks on embedded systems. IEEE Computer Magazine (October 2001) 67–75
6. Bortolozzo, M., Centenaro, M., Focardi, R., Steel, G.: Attacking and fixing PKCS#11 security tokens. In: Proc. 17th ACM Conference on Computer and Communications Security (CCS'10), Chicago, Illinois, USA, ACM Press (October 2010) 260–269
7. Cortier, V., Keighren, G., Steel, G.: Automatic analysis of the security of XOR-based key management schemes. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07). Number 4424 in LNCS (2007) 538–552
8. Delaune, S., Kremer, S., Steel, G.: Formal analysis of PKCS#11 and proprietary extensions. Journal of Computer Security **18**(6) (November 2010) 1211–1245
9. Cachin, C., Chandran, N.: A secure cryptographic token interface. In: Proc. 22th IEEE Computer Security Foundation Symposium (CSF'09), IEEE Comp. Soc. Press (2009) 141–153
10. Kremer, S., Steel, G., Warinschi, B.: Security for key management interfaces. In: Proc. 24th IEEE Computer Security Foundations Symposium (CSF'11), IEEE Comp. Soc. Press (2011) 66–82
11. Hofheinz, D., Shoup, V.: GNUC: A new universal composability framework. Cryptology ePrint Archive, Report 2011/303 (2011) http://eprint.iacr.org/.
12. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067 (December 2005) Updated version of [19] http://eprint.iacr.org/.
13. Küsters, R., Tuengerthal, M.: Ideal Key Derivation and Encryption in Simulation-Based Security. In: Topics in Cryptology - CT-RSA'11. Volume 6558 of LNCS., Springer (2011) 161–179
14. Hofheinz, D.: Possibility and impossibility results for selective decommitments. J. Cryptology **24**(3) (2011) 470–516
15. Backes, M., Dürmuth, M., Hofheinz, D., Küsters, R.: Conditional reactive simulatability. International Journal of Information Security (IJIS) (2007)
16. Küsters, R.: Simulation-Based Security with Inexhaustible Interactive Turing Machines. In: Proc. 19th IEEE Computer Security Foundations Workshop (CSFW'06), IEEE Comp. Soc. Press (2006) 309–320
17. Maurer, U., Renner, R.: Abstract cryptography. In: Proc. 2nd Symposium in Innovations in Computer Science (ICS'11), Tsinghua University Press (2011) 1–21

18. Kremer, S., Künnemann, R., Steel, G.: Universally composable key-management. IACR Cryptology ePrint Archive (2012)
19. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proc. 42nd Annual Symposium on Foundations of Computer Science (FOCS'01), IEEE Computer Society Press (October 2001) 136–145
20. Rogaway, P., Shrimpton, T.: Deterministic authenticated encryption: A provable-security treatment of the keywrap problem. In: Advances in Cryptology — EUROCRYPT'06. Volume 4004 of LNCS., Springer (2006) 373–390
21. Küsters, R., Tuengerthal, M.: Joint State Theorems for Public-Key Encryption and Digitial Signature Functionalities with Local Computation. In: Proc. 21st IEEE Computer Security Foundations Symposium (CSF'08), IEEE Comp. Soc. Press (2008) 270–284

A Initialisation and Setup

A.1 Initialisation phase

All regular protocol machines that shall accept messages from $\mathcal{F}_{\mathrm{KM}}$ need to send a message to $\mathcal{F}_{\mathrm{KM}}$ first [11, § 4.5]. A similar behaviour needs to be emulated by $\mathcal{F}_{\mathrm{setup}}$ in the network with the actual tokens. The involved protocol machines are $\mathcal{M} := \mathcal{U} \cup \mathcal{ST} \cup \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$, where $\mathcal{F}_i$ denotes the regular protocol machine that makes subroutine calls to $\mathcal{F}_i$, identified with the machine id <<reg, $\mathcal{F}_i$>, sid >.

We add the following part to the definition of $\mathcal{F}_{\mathrm{KM}}$:

ready$-P$: accept <ready> from $P \in \mathcal{M}$
  send <ready$^\bullet$, P> to A
ready [ready$-P$ $\forall P \in \mathcal{M}$]

Listing 10: Initilisation ($\mathcal{F}_{\mathrm{KM}}$).

We add the following part to the definition of ST:

ready [¬ ready]:
accept <ready> from parentId
  call $\mathcal{F}_{\mathrm{setup}}$ with <ready>

Listing 11: Initilisation (ST_i).

A.2 Handling of the setup phase in $\mathcal{F}_{\mathrm{KM}}$ and ST_i

The following listings describe the setup phase, introduced on page 9:

share[¬finish_setup∧ready]: accept <share, h_1, U_2> from $U_1 \in \mathcal{U}$;
  if $U_i, U_2 \in$ *Room*
    create h_2; Store [U_2, h_2]=Store [U_1, h_1]; send <share$^\bullet$, h_2> to U_2

Listing 12: The setup phase: sharing keys ($\mathcal{F}_{\mathrm{KM}}$).

share[¬finish_setup∧ready]: accept <share, h_1, U_2> from $\mathcal{F}_{\mathrm{setup}}$;
  if Store [U, h_1]=s call $\mathcal{F}_{\mathrm{setup}}$ with <send, s, U_2>
import[¬finish_setup∧ready]:accept <deliver, s, U_1> from $\mathcal{F}_{\mathrm{setup}}$
  if $U_1 \in$ *Room* create h_2; Store [U_i, h_2]=s; send <share$^\bullet$, h_2> to $\mathcal{F}_{\mathrm{setup}}$

Listing 13: The setup phase: sharing keys (ST_i).

finish_setup[¬finish_setup∧ready]: accept <finish_setup> from $U \in \mathcal{U}$;
  send <finish_setup$^\bullet$> to A

Listing 14: The setup phase: terminating the setup phase ($\mathcal{F}_{\mathrm{KM}}$).

finish_setup[¬finish_setup∧ready]: accept <close> from $\mathcal{F}_{\text{setup}}$;
  send <close$^\bullet$> to $\mathcal{F}_{\text{setup}}$

Listing 15: The setup phase: terminating the setup phase (ST_i).

A.3 Setup assumptions for the implementation

The setup assumption used for $\mathcal{ST}$ is subsumed in the setup functionality $\mathcal{F}_{\text{setup}}$, which is defined as follows:

ready$-U_i$: accept <ready, ST_i> from $U_i \in \mathcal{U}$
  send <ready$^\bullet$, U_i> to A
ready$-P$: accept <ready> from $P \in \mathcal{M} \setminus \mathcal{U}$
  send <ready$^\bullet$, P> to A
ready [ready$-P$ $\forall P \in \mathcal{M}$]
share[ready∧ ¬finish_setup]: accept <send, x, ST_j> from ST_i
  if $U_i, U_j \in$ *Room*
    send <deliver, x, ST_i> to ST_j
  else
    send <$\perp$, ST_i> to U_i
finish_setup[ready∧¬finish_setup]:
  accept <finish_setup> from $U \in \mathcal{U}$
  from i:=1 to n
    send <close> to ST_i; accept <close$^\bullet$> from ST_i
  send <finish_setup$^\bullet$> to A
relay_receive [ready]: accept <x, ST_i> from U_i; send <x> to ST_i
relay_send [ready]: accept <x> from ST_i;send <x, ST_i> to U_i

B Security Token Network

Definition 10 (Security token network). *For KU parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ and implementation functions $\overline{\texttt{Impl}} := \{\texttt{Impl}_F\}_{F \in \overline{\mathcal{F}}}$, define the protocol $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ as follows: $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ defines only $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}(\texttt{prot} - \texttt{fkm})$ and $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}(\texttt{prot} - \texttt{fsetup})$. The session parameter is expected to be an encoding of the network parameters $\mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}$, Room. The code executed depends on the party running $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}(\texttt{prot} - \texttt{fkm})$: if the party has identity <<reg, u−i>, sid>, the following code U_i is executed:*

relay_to: *accept <m> from parentId*
  call $\mathcal{F}_{\text{setup}}$ with <m, ST_i>
relay_from: *accept <m, ST_i> or <$m = \perp$> from $\mathcal{F}_{\text{setup}}$*
  send <m> to parentId
public_command:
  accept <C, public, m> from parentId
  if $C \in \mathcal{C}_{i,pub}$
    send <$C^\bullet$, $impl_C(public, m)$> to parentId

If the party has identity ext−u−i, the following code U_i^{ext} is executed:

public_command:
  accept <C, public, m> from parentId
  if $C \in \mathcal{C}_{i,pub}$
    send <$C^\bullet$, $impl_C(public, m)$> to parentId

A regular protocol machine with machine id <reg, $\mathcal{F}_i$>, sid for $\mathcal{F}_i \in \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$ runs the following code:

ready: *accept* <ready> *from parentId*
call $\mathcal{F}_{\text{setup}}$ *with* <ready>

All other regular protocol machines run the code of the dummy adversary. If the party has identity <<reg, st-i>, sid>, *then* ST_i *is executed. The code for* ST_i *is given in Section 3.4 and in Appendix A. For parties with the identities* <<reg, u-i>, sid> *or* <<reg, ext-u-i>, sid>, $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ *will act according to the convention on machine corruption defined in [11, § 8.1], while for* <<reg, st-i>, sid>, *it will ignore corruption request (security tokens are assumed to be incorruptible). For other machines, including ideal machines, it responds to any message with an error message to the adversary, i. e.,* $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}(\texttt{prot}-\texttt{fkm})$ *is* totally regular. $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}(\texttt{prot}-\texttt{fkm})$ *declares the use of* prot-fsetup *as a subroutine.* $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}(\texttt{prot}-\texttt{fsetup})$ *runs* $\mathcal{F}_{\text{setup}}$, *i. e.,* $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ *is a* $\mathcal{F}_{\text{setup}}$*-hybrid protocol.*

C Proof for Theorem 1

Theorem 1 (Properties of $\mathcal{F}_{\text{KM}}$). *Every instance of* $\mathcal{F}_{\text{KM}}$ *with parameters* $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ *and session parameters* $\mathcal{U}, \mathcal{U}^{\text{ext}}$, $\mathcal{ST}$, *Room has the following properties:*

(1) At any step of an execution of $[\mathcal{F}_{\text{KM}}, A_D, Z]$, *the following holds for* $\mathcal{F}_{\text{KM}}$*: for all* $\texttt{Store}[U,h] =< \mathcal{F}, a, c >$ *such that* $c \notin \mathcal{K}_{\text{cor}}$, *there is a node* a' *marked* new *in the attribute policy graph* $\mathcal{A}_{\Pi,\mathcal{F}}$ *such that* a *is reachable from* a' *in* $\mathcal{A}_{\Pi,\mathcal{F}}$ *and there was a step* new *where* $\texttt{Store}[U',h'] =< \mathcal{F}, a', c >$ *was added.*

(2) At any step of an execution of $[\mathcal{F}_{\text{KM}}, A_D, Z]$, *the following holds for* $\mathcal{F}_{\text{KM}}$*: all* $c \in \mathcal{K}_{\text{cor}}$ *were either*

(a) directly corrupted: *there was a* corrupt *triggered by a query* $<$ corrupt, h $>$ *from* U *while* Store[U, h]=< $\mathcal{F}, a, c >$, *or indirectly, that is,*

(b) corrupted via wrapping: *there is* $c' \in \mathcal{K}_{\text{cor}}$ *such that at some point the* wrap *step was triggered by a message* <wrap, h', h, id> *from* U *while* Store[U, h']=< KW, $a', c' >$, Store[U, h]=< $\mathcal{F}, a, c >$, *or*

(c) corrupted via unwrapping (injected): *there is* $c' \in \mathcal{K}_{\text{cor}}$ *such that at some point the* unwrap *step was triggered by a message* <unwrap, h', w, a, F, id> *from* U *while* Store[U, h']=< KW, $a', c' >$ *and* $c = \text{unwrap}_{c'}^{<F,a,id>}(w)$ *for some* a, F *and* id.

(3) At any step of an execution of $[\mathcal{F}_{\text{KM}}, A_D, Z]$, *the following holds: whenever an ideal machine* $\mathcal{F}_i =$ <ideal, <sid, <$\mathcal{F}_i$, F>>>, $F =$<<reg, $\mathcal{F}$ >, <sid>>, *accepts the message* <corrupt, c> *for some* c *such that* $\mathcal{F}_{\text{KM}}$ *in session* sid *has an entry* Store[U, h]= <$\mathcal{F}_i$, a, c>, *then* $c \in \mathcal{K}_{\text{cor}}$ *in* $\mathcal{F}_{\text{KM}}$.

Proof. (1) Proof by induction over the number of epochs since $\mathcal{F}_{\text{KM}}$'s first activation, t: if $t = 0$, Store is empty. $t > 0$: since the property was true in the previous step, there are only three steps we need to look at: if a key $< \mathcal{F}, \texttt{a}, \texttt{c} >$ is added to Store at step new, then it is created only if the policy contains an entry $(\mathcal{F}, \texttt{new}, \texttt{a})$, i. e., a itself is a new node. If a key $< \mathcal{F}, \texttt{a}, \texttt{c} >$ is added to Store at step unwrap, let <unwrap, h_1, w, $\mathcal{F}$, id> be the arguments sent by a user U, and Store[U, h_1]=<KW, a_w, c_w>. If $c \notin \mathcal{K}_{\text{cor}}$, then $c_w \notin \mathcal{K}_{\text{cor}}$, too, and thus there is an entry $< c, < \mathcal{F}, a, id >, w > \in$ encs$[c_w]$. encs is only written in wrap, therefore there was a position $[U', h']$ in the store, such that Store[U', h']=<$\mathcal{F}$, a, c>. Using the induction hypothesis, we see that a is reachable in the attribute policy graph. The third and last step where the store is written to is AttributeChange. If this step alters the attribute from a' to a, there must have been an entry $(\mathcal{F}, \texttt{attribute\_change}, a', a) \in \Pi$. By induction hypothesis, a' is reachable from a new node, therefore a' is, too.

(2) A credential c is only added to the set $\mathcal{K}_{\text{cor}}$ in three steps: if it is added in corrupt, then a message <corrupt, h> was received from $U \in \mathcal{U}$ and Store[U, h]=< $F, a, c >$. If it was added in wrap, then a message <wrap, h_1, h_2> must have been received from $U \in \mathcal{U}$ while Store[U, h_1]=< KW, $a_1, c_1 >$, and c was reachable from c_1. Let c' be the last node on the path to c. $c' \in \mathcal{K}_{\text{cor}}$ because it is reachable from c_1, too. Since $(c', c) \in \mathcal{W}$, there was another wrapping query <wrap, h', h, id> with Store[U, h']=< KW, $a', c' >$ and Store[U, h]=< $\mathcal{F}, a, c >$. Since entries in the store are never deleted (only the attribute can be altered), and credentials are never removed from $\mathcal{K}_{\text{cor}}$, the property holds in this case. If it was added in unwrap, then $c \notin \mathcal{K}$ at this point in time, and $c = \text{unwrap}_{c'}^{<F,a,id>}(w) \neq \perp$. Furthermore, we observe that the conditionals prior to adding c to $\mathcal{K}_{\text{cor}}$ require that Store[U,h']=<KW, a', c'>, $c' \in \mathcal{K}_{\text{cor}}$, and that the step was triggered by a message <unwrap, h', w, a, F, id>.

(3) $\hat{I}_i$ accepts only messages coming from the party F, and F in turn only accepts messages coming from $\mathcal{F}_{\mathrm{KM}}$. Therefore, we can conclude from the definition of step `corrupt` in $\mathcal{F}_{\mathrm{KM}}$ that $c \in \mathcal{K}_{\mathrm{cor}}$.

D Proofs for Lemma 1 and Lemma 2

The static call graph has only an edge from `prot-fkm` to `prot-fsetup`, $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ is thus rooted at `prot-fkm`.

Lemma 3. *For all KU parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$, $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ is a poly-time protocol.*

Proof. By Definition 2 in [11, § 6], we need to show that there exists a polynomial p such that for every well-behaved environment Z that is rooted at `prot-fkm`, we have:

$$\begin{aligned}&\Pr[\mathrm{Time}_{\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta)\\ &\quad > p(\mathrm{Flow}_{Z\to\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},A_D}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta))]\\ &\quad = \mathrm{negl}(\eta).\end{aligned}$$

Let $p_{\max}$ be a polynomial such that for all $\mathcal{F}_i$ and $C \in \mathcal{C}$, the algorithm $impl_C$ terminates in a running time smaller than $p_{\max}(n)$, where n is the length of the input. $impl^F_{\texttt{new}}$ is always called on input of length η, thus all keys have a length smaller $p_{\max}(\eta)$. In step `new`, F and a are provided by the environment (as input to U_i, which then asks $\mathcal{F}_{\mathrm{setup}}$ to relay the request to ST_i). Since messages have at least length η, we can overapproximate by saying that `Store` grows at most by some polynomial $p_{\mathrm{growth-new}}$ in the length of the environment's input. Similarly, an `<unwrap,...>` query cannot grow the Store by more than $p_{\mathrm{growth-unwrap}}$. Therefore, at any point in time t (we simply count the number of epochs, i. e., activations of the environment), the store is smaller than $p'(\mathrm{Flow}_{Z\to\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},A_D}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta))$ for a polynomial p'.

We observe that there is not a single activation of a machine in $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$, neither a U_i, an ST_i nor $\mathcal{F}_{\mathrm{setup}}$, where the running time is not polynomial in the environment's input and the length of the `Store`. A_D might corrupt user $U \in \mathcal{U} \cup \mathcal{U}^{\mathrm{ext}}$, but they do not have any state. Thus, we have for the running time of $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ at point t, i. e., $\mathrm{Time}_{\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},t}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta)$,

$$\begin{aligned}&\mathrm{Time}_{\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},t}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta)\\ &\quad = \mathrm{Time}_{\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},t-1}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta)+\\ &\qquad p'(\mathrm{Flow}_{Z\to\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},A_D}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta))\\ &\quad \le t \cdot p'(\mathrm{Flow}_{Z\to\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},A_D}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta))\\ &\quad \le p''(\mathrm{Flow}_{Z\to\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},A_D}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta))\end{aligned}$$

for another polynomial p'', because

$$t < \mathrm{Flow}_{Z\to\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},A_D}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta)).$$

The proof that π implements $\mathcal{F}_{\mathrm{KM}}$ proceeds in several steps: making use of the composition theorem, the last functionality $\mathcal{F}_l$ in $\mathcal{F}_{\mathrm{KM}}$ can be substituted by its key-manageable implementation $\hat{I}_L$. Then, $\mathcal{F}_{\mathrm{KM}}$ can simulate $\hat{I}$ instead of calling it. Let $\mathcal{F}_{\mathrm{KM}}^{\{\mathcal{F}_l/\hat{I}_l\}}$ be the resulting functionality. In the next step, calls to this simulation are substituted by calls to the functions used in $\hat{I}$, $impl_C$ for each $C \in \mathcal{C}_l$. The resulting, partially implemented functionality $\mathcal{F}_{\mathrm{KM}}^{\{\mathcal{F}_l/\texttt{Impl}_{\mathcal{F}_l}\}}$ saves keys rather than credentials (for $\mathcal{F}_l$). We repeat the previous steps until $\mathcal{F}_{\mathrm{KM}}$ does not call any KU functionalities anymore, i. e., we have $\mathcal{F}_{\mathrm{KM}}^{\{\mathcal{F}_1/\texttt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_n/\texttt{Impl}_{\mathcal{F}_n}\}}$. Then we show that the network of distributed token π emulates the monolithic block $\mathcal{F}_{\mathrm{KM}}^{\{\mathcal{F}_1/\texttt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_n/\texttt{Impl}_{\mathcal{F}_n}\}}$ that does not call KU functionalities anymore, using a reduction to the security of the key-wrapping scheme.

The first four steps will be the subject of Lemma 1, the last step is Lemma 2. But before we come to this, the following definition expresses partial implementations of $\mathcal{F}_{\mathrm{KM}}$. In fact, the formal definition of $\mathcal{F}_{\mathrm{KM}}$ is the special case in which the set of substituted functionalities is empty:

Definition 11 ($\mathcal{F}_{\mathrm{KM}}$ with partial implementation). *Given the KU parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$, and functions $(impl^{\mathtt{KW}}_{\mathtt{new}}, \mathsf{wrap}, \mathsf{unwrap})$, let $\mathtt{Impl}_{\mathcal{F}_i}$ be the algorithms defining the keymanageable implementation $\hat{I}_i$ of $\mathcal{F}_i \in \{\mathcal{F}_1, \ldots, \mathcal{F}_p\} \subset \overline{\mathcal{F}}$. We will define the partial implementation of $\mathcal{F}_{\mathrm{KM}}$ with respect to the KU functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_p$, denoted $\mathcal{F}_{\mathrm{KM}}^{\{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1}, \ldots, \mathcal{F}_p/\mathtt{Impl}_{\mathcal{F}_p}\}}$,*

Let the ideal protocols $\mathcal{F}_{p+1}, \ldots, \mathcal{F}_l$ be rooted at $\mathtt{prot}\text{-}\mathcal{F}_{p+1}, \ldots, \mathtt{prot}\text{-}\mathcal{F}_l$. In addition to those protocols names, $\mathcal{F}_{\mathrm{KM}}^{\{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1}, \ldots, \mathcal{F}_p/\mathtt{Impl}_{\mathcal{F}_p}\}}$, defines the protocol name $\mathtt{prot}\text{-}\mathtt{fkm}$. For $\mathtt{prot}\text{-}\mathtt{fkm}$, the protocol defines the following behaviour: a regular protocol machine with machine id $\langle\langle \mathtt{reg}, \mathcal{F}_i \rangle, \mathtt{sid}\rangle$ for $\mathcal{F}_i \in \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$ runs the following code:

```
ready: accept <ready> from parentId
  send <ready> to <ideal, sid> (= F_KM)
relay_to: accept <m> from <ideal, sid> (= F_KM)
  send <m> to <<reg, F_i>, <sid, <prot-F_i, <>>>> (= F_i)
relay_from: accept <m> from <<reg, F_i>, <sid, <prot-F_i, <>>>>
  send <m> to <ideal, sid> (= F_KM)
```

The ideal party runs the logic for $\mathcal{F}_{\mathrm{KM}}$ described in Section-3.4, with the following alteration of the corrupt macro used in the $\mathtt{corrupt}$ and $\mathtt{wrap}$ step:

```
K_cor ← K_cor ∪ {c}
for any Store[U, h] = <F, a, c>
  if F ∈ {KW, F_1, ..., F_p}
    key[c] ← c
    send <corrupt•, h, c> to A
  else
    call F with <corrupt, c>
    accept <corrupt•, k> from F
    key[c] ← k
    send <corrupt•, h, k> to A
```

Listing 16: procedure for corrupting a credential c

and in the $\mathtt{new}$, $\mathtt{command}$, $\mathtt{public\_command}$ and $\mathtt{unwrap}$ steps:

```
new[ready]: accept <new, F, a> from U ∈ U
if <F, new, a, *> ∈ Π then
  if F ∈ {F_1, ..., F_p}
    (k, public) ← impl^F_new(1^η)
    create h; Store[U, h] ← <F, a, k>
    K ← K ∪ {k}
    send <new•, h, public> to U
  else if F = KW
    (k, public) ← impl^KW_new(1^η)
    if k ∈ K ∪ K_cor
      send <error> to A
    else
      create h; Store[U, h] ← <F, a, k>
      K ← K ∪ {k}
      send <new•, h, public> to U
  else
    call F with <new>
    accept <new•, c, public> from F
    if c ∈ K ∪ K_cor
      send <error> to A
    else
      create h; Store[U, h] ← <F, a, c>
      K ← K ∪ {c}
```

send <new$^\bullet$, h, public> *to* U

`command[finish_setup]`*:*
accept <C, h, m> *from* $U \in \mathcal{U}$
if *Store* [U, h]=<F, a, c> *and* <F, C, a, $\star$>$\in \Pi$
if $F \in \{\mathcal{F}_1, \ldots, \mathcal{F}_p\}$
send <$C^\bullet$, $impl_C(c, m)$> *to* U
else if $F \neq$KW
call F *with* <C, c, m>
accept <$C^\bullet$, r> *from* F
send <$C^\bullet$, r> *to* U

`public_command`*:*
accept <C, *public*, m> *from* $U \in \mathcal{U}$
if $C \in \mathcal{C}_{i,pub}$
if $F \in \{\mathcal{F}_1, \ldots, \mathcal{F}_p\}$
send <$C^\bullet$, $impl_C(public, m)$> *to* U
else
call F_i *with* <C, *public*, m>
accept <$C^\bullet$, r> *from* F_i
send <$C^\bullet$, r> *to* U

`unwrap[finish_setup]`*:*
accept <unwrap, h_1, w, a_2, F_2, id>
from $U \in \mathcal{U}$
if *Store* [U, h_1]=<KW, a_1, c_1> *and*
<KW, unwrap, a_1, a_2>$\in \Pi$,$F_2 \in \overline{\mathcal{F}}$
if $c_1 \in \mathcal{K}_{\text{cor}}$
$c_2 = \mathsf{unwrap}^{<F_2, a_2, id>}(c_1, w)$
if $c_2 \neq \perp$ *and* $c_2 \notin \mathcal{K}$
$\mathcal{K}_{\text{cor}} \leftarrow \mathcal{K}_{\text{cor}} \cup \{c_2\}$
if $F_2 \in \{\text{KW}, \mathcal{F}_1, \ldots, \mathcal{F}_p\}$
create h_2
Store [U, h_2] $\leftarrow$ <F_2, a_2, c_2>
key [c_2] =c_2
else
call F_2 *with* <inject, c_2>
accept <inject$^\bullet$, c'>
if $c' \notin \mathcal{K} \cup \mathcal{K}_{\text{cor}}$
create h_2
Store [U, h_2] $\leftarrow$ <F_2, a_2, c'>
key [c'] =c_2
send <unwrap$^\bullet$, h> *to* U
else if $c_2 \neq \perp$, $c_2 \in \mathcal{K}$ *and* $c_2 \in \mathcal{K}_{\text{cor}}$
create h_2
Store [U, h_2] $\leftarrow$ <F_2, a_2, c_2>
send <unwrap$^\bullet$, h> *to* U
else // $(c_2 = \perp \vee c_2 \in \mathcal{K} \setminus \mathcal{K}_{\text{cor}})$
send <error> *to* A
else if ($c_1 \notin \mathcal{K}_{\text{cor}}$ *and*
$\exists! c_2.$<c_2, <F_2, a_2, id>,w>$\in$*encs* [c_1] *)*
create h_2
Store [U, h_2] $\leftarrow$ <F_2, a_2, c_2>
send <unwrap$^\bullet$, h> *to* U

Note that the partial implementation of $\mathcal{F}_{\mathrm{KM}}$ is not an ideal protocol in the sense of [11, § 8.2], since not every regular protocol machine runs the dummy party protocol – the party <reg, $\mathcal{F}_i$> relays the communication with the KU functionalities.

Lemma 1. *Let $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ be KU parameters such that all $F \in \overline{\mathcal{F}}$ are key-manageable. Let $\mathtt{Impl}_{\mathcal{F}_i}$ be the functions defining the key-manageable implementation $\hat{I}_i$ of $\mathcal{F}_i$. Then $\mathcal{F}_{\mathrm{KM}}^{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_l/\mathtt{Impl}_{\mathcal{F}_l}}$ emulates $\mathcal{F}_{\mathrm{KM}}$. Furthermore, it is poly-time.*

Proof. Induction on the number of substituted KU functionalities.

Base case: $\mathcal{F}_{\mathrm{KM}}^{\emptyset}$ actually equals $\mathcal{F}_{\mathrm{KM}}$. Since emulation is reflexive, $\mathcal{F}_{\mathrm{KM}}^{\emptyset}$ emulates $\mathcal{F}_{\mathrm{KM}}$. It is left to show that $\mathcal{F}_{\mathrm{KM}}$ is poly-time: the argument is actually the same as for $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$ (see proof to Lemma 3), after we have established five things: 1. instead of calling implementation functions $impl_C^F$ for $\mathcal{F} \neq \mathtt{KW}$, $\mathcal{F}_{\mathrm{KM}}$ is calling the function $\mathcal{F}$ with the same value. Since $\mathcal{F}$ is key-manageable, it is also poly-time. 2. The implementation function for wrapping might run a different value, but it has the same length, i. e., the same upper bound holds for its running time. 3. Graph reachability is linear in the number of credentials, which in turn is polynomial, because the flow from the environment is polynomial, and thus the number of `new` queries. 4. The relaying of messages from U_i via $\mathcal{F}_{\mathrm{setup}}$ does not add more than linearly in η to the running time, 5. similarly, for the distribution of the <finish_setup> message.

Induction Step:. Assume $i \geq 1$ and that $\mathcal{F}_{\mathrm{KM}}^{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_{i-1}/\mathtt{Impl}_{\mathcal{F}_{i-1}}}$ emulates $\mathcal{F}_{\mathrm{KM}}$. Since emulation is transitive, it suffices to show that $\mathcal{F}_{\mathrm{KM}}^{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_i/\mathtt{Impl}_{\mathcal{F}_i}}$ emulates $\mathcal{F}_{\mathrm{KM}}^{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_{i-1}/\mathtt{Impl}_{\mathcal{F}_{i-1}}}$. We will proceed in three steps: first, we will substitute $\mathcal{F}_i$ by its key-manageable implementation $\hat{I}_i$. Then, we will alter $\mathcal{F}_{\mathrm{KM}}$ to simulate $\hat{I}_i$ inside. The main part of the proof is showing that $\hat{I}_i$ can be emulated by calling $\mathtt{Impl}_{F_i}$ inside $\mathcal{F}_{\mathrm{KM}}$, storing keys instead of credentials.

The first step is a consequence of composition theorem [11, Theorem 7]. The induction hypothesis gives us that $\mathcal{F}_{\mathrm{KM}}^{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_{i-1}/\mathtt{Impl}_{\mathcal{F}_{i-1}}}$ (in the following: $\mathcal{F}_{\mathrm{KM}}^{i-1}$) is a poly-time protocol, and it is rooted in `fkm`. Since $\mathcal{F}_i$ is key-manageable, we know that $\hat{I}_i$ is a polytime protocol that emulates $\hat{I}_i$. $\hat{I}_i$ defines only `F`$-i$, therefore $\hat{I}_F$ is substitutable for $\mathcal{F}_i$ in $\mathcal{F}_{\mathrm{KM}}^{i-1}$. Hence, $F^{i-1}[\mathcal{F}_i/\hat{I}_i]$ is poly-time and emulates $\mathcal{F}_{\mathrm{KM}}^{i-1}$.

In the second step, we alter $F^{i-1}[\mathcal{F}_i/\hat{I}_i]$ (in the following: $\mathcal{F}_{\mathrm{KM}}^{i-1'}$) such that the ideal functionality defined in $\mathcal{F}_{\mathrm{KM}}^{i-1'}(\mathtt{prot}-\mathtt{fkm})$ simulates $\hat{I}_i$ locally, and calls this simulation whenever <<reg, $\mathcal{F}_i$>, sid> would be addressed in $\mathcal{F}_{\mathrm{KM}}$. $\hat{I}_i$ might send a message to A, in which case this message is indeed relayed to A. Since the simulation will only be called by $\mathcal{F}_{\mathrm{KM}}$, it will only respond to $\mathcal{F}_{\mathrm{KM}}$. We will call this protocol $\mathcal{F}_{\mathrm{KM}}^{i-1''}$. To show that $\mathcal{F}_{\mathrm{KM}}^{i-1''}$ emulates $\mathcal{F}_{\mathrm{KM}}^{i-1'}$, we have to make sure that in $\mathcal{F}_{\mathrm{KM}}^{i-1'}$, $\hat{I}_i$ can only be addressed by $\mathcal{F}_{\mathrm{KM}}$, via the relay mechanism implemented in <<reg, $\mathcal{F}_i$>, sid>. (Which consequently is not present in $\mathcal{F}_{\mathrm{KM}}^{i-1''}$, since any call to <<reg, $\mathcal{F}_i$>, sid> is substituted by calls to the simulation of $\hat{I}_i$.) If this is the case, then the observable output to the environment is exactly the same. First, $\hat{I}_i$ never adresses <adv>, so by C5, it cannot be adressed by the adversary. Second, since the environment is rooted at `proto-fkm`, it cannot address $\hat{I}_i$. Third, there is no other regular party than <<reg, $\mathcal{F}_i$>, sid> in `sid` that calls $\hat{I}_i$. By C8, there cannot be other regular machines addressing $\hat{I}_i$. Therefore, $\mathcal{F}_{\mathrm{KM}}^{i-1''}$ emulates $\mathcal{F}_{\mathrm{KM}}^{i-1'}$. Since $\hat{I}_i$ is poly-time, $\mathcal{F}_{\mathrm{KM}}^{i-1''}$ can simulate it and is still poly-time.

In the third step, we show that F_i emulates $\mathcal{F}_{\mathrm{KM}}^{i-1''}$. We claim that in fact, with overwhelming probability, F_i provides a perfect simulation of $\mathcal{F}_{\mathrm{KM}}^{i-1''}$, namely, when the list L maintained in $\mathcal{F}_{\mathrm{KM}}^{i-1}$ describes a function from credentials to keys. Whenever a pair (c,k) is added to L (this happens only in steps `new` and `inject`), $impl_{\mathtt{new}}^{\mathcal{F}_i}$ is used to draw c. Since for all k, $\Pr[k' = k | k' \leftarrow impl_{\mathtt{new}}^{\mathcal{F}_i}(1^\eta)]$ is negligible by assumption (see Definition 4), there is (with overwhelming probability) for every c there is not more than one k such that $(c,k) \in L$, and thus L describes a function from credentials to keys. So we can assume without loss of generality that this list describes a function. We will inspect the steps `new`, `command`, `wrap`, `unwrap` and `corrupt`, as the only steps that produce an output depending on the value of the credential. Note first that, since $\Pr[k' = k | k' \leftarrow impl_{\mathtt{new}}^{\mathcal{F}_i}(1^\eta)]$ is negligible for all k, and since both $\mathcal{K}$ and $\mathcal{K}_{\mathrm{cor}}$ are only polynomial in size, the checks for $c \in \mathcal{K} \cup \mathcal{K}_{\mathrm{cor}}$ in step `new` pass only with negligible probability. Therefore, we can assume those checks to be non-operations. The steps `new`, `corrupt` and `command` are trivial to verify: each credential is substituted by the corresponding key. The corruption macro used in both steps `corrupt`

and `wrap` makes sure that for each credential $c \in \mathcal{K} \cap \mathcal{K}_{\text{cor}}$, `key[c]` contains the same key that the bijection defined by L in $\mathcal{F}_{\text{KM}}^{i-1}$ assigns to it. Furthermore, for each $c \notin \mathcal{K}, c \in \mathcal{K}_{\text{cor}}$, the step `unwrap` gives the same guarantee (by definition of step `inject` in Definition 4). Therefore, the step `wrap` correctly substitutes corrupted credentials by keys. Since $\hat{I}_i$ is key-manageable, and both credential and keys are drawn using $impl_{\texttt{new}}^{\mathcal{F}_i}$, with overwhelming probability, the substitution is correct for uncorrupted credentials, too. The last step to check is `unwrap`. Unless $c_1 \in \mathcal{K}_{\text{cor}}$ and $c_2 \notin \mathcal{K}$, this step restores only a previously created credential in the Store, so no substitution necessary. In case that $c_1 \in \mathcal{K}_{\text{cor}}$ and $c_2 \notin \mathcal{K}$, a credential that is freshly created and linked to the content of the wrapping (see the `inject` step) is stored, whereas in $\mathcal{F}_{\text{KM}}^{i-1''}$, it is the content of the wrapping itself that is stored.

By transitivity of emulation, we have that F_i emulates $\mathcal{F}_{\text{KM}}$. By the fact that F_i actually computes less than $\mathcal{F}_{\text{KM}}^{i-1''}$, we know it is poly-time.

For the next step, we need to define what we understand under a key-wrapping scheme. We took the definition from [10] as a basis and will repeat it here. It is based on the notion of deterministic, authenticated encryption from [20], but it additionally supports key-dependant messages. We changed the definition, so it allows to wrap the same key with the same wrapping key but under different attributes, just like in the DAE definition from [20].

Definition 12 (Multi-user setting for key wrapping). *We define experiments* $\mathbf{Exp}_{\mathcal{A},\mathsf{KW}}^{\mathsf{wrap},\mathsf{real}}(\eta)$ *and* $\mathbf{Exp}_{\mathcal{A},\mathsf{KW}}^{\mathsf{wrap},\mathsf{fake}}(\eta)$*. In both experiments the adversary can access a number of keys* $k_1, k_2, \ldots, k_n \ldots$ *(which he can ask to be created via a query* NEW*). In his other queries, the adversary refers to these keys via symbols* $K_1, K_2, \ldots, K_n$ *(where the implicit mapping should be obvious). By abusing notation we often use* K_i *as a placeholder for* k_i *so, for example,* $\mathsf{Wrap}_{K_i}^a(K_j)$ *means* $\mathsf{Wrap}_{k_i}^a(k_j)$*. We now explain the queries that the adversary is allowed to make, and how they are answered in the two experiments.*

- $\mathsf{NEW}(K_i)$*: a new key* k_i *is generated via* $k_i \leftarrow \mathsf{KG}(\eta)$
- $\mathsf{ENC}(K_i, a, m)$ *where* $m \in \mathcal{K} \cup \{K_i \mid i \in \mathbb{N}\}$ *and* $h \in \mathcal{H}$*. The experiment returns* $\mathsf{Wrap}_{k_i}^a(m)$*.*
- $\mathsf{TENC}(K_i, a, m)$ *where* $m \in \mathcal{K} \cup \{K_i \mid i \in \mathbb{N}\}$ *and* $a \in \mathcal{H}$*. The real experiment returns* $\mathsf{Wrap}_{k_i}^a(m)$*, whereas the fake experiment returns* $\$^{|\mathsf{Wrap}_{k_i}^a(m)|}$
- $\mathsf{DEC}(K_i, a, c)$*: the real experiment returns* $\mathsf{UnWrap}_{k_i}^a(c)$*, the fake experiment returns* $\perp$*.*
- $\mathsf{CORR}(K_i)$*: the experiment returns* k_i

Correctness of the wrapping scheme requires that for any $k_1, k_2 \in \mathcal{K}$ and any $a \in \mathcal{H}$, if $c \leftarrow Wrap_{k_1}^a(k_2)$ then $Unwrap_{k_1}^a(c) = k_1$.

Consider the directed graph whose nodes are the symbolic keys K_i and in which there is an edge from K_i to K_j if the adversary issues a query $\mathsf{ENC}(K_i, a, K_j)$. We say that a key K_i is corrupt if either the adversary corrupted the key from the start, or if the key is reachable in the above graph from a corrupt key. If a handle, respectively pointer, points to a corrupted key, we call the pointer corrupted as well.

We make the following assumptions on the behaviour of the adversary.

- For all i the query $\mathsf{NEW}(K_i)$ is issued at most once.
- All the queries issued by the adversary contain keys that have already been generated by the experiment.
- The adversary never makes a test query $\mathsf{TENC}(K_i, a, K_j)$ if K_i is corrupted at the end of the experiment.
- If A issues a test query $\mathsf{TENC}(K_i, a, m)$ then A does not issue $\mathsf{TENC}(K_j, a', m')$ or $\mathsf{ENC}(K_j, a', m')$ with $(K_i, a, m) = (K_j, a', m')$
- The adversary never queries $\mathsf{DEC}(K_i, a, c)$ if c was the result of a query $\mathsf{TENC}(K_i, a, m)$ or of a query $\mathsf{ENC}(K_i, a, m)$ or K_i is corrupted.

At the end of the execution the adversary has to output a bit b which is also the result of the experiment. The advantage of adversary A in breaking the key-wrapping scheme KW is defined by:

$$A_{\mathsf{KW},A}^{\mathsf{wrap}}(\eta) = \Big|\Pr\left[b \leftarrow \mathbf{Exp}_{\mathsf{KW},A}^{\mathsf{wrap},\mathsf{real}}(\eta) \ : b = 1\right] - \Pr\left[b \leftarrow \mathbf{Exp}_{\mathsf{KW},A}^{\mathsf{wrap},\mathsf{fake}}(\eta) \ : b = 1\right]\Big|$$

and KW is secure if the advantage of any probabilistic polynomial time algorithm is negligible.

Definition 13 (guaranteeing environment). *Suppose Z is an environment that is rooted at r, and p is a predicate on sequences of (id_0, id_1, m). Let $S_p(Z)$ be a sandbox that runs Z but checks at the end of each activation if the predicate holds true on the list of messages sent and received by the environment (including the message about to be send). If the predicate does not hold true, S_p aborts Z_p and outputs some error symbol $fail \in \Sigma$. We say that Z* guarantees *a predicate p, if there exists such a sandbox $S_p(Z)$, and for every protocol Π rooted at r, for every adversary A, we have that:*

$$\Pr[\text{Exec}[\Pi, A, Z] = fail]$$

is negligible in η.

Let us denote a list of messages m_i from a_i to b_i, as $M^t = ((a_0, b_0, m_0), \ldots, (a_t, b_t, m_t))$. We will denote the i-prefix of this list by M^i. We can filter messages by their session id: $M^i_{|\text{SP}}$ denotes a messages (a_i, b_i, m_i) where either $a_i = <\texttt{env}>$ and b_i is of the form <<reg, *basePID*>, <$\alpha_1, \ldots, \alpha_{k-1}$, <prot-fkm, <SP>>>>, or vice versa. We say (a_j, b_j, m_j) is a response to (a_i, b_i, m_i) if (a_j, b_j, m_j) is the earliest message such that $i < j$, $a_i = b_j$, $b_j = a_i$, and that no other message at an epoch $k < i$ exists such that (a_i, b_i, m_i) is a response to (a_k, b_k, m_k). This assumes that there is a response to every query. (In case of an error, $\mathcal{F}_{\text{KM}}$ responds with $\perp$ rather than ignoring the query.) In order to tell which handles are corrupted, we need to define which handles point to the same key a given moment t.

Given $M_{|NP} = M_{|\mathcal{U},\mathcal{U}^{\text{ext}},\mathcal{ST},Room} = ((a_0, b_0, m_0), \ldots, (a_n, b_n, m_n))$, we define $\equiv^0$ to be the empty relation and for all $1 \leq t \leq n$, we define $\equiv^t$ as the least symmetric transitive relation such that

1. $\equiv^t \subset \equiv^{t-1} \cup \{(U, h), (U, h)\}$, if $m_t = <\texttt{new}^\bullet, \texttt{h}, public>$, $a_t = U$ and $\exists s < t, F, a : m_s = <\texttt{new}, \texttt{F}, \texttt{a}>$ and (a_t, b_t, m_t) is a response to (a_s, b_s, m_s)
2. $\equiv^t \subset \equiv^{t-1} \cup \{(U_1, h_1), (U_2, h_2)\}$, if $m_t = <\texttt{share}^\bullet>$, $a_t = U_1$ and $\exists s < t :\ m_s = <\texttt{share}, (\texttt{U}_1, \texttt{h}_1), (\texttt{U}_2, \texttt{h}_2)>$ and (a_t, b_t, m_t) is a response to (a_s, b_s, m_s)
3. $\equiv^t \subset \equiv^{t-1} \cup \{(U_1, h_1), (U_2, h_2)\}$, if $m_t = <\texttt{unwrap}^\bullet, \texttt{h}_2>$, $a_t = U_2$ and $\exists q, r, s$: such that (a_t, b_t, m_t) is a response to (a_s, b_s, m_s), and (a_r, b_r, m_r) is a response to (a_q, b_q, m_q), and $r < s$. Furthermore: $m_q = <\texttt{wrap}, \texttt{h}_1, \texttt{h}_2, \texttt{id}>, b_q = U_1, m_r = <\texttt{wrap}^\bullet, \texttt{w}>, a_r = U_1$ and $m_s = \texttt{unwrap}, \texttt{h}'_1, \texttt{w}, \texttt{a}, \texttt{F}, \texttt{id}>, b_s = U_1$ and $(U_2, h'_1) \equiv^{t-1} (U_1, h_1)$.
4. $\equiv^t = \equiv^{t-1}$, otherwise.

Using this relation, we define following predicate: $corrupted_{M_{|NP}}(U, h)$ holds iff either some (U^*, h^*), $((U^*, h^*) \equiv^t (U, h))$ were corrupted directly, via wrapping with a corrupted key, or injected via unwrapping, formally:

- $m_j = (\texttt{adv}, \texttt{env}, <\texttt{corrupt}^\bullet, \texttt{h}^*, \texttt{c}>), m_i = (\texttt{env}, U^*, <\texttt{corrupt}, \texttt{h}>) \in M_{|NP}$ and m_j is a response to m_i (for some c), or
- there are $m_j = (U^*, \texttt{env}, <\texttt{wrap}^\bullet, \texttt{w}>), m_i = (\texttt{env}, U^*, <\texttt{wrap}, \texttt{h}_1, \texttt{h}^*, \texttt{id}>) \in M_{|NP}$ and m_j is a response to m_i, with $corrupted_{M_{|NP}}(U^*, h_1)$, or
- there are $m_j = (U^*, \texttt{env}, <\texttt{unwrap}^\bullet, \texttt{h}^*>), m_i = (\texttt{env}, U^*, <\texttt{unwrap}, h_1, w, a_2, F_2, id>) \in M_{|NP}$ and m_j is a response to m_i, while $corrupted_{M_{|NP}}(U^*, h_1)$.

Finally, let *corrupt-before-wrap* be the following predicate on a list of messages $M^t = ((a_0, b_0, m_0), \ldots, (a_t, b_t, m_t))$: for all $i \leq t$ and network parameters $NP = \mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}, Room$, we have

$$corrupted_{M_{|NP}}(U, h) \wedge (\texttt{env}, U, <\texttt{wrap}, \texttt{h}, \texttt{h}'>) \in M^i_{|NP} \Rightarrow corrupted_{M^i_{|NP}}(U, h).$$

Lemma 2. *For any KU parameter $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ and set of sets of PPT algorithms $\overline{\texttt{Impl}}$, let $\mathcal{F}^{\texttt{impl}}_{\text{KM}} := \mathcal{F}^{\mathcal{F}_1/\texttt{Impl}_{\mathcal{F}_1}, \ldots, \mathcal{F}_l/\texttt{Impl}_{\mathcal{F}_l}}_{\text{KM}}$ be the partial implementation of $\mathcal{F}_{\text{KM}}$ with respect to all KU functionalities in $\overline{\mathcal{F}}$. If $KW = (impl^{\texttt{KW}}_{\texttt{new}}, \texttt{wrap}, \texttt{unwrap})$ is a secure and correct key-wrapping scheme (Definition 12) then $\pi_{\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi, \overline{\texttt{Impl}}}$ emulates $\mathcal{F}^{\texttt{impl}}_{\text{KM}}$ for environments that guarantee* corrupt-before-wrap.

Proof. Proof by contradiction: Assuming that there is no adversary Sim such that for all well-behaved environments Z that are rooted at `prot-fkm` and guarantee *corrupt-before-wrap* $\mathrm{Exec}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z] \approx \mathrm{Exec}[\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}, Sim, Z]$ holds, we chose a Sim that basically simulates $\mathcal{F}_{\mathrm{setup}}$ for corrupted users in $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$, and a Z that is indeed able to distinguish $\mathrm{Exec}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$ and $\mathrm{Exec}[\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}, Sim, Z]$. Then, we use it to construct an attacker B_Z against the key-wrapping challenger. B_Z will be carefully crafted, such that *a*) it is a valid adversary *b*) it has the same output distribution in the fake key-wrapping experiment as Z has when interacting with $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ and Sim *c*) it has the same output distribution in the real key-wrapping experiment as Z has when interacting with $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$ and A_D.

Sim defines the same code as the dummy adversary (see [11, §4.7]), but when instructed by the environment to instruct a corrupted party to call $\mathcal{F}_{\mathrm{setup}}$, it simulates $\mathcal{F}_{\mathrm{setup}}$ (because $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ does not define `prot-fsetup`). This means: Sim waits for <ready•,P> from $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ from all parties $P \in \mathcal{U} \cup \mathcal{ST} \cup \hat{\mathcal{F}}$ before operating - for corrupted parties $U_i \in \mathcal{U}$ (security tokens are incorruptible, $U^{\mathrm{ext}} \in \mathcal{U}^{\mathrm{ext}}$ are ignored), it waits be instructed to send ready and simulates the reception of <ready•,P> itself. Afterwards, it accepts instructions to send <m> as U_i to $\mathcal{F}_{\mathrm{setup}}$– in this case, Sim instructs U_i to send m to $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$. Similarly, the response from $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ is simulated to be transmitted via $\mathcal{F}_{\mathrm{setup}}$. When U_i is instructed to send `finish_setup` to $\mathcal{F}_{\mathrm{setup}}$, Sim sends `finish_setup` to $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ instead (and relays the answer), but only if it received ready from all parties $P \in \mathcal{U} \cup \mathcal{ST} \cup \hat{\mathcal{F}}$ before (as we already mentioned), and only the first time.

Given Z, we will now construct the attacker $\mathcal{B}_Z$ against the key-wrapping game in Definition 12. Recall that Z is rooted at `prot-fkm`. This means that Z only calls machines with the same SID and the protocol name `prot-fkm`. In particular, the session parameters $(\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi)$ are the same (see [11, §5.3]), so from now on, we will assume them to be arbitrary, but fixed. The construction of $\mathcal{B}_Z$ aims at simulating Z in communication with the simulator Sim given above and the key-management functionality $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$, but instead of performing wrapping and unwrapping in $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ itself, $\mathcal{B}_Z$ queries the challenger in the wrapping experiment. In case of the fake experiment, the simulation is very close to the network $[\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}, Sim, Z]$, for the case of the real experiment, we have to show that the output is indistinguishable from a network of distributed security tokens and a dummy adversary $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$. This will be the largest part of the proof. $\mathcal{B}_Z$ is defined as follows: $\mathcal{B}_Z$ simulates the network $[\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}, Sim, Z]$, where $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ is defined just as $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$, but `new`, `wrap`, `unwrap` and `corrupt` are altered such that they send queries to the experiment instead. Note that for this reason, $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ is not a valid machine in the GNUC model - we just use it as a convenient way to describe the simulation that $\mathcal{B}_Z$ runs. We assume that the place-holder symbols $K_1, \ldots$ from Definition 12 are distinguishable from other credentials and that there is some way to select those symbols such that each of them is distinct. Furthermore, they should be difficult to guess. One way to achieve this is to implement a pairing of i and j, for $U = U_i$ and $j \leftarrow \{0,1\}^\eta$, using Cantor's pairing function.

`new[ready]`: accept <new,F,a> from $U \in \mathcal{U}$
if <F,new,a,$\star$> $\in \Pi$ then
  if $F \in \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$
    $(k, public) \leftarrow impl^F_{\mathtt{new}}(1^\eta)$
    create h; Store [U,h] $\leftarrow$ <F,a,k>
    $\mathcal{K} \leftarrow \mathcal{K} \cup \{k\}$
    send <new•,h,$public$> to U
  else if $F = \mathtt{KW}$
    create K_i, h
    query $\mathsf{NEW}(K_i)$
    $\mathcal{K} \leftarrow \mathcal{K} \cup \{K_i\}$
    Store [U,h] $\leftarrow$ <KW,a,K_i>
    send <new•,h,> to U

`wrap[finish_setup]`:
accept <wrap,h_1,h_2,id> from $U \in \mathcal{U}$
if Store [U,h_1]=<KW,a_1,c_1> and Store [U,h_2]=<F_2,a_2,c_2>
  and <KW,wrap,a_1,a_2>$\in \Pi$
  if $\exists w$.<c_2,<F_2,a_2,id>,w>$\in$encs[c_1]
    send <wrap•,w> to U

```
  else
    W ← W ∪ {(c1, c2)}
    if c1 ∈ K_cor
     for all c3 reachable from c2 in W
       corrupt c3
     w ← wrap^<F2,a2,id> (key[c1], key[c2])
    else
      w = TENC(c1, <F2, a2, id>, c2)
    encs[c1] ← encs[c1] ∪{ <c2, <F2, a2, id>, w>}
    send <wrap•, w> to U
```

```
unwrap[finish_setup]:
accept <unwrap, h1, w, a2, F2, id> from U ∈ U
if Store[U, h1]=<KW, a1, c1> and <KW, unwrap, a1, a2> ∈ Π, F2 ∈ F̄
  if c1 ∈ K_cor
    c2 = unwrap^<F2,a2,id> (key[c1], w)
    if c2 ≠ ⊥ and c2 ∉ K \ K_cor
      K_cor ← K_cor ∪ {c2}
      create h2
      Store[U, h2] ← <F2, a2, c2>
      key[c2]=c2
      send <unwrap•, h> to U
    else // first bad event
      send <error> to A
  else
    if ∃!c2.<c2, <F2, a2, id>, w> ∈ encs[c1]
      create h2
      Store[U, h2] ← <F2, a2, c2>
      send <unwrap•, h> to U
    else // c1 ∉ K_cor ∧ ¬∃c2.<c2, <F2, a2, id>, w> ∈ encs[c1])
      query c2 = DEC(c1, <F2, a2, id>, w);
      if c2 ≠ ⊥ //second bad event
        halt and output 0.
```

```
if c ∈ K_cor
  send <corrupt•, h, key[c]> to A
else
  K_cor ← K_cor ∪ {c}
  for any Store[U, h]=<F, a, c>
    if F ∈ KW
      query k = CORR(c)
      key[c] ← k
    else
      key[c] ← c
    send <corrupt•, h, key[c]> to A
```

Listing 17: procedure for corrupting a credential c

$\mathcal{B}_Z$ is a valid adversary. We will argue about each assumption on the behaviour of the adversary, one after another:

1. For all i, the query $\mathsf{NEW}(K_i)$ is issued at most once, because we specified $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ to select a new K_i for each such query
2. All queries issued by $\mathcal{B}_Z$ contain keys that have already been generated by the experiment. Observe that all queries are preceeded by a conditional that checks if the argument to the query is in the third position of the store, i. e., there are U, h, a such that $\mathtt{Store}[U, h] =< \mathtt{KW}, a, k >$ at some point of the execution of $\mathcal{B}_Z$. We claim that each

such k has either been generated using NEW or is in $\mathcal{K}_{\text{cor}}$ (in which case no query is made). Proof by contradiction: assume we are at the first point of the execution where such a key is added to the store. The store is only written in the `new` and `unwrap` step. In `new`, a new K_i is created. In `unwrap`, there are three cases in which the store is written to: *a*) If $c_1 \in \mathcal{K}_{\text{cor}}$, then $c_2 \in \mathcal{K}_{\text{cor}}$. Once something is marked as corrupted, it stays corrupted. *b*) If $c_1 \notin \mathcal{K}_{\text{cor}}$, but $\exists \mathtt{c_2}.\ < \mathtt{c_2}, < \mathtt{F_2}, \mathtt{a_2}, id >, \mathtt{w} > \in \mathtt{encs}[\mathtt{c_1}]$). Only `wrap` can write to `encs`, so c_2 must have been in the store before.

3. The adversary never makes a test query $\mathsf{TENC}(K_i, a, K_j)$ if K_i is corrupted at the end of the experiment, because a TENC query is only output in the step `wrap` if $c_1 \notin \mathcal{K}_{\text{cor}}$. The condition *corrupt-before-wrap* enforces that if c_1 is not corrupted at that point, it will never be corrupted. (A detailed analysis about how *corrupt-before-wrap* is correct with respect to the definition if $\mathcal{F}_{\text{KM}}^{\mathtt{impl},\mathsf{KW}}$ is left to the reader.)
4. If $\mathcal{B}_Z$ issues a test query $\mathsf{TENC}(K_i, a, m)$ then $\mathcal{B}_Z$ does not issue $\mathsf{TENC}(K_j, a', m')$ or $\mathsf{ENC}(K_j, a', m')$ with $(K_i, a, m) = (K'_j, a, m')$, since $\mathcal{B}_Z$ never issues ENC queries at all and only issues TENC queries if the same combination of (K_i, a, m) was not stored in `encs` before. Every time TENC is called, `encs` is updated with those parameters.
5. $\mathcal{B}_Z$ never queries $\mathsf{DEC}(K_i, a, c)$ if c was the result of a query $\mathsf{TENC}(K_i, a, m)$ or of a query $\mathsf{ENC}(K_i, a, m)$ or K_i is corrupted, because *a*) TENC queries are stored in `encs` and the step `unwrap` checks this variable before querying DEC, *b*) `enc` queries are never issued, *c*) if a credential c_1 inside the `unwrap` step is corrupted, the query DEC is not issued.

We conclude that $\mathcal{B}_Z$ fulfills the assumptions on the behaviour of the adversary expressed in Definition 12.

$\mathcal{B}_Z$ simulates $\mathcal{F}_{\text{KM}}^{\mathtt{impl},\mathsf{KW}}$ in the fake experiment.

$\mathcal{B}_Z$ is defined to be a simulation of the network $\left[\mathcal{F}_{\text{KM}}^{\mathtt{impl},\mathsf{KW}}, Sim, Z\right]$, where $\mathcal{F}_{\text{KM}}^{\mathtt{impl},\mathsf{KW}}$ is $\mathcal{F}_{\text{KM}}^{\mathtt{impl}}$, except for the altered steps `new`, `wrap`, `unwrap` and `corrupt`. We claim that, in the fake experiment, those alterations do not change the input/output behaviour. First, we will simplify the `unwrap` step in $\mathcal{F}_{\text{KM}}^{\mathtt{impl}}$:

```
1  unwrap[finish_setup]:
2  accept <unwrap, h1, w, a2, F2, id>
3                    from U ∈ 𝒰
4  if  Store[U, h1]=<KW, a1, c1> and
5     <KW, unwrap, a1, a2>∈ Π,F2 ∈ F̄
6    if  c1 ∈ 𝒦cor
7      c2 = unwrap^<F2,a2,id>(c1, w)
8      if  c2 ≠ ⊥ and c2 ∉ 𝒦
9        𝒦cor ← 𝒦cor ∪ {c2}
10       create h2
11       Store[U, h2] ← <F2, a2, c2>
12       key[c2]=c2
13       send <unwrap•, h> to U
14     else  if  c2 ≠ ⊥, c2 ∈ 𝒦 and c2 ∈ 𝒦cor
15       create h2
16       Store[U, h2] ← <F2, a2, c2>
17       send <unwrap•, h> to U
18     else  //  (c2 = ⊥ ∨ c2 ∈ 𝒦 \ 𝒦cor)
19       send <error> to A
20   else  if  ( c1 ∉ 𝒦cor and
21      ∃!c2.<c2, <F2, a2, id>,w>∈encs[c1])
22     create h2
23     Store[U, h2] ← <F2, a2, c2>
24     send <unwrap•, h> to U
```

We first observe that it would not make a difference if the code in the branch at Line 14 would execute the same code as in the branch at Line 8, i. e., additionally perform the computations in Line 9 and 12, since from the definition of the steps `unwrap` and the corruption procedure, if $c_2 \in \mathcal{K}_{\text{cor}}$, then already $\mathtt{key}[c_2] = c_2$. This means that Lines 8 to

13 are executed if $c_2 \neq \bot \wedge c_2 \notin \mathcal{K} \setminus \mathcal{K}_{\text{cor}}$, otherwise a bad event is produced, i. e., an error is send to the adversary. For reference, this is the equivalent, simpler code:

```
1  unwrap[finish_setup]:
2  accept <unwrap, h_1, w, a_2, F_2, id>
3                  from U ∈ 𝒰
4  if Store[U, h_1]=<KW, a_1, c_1> and
5     <KW, unwrap, a_1, a_2>∈ Π, F_2 ∈ F̄
6    if c_1 ∈ 𝒦_cor
7      c_2 = unwrap^{<F_2,a_2,id>}(c_1, w)
8      if c_2 ≠ ⊥ and c_2 ∉ 𝒦 \ 𝒦_cor
9        𝒦_cor ← 𝒦_cor ∪ {c_2}
10       create h_2
11       Store[U, h_2] ← <F_2, a_2, c_2>
12       key[c_2]=c_2
13       send <unwrap•, h> to U
14     else // (c_2 = ⊥ ∨ c_2 ∈ 𝒦 \ 𝒦_cor)
15       send <error> to A
16   else if ( c_1 ∉ 𝒦_cor and
17      ∃!c_2.<c_2, <F_2, a_2, id>, w>∈encs[c_1])
18     create h_2
19     Store[U, h_2] ← <F_2, a_2, c_2>
20     send <unwrap•, h> to U
```

We claim that the following invariant holds: For all Z, and at the end of every epoch [11, §5.3], i. e., after each activation of Z:

- the distribution of the input received by Z (the view of Z) is the same for Z in $\left[\mathcal{F}_{\text{KM}}^{\texttt{impl}}, Sim, Z\right]$ as well as in the network $\left[\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}, Sim, Z\right]$ simulated by $\mathcal{B}_Z$.
- in the same two execution, the entry $\texttt{Store}[U, h] =< \texttt{KW}, a, c >$ exists in $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$ if, and only if, the entry $\texttt{Store}[U, h] =< \texttt{KW}, a, \overline{c} >$ exists in $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$, where the following holds for c and $\overline{c}$
  - $c \in \mathcal{K} \leftrightarrow \overline{c} \in \mathcal{K}$ and $c \in \mathcal{K}_{\text{cor}} \leftrightarrow \overline{c} \in \mathcal{K}_{\text{cor}}$
  - if $c \in \mathcal{K}$, then $\overline{c}$ is the key drawn for c in the NEW step. Furthermore, if $c \in \mathcal{K}_{\text{cor}}$, in $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$ $\texttt{key}[c] = \overline{c}$ and in $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$ $\texttt{key}[\overline{c}] = \overline{c}$
  - if $c \in \mathcal{K}_{\text{cor}} \setminus \mathcal{K}$ (i. e., c was injected), $c = \overline{c}$ and $\texttt{key}[c] = c$ in both $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$ $\texttt{key}[c] = \overline{c}$ and $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$.

The only relevant steps are the altered steps `new`, `wrap`, `unwrap`, and the corruption macro.

- `corrupt`: For honestly generated wrapping keys $c \in \mathcal{K}$, by IH, the corruption procedure outputs the key generated with NEW in $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$, respectively the equally drawn key in case of $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$. For dishonestly generated keys, also by IH, both output the same keys. The second condition holds by definition of this step and IH.
- `new`: From the definition of the fake experiment follows that the first condition holds. The second condition holds since the freshly created c is added to $\mathcal{K}$.
- `wrap`: By definition of the fake experiment, ENC and TENC substitute credentials c in $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$. If $c_1 \notin \mathcal{K}_{\text{cor}}$, then $c_1 \in \mathcal{K}$ (as otherwise it would not be in the database). Thus, in $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$, TENC will perform this substitution for both c_1 and c_2, resulting in the same output that $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$ produces. If $c_1 \in \mathcal{K}_{\text{cor}}$, then, first, by definition of this step in $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$ $c_2 \in \mathcal{K}_{\text{cor}}$ at the point the unwrap function is computed, and therefore, $\texttt{key}[c_1]$ in $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$ contains $\overline{c}_1$ (similar for c_2). Therefore, the same value is computed and output in $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$ and $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$. The second condition holds because of the first condition of the IH and the fact that both functions call `corrupt` on c_2 if $c_1 \in \mathcal{K}_{\text{cor}}$.
- `unwrap`: Since for $c_1 \notin \mathcal{K}_{\text{cor}}$ (by definition of the fake experiment), DEC produces $\bot$, i. e., the conditional marked "second bad event" never evaluates. Note furthermore, that this step is the same in $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$ and $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$, only that the call to the unwrap function in case $c_1 \in \mathcal{K}_{\text{cor}}$ substitutes c_1 by $\texttt{key}(c_1)$ in $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$. This is correct by the second condition of the IH. Therefore, the first and the second condition are preserved for the next step.

We can conclude that

$$\mathbf{Exp}^{\mathsf{wrap},\mathsf{fake}}_{\mathsf{KW},\mathcal{B}_Z}(\eta) = \mathrm{Exec}[\mathcal{F}^{\mathtt{impl}}_{\mathrm{KM}}, Sim, Z](\eta).$$

$\mathcal{B}_Z$ simulates $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$ in the real experiment.

In the fake experiment, it is not possible that $\mathcal{B}_Z$ halts at the end of the `unwrap` step (marked "second bad event"), since DEC always outputs $\perp$. Thus, the probability that $\mathcal{B}_Z$ halts at the "second bad event" mark whilst in the real experiment must be negligible, as this would contradict the assumption that KW is a secure wrapping scheme right here. The representation of the this part of the proof benefits from altering $\mathcal{B}_Z$ such that instead of halting, $\mathcal{B}_Z$ continues to run $\mathcal{F}^{\mathtt{impl},\mathsf{KW}}_{\mathrm{KM}}$, by running the following code:

create h_2; Store`[`U`,`h_2`]` ← <F_2`,`a_2`,`c_2>
send <`unwrap`$^\bullet$`,`h> to U

We further modify the code of $\mathcal{B}_Z$ by removing the conditional marked "first bad event". Here as well, the probability that this conditional evaluates to true is negligible, following from the assumption that KW is a secure wrapping scheme. Proof by contradiction: if $\mathcal{B}_Z$ could produce this conditional, then he knows a wrapping w such that $unwrap^a(k_1, w) = c_2$ for $c_2 \in \mathcal{K}$ and $c_2 \notin \mathcal{K}_{\mathrm{cor}}$. Since $c_1 \in \mathcal{K}_{\mathrm{cor}}$ and $k_1 = \mathtt{key}[\mathtt{c_1}]$, $\mathcal{B}_Z$ either knows k_1 (since it injected it, i. e., $c_1 \notin \mathcal{K}$), or can learn by corrupting it (if $c_1 \in \mathcal{K}$). If there was a path in $\mathcal{W}$ from c_1 to c_2, it would already be corrupted, so the attacker can learn k_1 while $c_2 \notin \mathcal{K}_{\mathrm{cor}}$. He can use k_1 as to decrypt w himself and learn c_2. But he should not be able to learn c_2 in the fake experiment, since it is randomly drawn and did never appear in any output. (The adversary can check if it guessed c_2 correctly, for example, by requesting a wrapping of a third, corrupted wrapping key under c_2 via $\mathcal{F}^{\mathtt{impl},\mathsf{KW}}_{\mathrm{KM}}$ and then calling DEC to check if the received wrapping (using the same argument) decrypts to the corrupted and thus known key.) Since this happens only with neglible probability, it can only happen with negligible probability in the real experiment, too, as otherwise the assumption that KW is a secure wrapping scheme would be contradicted.

Therefore, we perform those modifications and call the sightly different attacker $\mathcal{B}'_Z$. Since those parts of the code are only executed with negligible probability, we have that

$$\begin{aligned}&\Pr[b \leftarrow \mathbf{Exp}^{\mathsf{wrap},\mathsf{real}}_{\mathsf{KW},\mathcal{B}_Z}(\eta) : b = 1]\\ &\quad - \Pr[b \leftarrow \mathbf{Exp}^{\mathsf{wrap},\mathsf{real}}_{\mathsf{KW},\mathcal{B}'_Z}(\eta) : b = 1]\end{aligned}$$

is negligible in η.

Fix an arbitrary security parameter η. Then, let $View_{\mathcal{B}'_Z}(t)$ be the view of Z, i. e., the distribution of messages it is simulated to send to the protocol machine or the adversary, in the tth step of the simulation of $\mathcal{B}'_Z$ in the real experiment. Furthermore, let $Store_{\mathcal{B}_Z;}(t)$ be the distribution of the variable `Store` within the simulated machine <`ideal`,`sid`>, i. e., $\mathcal{F}^{\mathtt{impl},\mathsf{KW}}_{\mathrm{KM}}$, but with the following substitution that affects the wrapping keys: every entry <`KW`,a,K_i> in the variable `Store`$[U,h]$ for some U and h is substituted by an entry <`KW`,a,k_i>, where k_i is the key that the key-wrapping experiment associates to K_i, denoted in the following by $k(K_i)$. If K_i was not created by the key-wrapping experiment, it is left untouched.

We denote the view of Z in the execution of the network $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$ by $View_\pi$ and the distribution of the union of all `Store` variables of all security tokens $ST_1, \ldots, ST_n$ in the network as $Store_\pi$. The union of those variables is well defined, because the first element of each key-value of this table is different for all ST. A step t is an epoch [11, §5.3], i. e., it begins with an activation of Z and ends with the next activation.

We define the following invariant, which will allow us to conclude that $\mathcal{B}'_Z$ has the same output distribution as $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$. For each number of steps t, the following three conditions hold:

- *state consistency (s.c.)* $Store_{\mathcal{B}_Z;}$ (t) and $Store_\pi$ are equally distributed
- *output consistency (o.c.)* $View_{\mathcal{B}'_Z}$ (t) and $View_\pi$ are equally distributed
- *phase consistency (p.c.)* The probability that the flag `ready` is set in $\mathcal{F}^{\mathtt{impl},\mathsf{KW}}_{\mathrm{KM}}$ in $\mathbf{Exp}^{\mathsf{wrap},\mathsf{real}}_{\mathsf{KW},\mathcal{B}'_Z}$ equals the probability that `ready` is set in $\mathcal{F}_{\mathrm{setup}}$ in $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$. Furthermore, the probability that the flag `setup_finished` is set in $\mathcal{F}^{\mathtt{impl},\mathsf{KW}}_{\mathrm{KM}}$ in $\mathbf{Exp}^{\mathsf{wrap},\mathsf{real}}_{\mathsf{KW},\mathcal{B}'_Z}$ equals the probability that `setup_finished` is set in all $ST \in \mathcal{ST}$. .

If $t = 0$, the protocol has not been activated, thus there was no output, and not state changes. The invariant holds trivially. If $t > 0$, we can assume that s.c., o.c. and p.c. were true at the end of the preceding epoch. Note that Z is restricted to addressing top-level parties with the same `sid`. In particular, it cannot address $\mathcal{F}_{\text{setup}}$ directly (but it can corrupt a user to do this). Since the `sid` has to have a specific format that is expected by all machines in both networks, we assume `sid` to encode $\mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}, Room$. Case distinction over the recipient and content of the message that Z sends at the beginning of the next epoch:

1. Z sends a message to $ST_i \in \mathcal{ST}$, and
   (a) the message is <ready>: In $\mathbf{Exp}^{\mathsf{wrap,real}}_{\mathsf{KW},\mathcal{B}'_Z}$, $\mathcal{F}_{\text{KM}}^{\mathtt{impl},\mathsf{KW}}$ records ready–ST_i and sends $<$ `ready`$^\bullet$, $ST_{\mathtt{i}} >$ to Sim, if the message is recorded for the first time. Sim behaves just like A_D in this case. If all other $ST_j \in \mathcal{ST}$ and all $U \in \mathcal{U}$ have sent this message before, the flag `ready` is set to true.
   In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$, ST_i accepts the message and forwards it (as <ready>) to $\mathcal{F}_{\text{setup}}$. Then, $\mathcal{F}_{\text{setup}}$ records ready–ST_i and sends $<$ `ready`$^\bullet$, $ST_{\mathtt{i}} >$ to A_D, if the message is recorded for the first time. If all other $ST_j \in \mathcal{ST}$ and all $U \in \mathcal{U}$ have sent this message before, the flag `ready` is set to true. We see that p.c. and o.c. hold. S.c. holds trivially, because the Store did change in neither execution.
   (b) the message is of any other form: in $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$, ST accepts no other message from the environment. In $\mathbf{Exp}^{\mathsf{wrap,real}}_{\mathsf{KW},\mathcal{B}'_Z}$, $\mathcal{F}_{\text{KM}}^{\mathtt{impl},\mathsf{KW}}$ ignores any other message coming from ST_i, too. So p.c., o.c. and s.c. hold.
2. Z sends a message to $U_i \in \mathcal{U}$:
   In $\mathbf{Exp}^{\mathsf{wrap,real}}_{\mathsf{KW},\mathcal{B}'_Z}$, $\mathcal{F}_{\text{KM}}^{\mathtt{impl},\mathsf{KW}}$ will receive this message, and treat it depending on its form (if its flag `ready` is set).
   In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$, U_i will relay this message m in the form $<m, ST_i>$ to $\mathcal{F}_{\text{setup}}$, who in turn will send m to ST_i, (if its flag `ready` is set).
   (a) Let m be <ready>: If the `ready` flag has not been set before, and U_i is the last party in $\mathcal{U} \cup \mathcal{ST}$ that has not sent this message yet, $\mathcal{F}_{\text{KM}}^{\mathtt{impl},\mathsf{KW}}$ in $\mathbf{Exp}^{\mathsf{wrap,real}}_{\mathsf{KW},\mathcal{B}'_Z}$ will set the `ready` flag, otherwise it will not. The same holds for $\mathcal{F}_{\text{setup}}$ in $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$. Therefore, we have p.c. In both cases, Sim, respectively A_D, forwards the acknowledgement to the environment (after recording the state change). Thus, s.c. and o.c. hold trivially.
   For the following cases, assume `ready` to be set in both $\mathbf{Exp}^{\mathsf{wrap,real}}_{\mathsf{KW},\mathcal{B}'_Z}$ and $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$. If it is unset in one of them, by induction hypothesis, it is unset in both. If it is not set, any other message will not be accepted by neither $\mathcal{F}_{\text{KM}}^{\mathtt{impl},\mathsf{KW}}$, nor $\mathcal{F}_{\text{setup}}$ (thus never reach ST_i). Therefore, in the following cases we will assume `ready` to be set in $\mathcal{F}_{\text{KM}}^{\mathtt{impl},\mathsf{KW}}$ and $\mathcal{F}_{\text{setup}}$, i. e., $\mathcal{F}_{\text{setup}}$ delivering commands that U_i receives from the environment to ST_i.
   (b) Let m=<new, F, a> : If $F \neq$ `KW`, the same code is executed (except for the step adding adding the freshly created credential to $\mathcal{K}$), p.c.,s.c. and o.c. hold trivially. Assume $F =$ `KW` and <KW, new, a, $\star$>$\in \Pi$ (otherwise, $\perp$ is output in both executions). $\mathcal{F}_{\text{KM}}^{\mathtt{impl},\mathsf{KW}}$ draws a new K_i and call NEW to create the key. K_i is created, just like handles, in a way that makes sure it is unique. Therefore, since throughout $\mathbf{Exp}^{\mathsf{wrap,real}}_{\mathsf{KW},\mathcal{B}'_Z}$, K_i is always substituted for the same key, there is a function mapping K_i to the key k_i created by the experiment, and this function is injective. Note that, by the definition of $Store_{\mathcal{B}_Z;}$, $Store_{\mathcal{B}_Z;}(t)$ is $Store_{\mathcal{B}_Z;}(t-1)$ with an additional entry `KW`, a, k_i at $[U, h]$, where k_i is distributed according to KG. In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$, ST_i calls the key-generation directly ($impl^{KW}_{\mathtt{new}}$ calls KG, adding nothing but an empty public part). The output in both cases is <new$^\bullet$, h, > for an equally distributed h. Thus, o.c. holds. $Store_\pi$ is $Store_\pi(t-1)$ with an additional entry `KW`, a, k_i at $[U, h]$ where k_i is distributed according to the same KG as above. Therefore, s.c. holds. (P.c. holds trivially.)
   (c) Let m=<share, h_i, U_j> : In $\mathbf{Exp}^{\mathsf{wrap,real}}_{\mathsf{KW},\mathcal{B}'_Z}$, assuming $U_i, U_j \in Room$, $\mathcal{F}_{\text{KM}}^{\mathtt{impl},\mathsf{KW}}$ outputs <share$^\bullet$, h_j>, and $Store_{\mathcal{B}_Z;}(t)$ is $Store_{\mathcal{B}_Z;}(t-1)$ extended by a copy of its entry $[U_i, h_i]$ at $[U_j, h_j]$. In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$, ST_i checks the same conditions, which by p.c. have an equal probability of success, implicitly: it sends the content of it's store at $[U_i, h_i]$ to $\mathcal{F}_{\text{setup}}$, which verifies $U_i, U_j \in Room$. If this is not the case, $\mathcal{F}_{\text{setup}}$ sends $\perp$ to ST_i, which sends this to the environment (via $\mathcal{F}_{\text{setup}}$), behaving just like $\mathbf{Exp}^{\mathsf{wrap,real}}_{\mathsf{KW},\mathcal{B}'_Z}$. If the condition is met, ST_i sends the content of the store at $[U_i, h_i]$ to $\mathcal{F}_{\text{setup}}$, who delivers this information to ST_j, which in the next step extends $Store_\pi(t-1)$ by a copy of its entry $[U_i, h_i]$ at $[U_j, h_j]$. Thus, s.c. holds. Both output <share$^\bullet$, h_j> upon success, so o.c. holds, too. p.c. holds trivially.

(d) Let m=<finish_setup> : By p.c., we have that $\mathbf{Exp}^{\mathsf{wrap,real}}_{\mathsf{KW},\mathcal{B}'_Z}$ and $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$ either both have finish_setup set, or none has. If both have it set, both output $\perp$ and do nothing.

Assume none has finish_setup set and both ready. In **Exp**, $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ sets the flag finish_setup and responds. In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, U_i sends <finish_setup> to $\mathcal{F}_{\mathrm{setup}}$, which in turn, instead of forwarding it to ST_i like for the majority of commands, sends <close> to every single $ST_j \in \mathcal{ST}$, accepting the response (and thus taking control) after each of those have set the finish_setup flag. By the time $\mathcal{F}_{\mathrm{setup}}$ finishes this step, and hands communication over to U_i, which forwards finish_setup to the environment, every ST_i has left the setup phase. We see that p.c. and o.c. are preserved.

(e) Let m=<C, h, m> : $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ and ST_i execute the same code on their inputs, so by s.c., the invariant is preserved.

(f) Let m=<corrupt, h> : ST_i outputs the credential. $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ does the same, except for wrapping keys. It substitutes the credential by the output of CORR, i. e., $k = k(c)$. By definition of $Store_\pi$ as s.c., $< \mathtt{KW}, \mathtt{a}, \mathtt{k} > \in Store_\pi[U_i, h]$ with the same probability as $< \mathtt{KW}, \mathtt{a}, \mathtt{c} > \in Store_{\mathcal{B}_Z;}[U_i, h]$, thus the output is equally distributed. S.c. and p.c. hold trivially.

(g) Let m=<wrap, h_1, h_2, id> : For this case and the following case, observe that $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ initialises $\mathtt{key}[c]$ only at steps wrap, unwrap and corrupt.

It either contains the output of a query CORR, thus $k(c)$, or the same value as c, if it is defined. It is defined whenever $c \in \mathcal{K}_{\mathrm{cor}} \cap \mathcal{K}$, because if c is added to $\mathcal{K}_{\mathrm{cor}}$ at step corrupt, the response is written to $\mathtt{key}[c]$, and if a $c_3 \notin \mathcal{K}_{\mathrm{cor}}$ is found during step wrap, the condition *corrupt-before-wrap* must have been violated by Z: if such a c_3 is reachable from c_1, without loss of generality, assume it to have minimal distance from c_1 in $\mathcal{W}$. Then, second-before last node on this path is in $\mathcal{K}_{\mathrm{cor}}$, as the distance would not be minimal otherwise. By definition of the step wrap, this node could not have been wrapped without adding it to $\mathcal{K}_{\mathrm{cor}}$, therefore this node was corrupted after it was used to create this wrapping. If $c \in \mathcal{K}_{\mathrm{cor}}$, but $c \notin \mathcal{K}$, then $\mathtt{key}[c] = c$ (see step unwrap).

Assume now ST_i and $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ both have finish_setup set, as otherwise either p.c. was violated in the previous step, or both would output $\perp$ and trivially satisfy the invariant. (This argument is valid for each of the following sub-cases, but the last one).

Both machines check the same conditions on the Store and the policy. ST_i computes $w = \mathsf{wrap}^{<F_2,a_2,id>}(c_1, c_2)$ on the values $< \mathtt{KW}, a_1, c_1 >$ and $< F_2, a_2, c_2 >$ at $[U_i, h_1]$ and $[U_i, h_2]$ in $Store_\pi$.

$\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ performs a case distinction, but we will show that in each cases, it outputs the same value. If $< \mathtt{c_2}, < \mathtt{F_2}, \mathtt{a_2}, id >, \mathtt{w} > \in \mathtt{encs}[\mathtt{c_1}]$, then by observing that encs is only written at the end of this function, we see that p.c. would have been violated in an earlier step, if the output now was differently distributed then the output in $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$.

If $c_1 \in \mathcal{K}_{\mathrm{cor}}$, then $c_2 \in \mathcal{K}_{\mathrm{cor}}$, too. Assume $c_1, c_2 \in \mathcal{K}$. Then, since $\mathtt{key}(c_1) = k(c_1)$ (and $\mathtt{key}(c_2) = k(c_2)$ in case $F_2 = \mathtt{KW}$), the output is $w = \mathsf{wrap}^{<F_2,a_2,id>}(k(c_1), c_2))$ (or $w = \mathsf{wrap}^{<F_2,a_2,id>}(k(c_1), k(c_2))$), which preserves o.c., since s.c. from the last step guarantees that $< \mathtt{KW}, \mathtt{a}, \mathtt{c} > \in Store_\pi[U_i, h] = < \mathtt{KW}, \mathtt{a}, \mathtt{k(c)} > \in Store_{\mathcal{B}_Z;}[U_i, h]$ for $c = c_1$ and $c = c_2$, in case c_2 is a wrapping key. By definition of the real experiment, it performs the same substitutions in case $c_1 \notin \mathcal{K}_{\mathrm{cor}}$, so the same argument can be applied. In case that $c_1 \notin \mathcal{K}$, or $c_2 \notin \mathcal{K}$, the substitution performed is the identity, as key[] refers to the the same value, therefore the same output is produced in $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ and $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$.

Therefore, the output is equally distributed in all three cases, assuming that s.c. was true for the previous step. s.c. and p.c. hold trivially.

(h) Let m=<unwrap, h_1, w, a_2, F_2, id> : In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, if policy and store allow, i. e., $< F_1, a_1, c_1 > \in Store_\pi(U_i, h_1)$, ST_i writes $< F_2, a_2, \mathsf{unwrap}^{<F_2,a_2,id>}(c_1, w) >$ at a fresh place $[U_i, h]$ in $Store_\pi$, unless unwrap returned $\perp$.

$\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ chooses U_i and a new h exactly the same way. By s.c., we have, with equal probability, that $< F_1, a_1, c_1 > \in Store_{\mathcal{B}_Z;}(U_i, h_1)$. Since $F_1 = \mathtt{KW}$, we will use $\hat{c}_1$ for the actual value in $\mathcal{B}_Z$'s store before substitution, so $k(\hat{c}_1) = c_1$.

- If $\hat{c}_1 \in \mathcal{K}_{\mathrm{cor}}$ and $\hat{c}_1 \notin \mathcal{K}$, we have that $c_1 = \hat{c}_1$, and that $\mathtt{key}[\hat{c}_1] = \hat{c}_1$ (only in step unwrap, a key can be added to the store that is not in $\mathcal{K}$). Thus $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ writes $< F_2, a_2, \mathsf{unwrap}^{<F_2,a_2,id>}(k(\hat{c}_1), w)$.

- If $\hat{c}_1 \in \mathcal{K}_{\text{cor}}$ and $\hat{c}_1 \in \mathcal{K}$, $k(\hat{c}_1) = c_1$. Since a key in $\mathcal{K}_{\text{cor}}$ but not $\mathcal{K}$ must have been added using the corruption procedure, we have that $\texttt{key}[\hat{c}_1] = c_1 = k(\hat{c}_1)$. Thus, $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$ writes $< F_2, a_2, \mathsf{unwrap}^{<F_2,a_2,id>}(k(\hat{c}_1), w)$.
- If $\hat{c}_1 \notin \mathcal{K}_{\text{cor}}$ and w was recorded earlier, inspection of $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$ shows that `encs` is written to only at the `wrap` step, which implies that $w = \mathsf{wrap}^{<F_2,a_2,id>}(k(\hat{c}_1), c_2)$ for some c_2. From the correctness of the scheme, we conclude that $< F_2, a_2, c_2 = \mathsf{unwrap}^{<F_2,a_2,id>}(k(\hat{c}_1), w) >$ is written to this position.
- If $\hat{c}_1 \notin \mathcal{K}_{\text{cor}}$ and w is not recorded earlier, by definition of DEC, $< F_2, a_2, \mathsf{unwrap}^{<F_2,a_2,id>}(k(\hat{c}_1), w)$ is written. (Same argument as in the first case, follows from s.c.)

(i) Let m=<`attr_change`, h, a'> : The same code is executed in $\mathbf{Exp}_{\mathsf{KW},\mathcal{B}'_Z}^{\mathsf{wrap,real}}$ and $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, thus p.c., s.c., and o.c. hold trivially.

(j) Let m=<C, $public$, m> : In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, U_i and $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$ perform the same computations, thus p.c., s.c., and o.c. hold trivially.

3. Z sends a message to $U^{\text{ext}} \in \mathcal{U}^{\text{ext}}$

   $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$, as well as U^{ext} only accept messages of the form <C, $public$, m> for $C \in \mathcal{C}_{i,pub}$. Both perform the same computations, thus p.c., s.c., and o.c. hold trivially.

4. Z sends a message to <`adv`>.

   Both A_D and Sim ignore messages that are no instruction. So we can assume that Z instructs the adversary to send a message to some party.

   (a) Assume Z instructs <`adv`> to send a message to a corrupted party, namely:

      i. $U_i \in \mathcal{U}$: U_i can only be addressed by the adversary if it was corrupted before, as otherwise it has never sent a message to the adversary. Note that the code run by U_i in $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, as well as in $\mathbf{Exp}$, does not depend on any internal state. U_i can only talk to the environment, the adversary (Sim acts like A_D in this case) and it can call $\mathcal{F}_{\text{setup}}$, which Sim has to simulate in $\mathbf{Exp}_{\mathsf{KW},\mathcal{B}'_Z}^{\mathsf{wrap,real}}$. Sim is described above and receives all the information necessary to simulate it, that is: <`ready`$^\bullet$, P>, when a protocol party in $\mathcal{U} \cup ST$ receives <`ready`> from the environment, <`finish_setup`$^\bullet$>, when a protocol party in $\mathcal{U}$ receives <`ready`> from the environment, and all messages that $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$ sends to the corrupted U_U (and $\mathcal{F}_{\text{setup}}$ would need to relay). Thus, if p.c. holds in the previous step, the invariant is preserved in case that the message is <`ready`, U_i> or<`finish_setup`>. A message of form <`send`, `...`> is ignored by $\mathcal{F}_{\text{setup}}$ and Sim, so the invariant is trivially preserved here. The communication relayed in the steps `relay_receive` and `relay_send` in $\mathcal{F}_{\text{setup}}$ is simulated as described above and thus falls back to case 2.
      ii. $U_i^{\text{ext}} \in \mathcal{U}^{\text{ext}}$: Like in the previous case, only that $\mathcal{F}_{\text{setup}}$ ignores messages from U_i^{ext}, which Sim simulates correctly.
      iii. other parties cannot become corrupted

   (b) Assume Z instructs <`adv`> to send a message to a party that cannot be corrupted, but that addressed <`adv`> before, i. e., $ST \in \mathcal{ST}$ or $\mathcal{F}_{\text{setup}}$. Since both ST and $\mathcal{F}_{\text{setup}}$ are specified to ignore messages in this case, Sim can simply mask their presence by reacting like ST or $\mathcal{F}_{\text{setup}}$ react upon reception of an unexpected message: answer with $\perp$.

We conclude that the invariant is preserved for an arbitrary number of steps. Since output consistency implies that Z has an identical view, the distribution of Z's output is the same in both games. Thus:

$$\Pr[b \leftarrow \text{Exec}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta) : b = 1] = \Pr[b \leftarrow \mathbf{Exp}_{\mathsf{KW},\mathcal{B}'_Z}^{\mathsf{wrap,real}}(\eta) : b = 1]$$

and therefore:

$$
\begin{aligned}
&|\Pr[b \leftarrow \mathrm{Exec}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z](\eta) : b = 1] \\
&\qquad - \Pr[b \leftarrow \mathrm{Exec}[\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}, Sim, Z](\eta) : b = 1]| \\
&= |\Pr[b \leftarrow \mathbf{Exp}^{\mathsf{wrap},\mathsf{fake}}_{\mathsf{KW},\mathcal{B}_Z}(\eta) : b = 1] \\
&\quad - \Pr[b \leftarrow \mathbf{Exp}^{\mathsf{wrap},\mathsf{real}}_{\mathsf{KW},\mathcal{B}'_Z}(\eta) : b = 1]| \\
&> |\Pr[b \leftarrow \mathbf{Exp}^{\mathsf{wrap},\mathsf{fake}}_{\mathsf{KW},\mathcal{B}_Z}(\eta) : b = 1] \\
&\quad - \Pr[b \leftarrow \mathbf{Exp}^{\mathsf{wrap},\mathsf{real}}_{\mathsf{KW},\mathcal{B}_Z}(\eta) : b = 1]| - \epsilon(\eta),
\end{aligned}
$$

where ϵ is negligible in η. This contradicts the indistinguishability of $\mathrm{Exec}[\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}, Sim, Z]$ and $\mathrm{Exec}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$ and thus concludes the proof.

E The signature functionality

The digital signature functionality is designed after the one described in [21] and detailed in Listing 18. It is parametrized by three algorithms KG, sign and verify. It expects the session parameter to encode a machine id P, and implements $\mathcal{C}^{priv} = \{\mathtt{sign}\}$ and $\mathcal{C}^{pub} = \{\mathtt{verify}\}$.

`new`: accept <`new`> from P
  $(sk, vk) \leftarrow \mathsf{KG}(1^\eta)$; (credential,<`ignore`>) $\leftarrow \mathsf{KG}(1^\eta)$;
  $L \leftarrow L \cup \{$ (credential,sk,vk) $\}$; send <`new`$^\bullet$, `credential`, vk> to P
`sign`: accept <`sign`, `credential`, m> from P
  if (credential , sk,vk)$\in$L for some key
    $\sigma \leftarrow \mathsf{sign}(sk, m)$
    if $\mathsf{verify}(vk, m, \sigma) \neq \bot \wedge sk \notin \mathcal{K}_{\mathrm{cor}}$
      signs[vk] = signs[vk] $\cup\{(m, \sigma)\}$
    else $\sigma \leftarrow \bot$
    send <`sign`$^\bullet$, σ> to P
`verify`: accept <`verify`, vk, <m, σ>> from P
  $b \leftarrow \mathsf{verify}(vk, m, \sigma)$
  if $\exists c, sk.$ (c,sk,vk)$\in$L and $sk \notin \mathcal{K}_{\mathrm{cor}}$ and $b = 1$ and $\nexists \sigma' : (m, \sigma') \in$signs[$vk$] or $b \notin \{0, 1\}$
    $b \leftarrow \bot$
  send <`verify`$^\bullet$, b> to P
`corrupt`: accept <`corrupt`, `credential`> from P
  if (credential , sk,vk)$\in$L for some sk, vk
    $\mathcal{K}_{\mathrm{cor}} \leftarrow \mathcal{K}_{\mathrm{cor}} \cup \{sk\}$; send <`corrupt`$^\bullet$, sk> to A
`inject`: accept <`inject`, <sk, vk>> from P
  (c,<`ignore`>) $\leftarrow \mathsf{KG}(1^\eta)$; $\mathcal{K}_{\mathrm{cor}} \leftarrow \mathcal{K}_{\mathrm{cor}} \cup \{sk\}$;
  $L \leftarrow L \cup \{(c, sk, vk)\}$; send <`inject`$^\bullet$, c> to parentId

Listing 18: A signature functionality $\mathcal{F}_{\mathrm{SIG}}$

In Section 6, we mention that future work could enable us to produce an implementation $\hat{I}^{\mathrm{SIG}}$ such that $\hat{I}^{\mathrm{SIG}}$ emulates $\mathcal{F}_{\mathrm{SIG}}$ from the proof presented in [21] for a non-key-manageable signature functionality. Since this is out of the scope of this work, we leave this as an assumption.

F An example implementation of $\mathcal{F}_{\mathrm{ach}}$

The following implementation of the authenticated channel functionality $\mathcal{F}_{\mathrm{ach}}$ (see page 4) may serve as an example on how to use $\mathcal{F}_{\mathrm{KM}}$ in a protocol. The idea is the following: two parties, the sender and the recipient, use the set-up phase to generate a shared signature key. The recipient creates this signature key, stores the public part, shares the

private part, which is hidden inside $\mathcal{F}_{\text{KM}}$, with the sender, and then announces the end of the set-up phase. At some later point, when the sender is instructed to send a message, it attaches the signature to the message. The recipient accepts only messages that carry a valid signature, which she can verify using the public part of the shared signature key. Obviously, a similar implementation without the key-management functionality $\mathcal{F}_{\text{KM}}$ is possible (although other means of pre-sharing signature keys, or MAC keys for that matter, would be required). However, our aim is to provide a very concise use case.

Formally, the protocol π_{ach} defines three protocol names: $\texttt{proto}-\texttt{ach}$, $\texttt{proto}-\texttt{fkm}$ and $\texttt{proto}-\texttt{sig}$. $\pi_{\text{ach}}(\texttt{proto}-\texttt{sig})$ is defined by the signature functionality $\mathcal{F}_{\text{SIG}}$. $\pi_{\text{ach}}(\texttt{proto}-\texttt{fkm})$ is defined by $\mathcal{F}_{\text{KM}}$, for the parameters $\overline{\mathcal{F}} = \{\mathcal{F}_{\text{SIG}}\}, \overline{\mathcal{C}} = \{\{\texttt{sign}, \texttt{verify}\}\}$ and a static key-hierarchy Π as defined in the previous section. $\pi_{\text{ach}}(\texttt{proto}-\texttt{ach})$ parses the session parameter as a tuple $< P_{pid}, Q_{pid}, \mathit{label} >$, where label is used to distinguish different channels, and P_{pid}, Q_{pid} to identify the sender and the recipient. Let $\texttt{sid}$ be the session id. Then we will use P to denote $< P_{pid}, \texttt{sid} >$ and Q to denote $< Q_{pid}, \texttt{sid} >$. As we want to keep the example simple, we do not model party corruption. The following code defines the behaviour of the sender P:

```
ready−sender: accept <ready> from parentId;
  call F_KM with <ready>
import[ready-sender]: accept <share•, h'> from F_KM; h̄' ← h';
  call F_KM with <finish_setup>
send[import]: accept <send, x> from parentId;
  call F_KM with <sign, h̄', x>; accept <sign•, σ> from F_KM; send <x, σ> to A
done [send]: accept <done> from A;
  send <done> to parentId
```

When we say that P calls $\mathcal{F}_{\text{KM}}$, we mean that P sends a message to $< P_{pid}, < \texttt{sid}, < \texttt{prot}-\texttt{fkm}, < \mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}, \mathit{Room} >>>$, where $\mathcal{U} = \{P, Q\}, \mathcal{U}^{\text{ext}} = \{Q\}$, $\mathit{Room} = \{P, Q\}$ and $\mathcal{ST}$ any two regular peers of P and Q, but not P and Q themselves. Similar for the receiver Q:

```
ready−receiver:
  accept <ready> from parentId; call F_KM with <ready>;
  accept <proceed> from A; call F_KM with <new, F_SIG, 0>;
  accept <new, h, vk> from F_KM; h̄ = h; v̄k = vk; call F_KM with <share, h>
deliver [ready-receiver]:
  accept <deliver, x, σ> from A; call F_KM with <verify, v̄k, <x, σ>>;
  accept <verify, 1> from F_KM; send <deliver, x> to parentId
```

The following lemma makes use of the fact that $\mathcal{F}_{\text{KM}}$ provides an authentic way to share keys during the set-up phase, and that $\mathcal{F}_{\text{SIG}}$ outputs $\texttt{<verify, 1>}$ only if the corresponding message was "registered" before. The full proof can be found in Appendix G.

Lemma 4. *π_{ach} emulates $\mathcal{F}_{\text{ach}}$.*

G Proof for Lemma 4

Lemma 4. *π_{ach} emulates $\mathcal{F}_{\text{ach}}$.*

Proof. We have to show that there exists a simulator Sim that is bounded for $\mathcal{F}_{\text{ach}}$ and that, for every well-behaved environment Z rooted at $\texttt{prot-fach}$,

$$\text{Exec}[\pi_{\text{ach}}, A_D, Z] \approx \text{Exec}[\mathcal{F}_{\text{ach}}, \mathit{Sim}, Z].$$

We define the following simulator Sim:

```
ready−sender:
  accept <sender-ready> from <ideal>
  send <F_KM, <ready•, P>> to <env>
```

faux−ready−receiver[¬ready-sender]:
  accept <ready-receiver-early> from <ideal>
  send <$\mathcal{F}_{\text{KM}}$, <ready$^\bullet$, Q> to <env>
  accept <Q, proceed> from <env>
  send <$\mathcal{F}_{\text{KM}}$, error> to <env>
ready−receiver[ready-sender]:
  accept <receiver-ready> from <ideal>
  send <$\mathcal{F}_{\text{KM}}$, <ready$^\bullet$, Q>> to <env>
  accept <Q, proceed> from <env>
  $\overline{sk}, \overline{vk} \leftarrow \mathsf{KG}(1^\eta)$
  send <$\mathcal{F}_{\text{KM}}$, <finish_setup$^\bullet$>> to <env>
send [ready-receiver]:
  accept <send, x> from <ideal>
  $\overline{\sigma} \leftarrow \mathsf{sign}(\overline{sk}, x)$
  send <P, <x, σ>> to <env>
done [send]:
  accept <P, done> from <env>
  send <done> to P
faux−deliver[ready-receiver∧¬send]:
  accept <Q, <deliver, x, σ>> from <env>
  send <Q, error> to <env>
deliver [send]:
  accept <Q, <deliver, x, σ>> from <env>
  if $\sigma = \overline{\sigma}$
    send <deliver, x> to <ideal>
error :
  accept <P, error> from <ideal>
  send <P, error> to <env>

Let us fix the session id sid, and assume it is of the form <sid', <prot-ach, <P_{pid}, Q_{pid}, $label$>>...>. Let $P =< P_{pid}, \texttt{sid} >$ and $Q =< Q_{pid}, \texttt{sid} >$. First, we show that Sim is bounded for $\mathcal{F}_{\text{ach}}$: It is trivial to verify that Sim is time-bounded, since KG and sign are assumed to be. Every flow from Sim to $\mathcal{F}_{\text{ach}}$ is provoked by an input from the environment, and the length of the message from Sim to $\mathcal{F}_{\text{ach}}$ is polynomially related to the length of the message from the environment to Sim.

Next, we show that, for every well-behaved environment Z rooted at prot−fach, $\text{Exec}[\pi_{\text{ach}}, A_D, Z] = \text{Exec}[\mathcal{F}_{\text{ach}}, Sim, Z]$, by showing a stronger invariant:

At the end of each epoch, the following conditions hold true:

1. The view of Z is the same in $[\pi_{\text{ach}}, A_D, Z]$ and $[\mathcal{F}_{\text{ach}}, Sim, Z]$.
2. If $\mathcal{F}_{\text{ach}}$ has finished a step S in $[\mathcal{F}_{\text{ach}}, Sim, Z]$, Sim has finished the same step, and in $[\pi_{\text{ach}}, A_D, Z]$, either P or Q has finished S.

Induction over the number of activations of Z. In the base case, Z has not called any party, so the invariant holds trivially. Now assume the invariant held true at the end of the previous activation of Z, after which Z sends a message m to some party. We have to show the invariant to hold true when this new epoch is over, i. e., Z is activated again. Case distinction over the steps that were completed by $\mathcal{F}_{\text{ach}}$ in $[\mathcal{F}_{\text{ach}}, Sim, Z]$ before Z emitted m. Note that the guards define a partial order of the steps, therefore it is sufficient to perform the distinction over the last completed step:

1. *$\mathcal{F}_{\text{ach}}$ has not finished any step yet.* If Z sends $m = <$ ready $>$ to P, then in $[\mathcal{F}_{\text{ach}}, Sim, Z]$, Sim will translate $\mathcal{F}_{\text{ach}}$'s response into <$\mathcal{F}_{\text{KM}}$, <ready$^\bullet$, P>>, which is what Z would receive in $[\pi_{\text{ach}}, A_D, Z]$ due to the definition of $\mathcal{F}_{\text{KM}}$. If Z sends the message to Q instead, in $[\mathcal{F}_{\text{ach}}, Sim, Z]$ an error message is sent to Sim, as the guard for the step ready−receiver would not be fulfilled. By induction hypothesis, Sim is in the same state as $\mathcal{F}_{\text{ach}}$, and therefore simulates the error that $\mathcal{F}_{\text{KM}}$ would send out in $[\pi_{\text{ach}}, A_D, Z]$, because it receives a share query from Q without having received ready from P before – otherwise Z would have sent ready to P and

then $\mathcal{F}_{\text{ach}}$ would have finished this step. If any other message is sent, $\mathcal{F}_{\text{KM}}$ sends an error message to Sim, who forwards it to Z just like A_D does in $[\pi_{\text{ach}}, A_D, Z]$.

2. $\mathcal{F}_{\text{ach}}$ *has finished* `ready-sender`. If Z sends $m = <$ `ready` $>$ to Q, in $[\pi_{\text{ach}}, A_D, Z]$ it will receive $<\mathcal{F}_{\text{KM}}, <$ `ready`$^\bullet, Q >$ from A_D. Similarly in $[\mathcal{F}_{\text{ach}}, Sim, Z]$, Sim would wait for the same response to proceed. If Sim and Q in π_{ach} were in this step before Z sent m, and $m =< Q,$ `proceed` $>$, then, in $[\pi_{\text{ach}}, A_D, Z]$, Q would create a key on $\mathcal{F}_{\text{KM}}$, and save the handle as well as the public part. Before Z's next activation, $\mathcal{F}_{\text{KM}}$ would activate P on step `import` which would store the same handle (by definition of $\mathcal{F}_{\text{KM}}$) and finish the setup phase, producing as only observable output to Z the message $<\mathcal{F}_{\text{KM}},$ `finish_setup`$^\bullet>$. In $[\mathcal{F}_{\text{ach}}, Sim, Z]$, Sim draws the secret and public part of the key itself (using the same function KG that $\mathcal{F}_{\text{SIG}}$, which is called by $\mathcal{F}_{\text{KM}}$ uses). It would then produce the same output to Z. Any other message m would result in an error-message and treated as described before.
3. $\mathcal{F}_{\text{ach}}$ *has finished* `ready-receiver`. If Z sends $m =<$ `send`$, x >$ to Q, then Z would receive $< P, < x, \sigma >>$, for an equally distributed σ, in both networks $[\pi_{\text{ach}}, A_D, Z]$ and $[\mathcal{F}_{\text{ach}}, Sim, Z]$, since, as mentioned before, the value at the key-position is equally distributed. If Z sends $m =< Q, <$ `deliver`$, x, \sigma >>$ to Sim, it receives $<Q,$ `error`$>$ as response. In $[\pi_{\text{ach}}, A_D, Z]$, A_D forwards this to Q who queries $\mathcal{F}_{\text{KM}}$. Because `deliver` was not executed yet, signs in $\mathcal{F}_{\text{SIG}}$ is empty. Since $\overline{vk}$ was created in $\mathcal{F}_{\text{SIG}}$/$\mathcal{F}_{\text{KM}}$ (since `ready-receiver`, by induction hypothesis holds in π_{ach}, too), and no corruption message is emitted by any party in π_{ach}, the response from $\mathcal{F}_{\text{SIG}}$/$\mathcal{F}_{\text{KM}}$ cannot be $<$`verify`$^\bullet,$ `1`$>$. Therefore, the second accept step in `deliver` in Q fails, and $<Q,$ `error`$>$ is output, just as in $[\mathcal{F}_{\text{ach}}, Sim, Z]$. Any other message m would result in an error-message and treated as described before.
4. $\mathcal{F}_{\text{ach}}$ *has finished* `send`. If $m =< P,$ `done` $>$ is send to A_D in $[\pi_{\text{ach}}, A_D, Z]$, Z receives $<$`done`$>$ from P. In $[\mathcal{F}_{\text{ach}}, Sim, Z]$, Sim translates this to $< done >$, which $\mathcal{F}_{\text{ach}}$ relays to the same response via the dummy party P. If Z sends $m =< Q, <$ `deliver`$, x, \sigma >>$ to A_D in $[\pi_{\text{ach}}, A_D, Z]$, it receives $<$`deliver`$, x$ from Q, but only in case that Z has sent a message $<$`send`$, x>$ to P earlier, when P was in state `import`, and gave the output $<x, \sigma>$ to A_D, since $\mathcal{F}_{\text{SIG}}$ keeps a list of previously signed message and their signatures. In $[\mathcal{F}_{\text{ach}}, Sim, Z]$, by definition of Sim and $\mathcal{F}_{\text{ach}}$, Z receives the same response from Q if x is the same x in a $<$`send`$, x>$ query accepted earlier, and σ is the output produced by Sim before forwarding. Since $\mathcal{F}_{\text{KM}}$ may have only accepted such queries when `ready-receiver` was finished, and `done` or `deliver` were not yet called, this corresponds to the same input z in $[\pi_{\text{ach}}, A_D, Z]$. Any other message m would result in an error-message and treated as described before.
5. $\mathcal{F}_{\text{ach}}$ *has finished* `deliver`. No messages are accepted anymore (i. e., they lead to error-messages).