Universally Composable Key-Management

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.

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*, e.g., the RSA PKCS#11 standard [1], IBM's CCA [2] and the trusted platform module (TPM) [3] API, all of which protect keys by providing an API that allows to address keys only indirectly, via pointers which are called *handles*. Recent work has tried to define appropriate security notions for APIs in terms of cryptographic games [4, 5]. 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. [5] to a framework that allows for composition.

Composability can be proven in frameworks for simulation-based security, such as GNUC [6], a deviation of the Universal Composability (UC) framework [7]. 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.

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 API designs, because changes to operations that are not part of the key-management or the addition of new functions do not affect the emulation proof. 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.

Limitations. Our key-management functionality is currently 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 practitioners indeed favour deterministic key-encryption in protocol design and standardization efforts (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 [8], so we place limitations on the types of corruptions that the environment may produce.

Related Work. Building on the work of Longley and Rigby [9] and Bond and Anderson [10] on API attacks, several recent papers have investigated the security of APIs on the logical level adapting symbolic techniques for protocol analysis [11–13], finding many new attacks. As discussed before, recent work on appropriate security notions for APIs in terms of cryptographic games [4, 5] lacks composability. Some aspects of the ideal functionality $\mathcal{F}_{\mathrm{crypto}}$ by Küsters et al. [14] 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. A full version of this paper with complete proofs is available at [15].

2 Background: GNUC

Hofheinz and Shoup [6] recently proposed the GNUC ("GNUC is Not UC") framework as an attempt to address several known shortcomings in UC. These shortcomings are also addressed to a greater or lesser extent by other altenative frameworks [17, 18]: 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 [6] for additional details.

2.1 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*. 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 $< \alpha_1, \ldots, \alpha_{\mathrm{k}} >$. The last component α_k specifies which protocol is run with which protocol parameters. A machine can come to life by being called by the environment or by a subroutine call. In this case, the session id of the caller has to be a prefix of the session id of the subroutine. 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 , but via the adversary, modelling an insecure network, or via the ideal peer, who can communicate with all regular protocol parties and the adversary.

The code of the machines is described by a sequence of steps similarly to [6, § 12]. Each step is a block of the form `name [conditions]: P`. The label `name` identifies 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.

2.2 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. 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.

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 [6, 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 [6, § 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 [6, § 6].

Definition 1 (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

The network we want to show secure has the following structure: a set of users which takes input from the environment, each of which is connected to his security token.

Each security tokens is a network entity, just like the users, but has a secure channel to the user it belongs to. Cryptographic keys are stored on the token, but are not given directly to the user – instead, at creation of a key, the user (and thus the environment) receives a handle to the key.

We consider such a network secure if it emulates a network in which the users are communicating with a single entity, the key-management functionality $\mathcal{F}_{\mathrm{KM}}$, instead of their respective security token. It gives the users access to its operations via handles, too, and is designed to model the "ideal" way of performing key-management. To show the security of the operations that have nothing to do with key-management, it accesses several other functionalities which model the security of the respective operations. This allows us to have a definition that is applicable to many different cases.

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

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 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. 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 [4, 5] 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, e. g., using semantically secure symmetric or asymmetric encryption, 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.

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: creating keys, changing their attributes, transferring keys and secure setup. We argue that a reasonable definition of secure key-management has to provide at least those operations. Furthermore, a user must be able to use the keys for cryptographic operations, e. g., generate a digital signature. 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.

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. 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., [7], bind the roles of the parties, e. g., signer and verifier, to a machine ID. 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. 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. As opposed to the real world, where security tokens map handles to keys, and compute the results based on the keys, in the ideal world, $\mathcal{F}_{\mathrm{KM}}$ maps handles to credentials, and uses those credentials to address KU functionalities, which compute the results. The implementation of a KU functionality maps credentials to cryptographic keys (see Definition 2). While credentials are part of the $\mathcal{F}_{\mathrm{KM}}$ and the KU-functionality, they are merely devices used for abstracting keys. They are used in the proofs, but disappear in the reference implementation presented in Section 3.4.

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

Definition 2 (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-(and credential-)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,* $\Pr[k' = k | (k', public) \leftarrow impl_{\mathtt{new}}(1^\eta)]$ *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 [6, §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* $\perp$*. The variable* L *holds a set of pairs and is initially empty.*

```
new: accept <new> from parentId;
  $(key, public) \leftarrow impl_{\mathtt{new}}(1^\eta)$; $(credential,\_) \leftarrow impl_{\mathtt{new}}(1^\eta)$;
  $L \leftarrow L \cup \{(credential, key)\}$; send <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. The calls $\langle\mathtt{corrupt}\rangle$ and $\langle\mathtt{inject}\rangle$ are necessary for cases where the adversary learns a key, or is able to insert dishonestly generated key-material.

Definition 3 (key-manageable functionality). *A poly-time functionality* $\mathcal{F}$ *(to be precise, an ideal protocol [6, § 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 4 (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* $\Pi \subset \{\mathcal{F}_1, \ldots, \mathcal{F}_l, \mathtt{KW}\} \times \cup_{i\in\{1,\ldots,l\}} \mathcal{C}_i^{priv} \cup \{\mathtt{new}, \mathtt{wrap}, \mathtt{unwrap}, \mathtt{attribute\_change}\} \times A \times A$*.*

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

- $C = \texttt{new}$, then $\mathcal{F}_{\text{KM}}$ allows the creation of a new key for the functionality $\mathcal{F}$ with attribute a.
- $\mathcal{F} = \mathcal{F}_i$ and $C \in \mathcal{C}_i^{priv}$, then $\mathcal{F}_{\text{KM}}$ will permit sending the command C to $\mathcal{F}$, if the key is of type $\mathcal{F}$ and has the attribute a.
- $\mathcal{F} = \texttt{KW}$ and $C = \texttt{wrap}$, then $\mathcal{F}_{\text{KM}}$ allows the wrapping of a key with attribute a' using a wrapping key with attribute a.
- $\mathcal{F} = \texttt{KW}$ and $C = \texttt{unwrap}$, then $\mathcal{F}_{\text{KM}}$ allows to unwrapping a wrap with attribute a' using a wrapping key with attribute a.
- if $C = \texttt{attribute\_change}$, then $\mathcal{F}_{\text{KM}}$ 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 $\mathcal{F}_{\text{KM}}$, corresponding to different security tokens in the real word.

Example 1. To illustrate the definition of policy consider the case of a single KU functionality for encryption $\mathcal{F}_{\text{enc}}$. 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	$*$
$\mathcal{F}_{\text{enc}}$	`new`	0	$*$
`KW`	`wrap`	1	0
`KW`	`unwrap`	1	0
$\mathcal{F}_{\text{enc}}$	`enc`	0	$*$

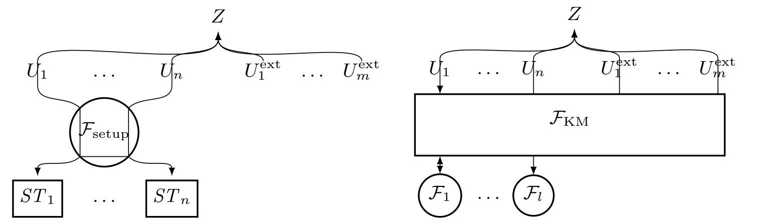
Fig. 1: Security policy

3.4 The key-management functionality and reference implementation

We are now in a position to give a full definition of $\mathcal{F}_{\text{KM}}$ together with an implementation. We give a description of $\mathcal{F}_{\text{KM}}$ in the Listings 2 to 7. For book-keeping purposes $\mathcal{F}_{\text{KM}}$ maintains a set $\mathcal{K}_{\text{cor}}$ 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.

Structure. $\mathcal{F}_{\text{KM}}$ acts as a proxy service to the KU functionalities. It is possible to create keys, which means that $\mathcal{F}_{\text{KM}}$ 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, $\mathcal{F}_{\text{KM}}$ 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.

Definition 5 (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).*


Z
U_1 … U_n U_1^{ext} … U_m^{ext}
$\mathcal{F}_{\mathrm{setup}}$
ST_1 … ST_n
Z
U_1 … U_n U_1^{ext} … U_m^{ext}
$\mathcal{F}_{\mathrm{KM}}$
$\mathcal{F}_1$ … $\mathcal{F}_l$

(a) Distributed security tokens in the network

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

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

The first tuple defines network parameters*:* $\mathcal{U} = \{U_1, \dots, U_n\}$ *are the party IDs of the users connected to a security token and* $\mathcal{U}^{\mathrm{ext}} = \{U_1^{\mathrm{ext}}, \dots, U_m^{\mathrm{ext}}\}$ *are the party IDs of external users, i. e., users that do not have access to a security token.* $\mathcal{S}T = \{ST_1, \dots, ST_n\}$ *are the party IDs of the security tokens accessed by* $U_1, \dots, U_n$*.* $Room \subset \mathcal{U}$*. The second tuple defines* key-usage parameters*:* $\overline{\mathcal{F}} = \{\mathcal{F}_1, \dots, \mathcal{F}_l\}$*,* $\overline{\mathcal{C}} = \{\mathcal{C}_1, \dots, \mathcal{C}_l\}$ *are key-manageable functionalities with corresponding sets of commands. Note that* `KW` $\notin \{\mathcal{F}_1, \dots, \mathcal{F}_l\}$*, 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 4) 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 $\mathcal{F}_{\mathrm{KM}}$ on the right. There are two kinds of users: $U_1, \dots, U_n =: \mathcal{U}$, each of whom has access to exactly one security token ST_i, and external users $U_1^{\mathrm{ext}}, \dots, U_m^{\mathrm{ext}} =: \mathcal{U}^{\mathrm{ext}}$, who cannot access any security token. The security token ST_i can only be controlled via the user U_i. The functionality $\mathcal{F}_{\mathrm{setup}}$ in the real world captures our setup assumptions, which need to be achieved using physical means. Among other things, $\mathcal{F}_{\mathrm{setup}}$ 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*) $U_1, \dots, U_n$ can be corrupted by the environment, while $ST_1, \dots, ST_n$ are incorruptible, since security tokens are designed to be better protected against physical attacks, as well as worms, viruses etc. Although we assume that the attacker cannot gain full control of the device (party corruption), he might obtain or inject keys in our model (key corruption).

ST_i makes subroutine calls to the functionality $\mathcal{F}_{\mathrm{setup}}$ which subsumes our setup assumptions. $\mathcal{F}_{\mathrm{setup}}$ 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 formally defined in the full version [15], and relays arbitrary commands sent by the environment via $\mathcal{F}_{\mathrm{setup}}$. 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 Appendix A in the full version of this paper [15]. All users in Room are allowed to share keys during the setup phase. 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.

Implementation. The implementation ST is inspired by [5] and is parametric on the KU parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ and the implementation functions $\overline{\mathtt{Impl}} := \{\mathtt{Impl}_{\mathcal{F}}\}_{\mathcal{F} \in \overline{\mathcal{F}}}$. 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.

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 1). Note that the security token communicates with its respective user via $\mathcal{F}_{\text{setup}}$, which forwards messages between ST_i and U_i, serving as a secret channel.

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

```
command[finish_setup]: accept <$C \in \mathcal{C}_{i'}^{priv}$,$h$,$m$> from $\mathcal{F}_{\text{setup}}$;
if  $\mathit{Store}[U_i,h]$ =<$\mathcal{F}_{i'}$,$a$,$k$> and <$\mathcal{F}_{i'}$,$C$,$a$,$*$>$\in \Pi$ and $\mathcal{F}_{i'} \neq$ KW
   send <$C^\bullet$,$\mathit{impl}_C(k,m)$> to $\mathcal{F}_{\text{setup}}$
```

Listing 1: 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 <`new`,$\mathcal{F}$,a>. 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 5).

```
new[ready]: accept <new,$F$,$a$> from $U \in \mathcal{U}$;
if  <$F$,new,$a$,$*$> $\in \Pi$
   if  $F$ =KW then $(c, \mathit{public}) \leftarrow \mathit{impl}_{\mathtt{new}}^{KW}(1^\eta)$
   else  call  $F$ with <new>; accept <new$^\bullet$,$c$,$\mathit{public}$> from $F$
   if  $c \in \mathcal{K} \cup \mathcal{K}_{\text{cor}}$ then send <error> to $A$
```

```
  else  create  $h$; $Store[U,h] \leftarrow$ <F,a,c>; $\mathcal{K} := \mathcal{K} \cup \{c\}$; send <new$^\bullet$,$h$,$public$> to $U$
```

```
new[ready]: accept <new,$F$,$a$> from $\mathcal{F}_{\text{setup}}$;
  if  <$F$,new,$a$,$*$> $\in \Pi$
    $(k, public) \leftarrow impl^F_{\texttt{new}}(1^\eta)$; create $h$; $Store[U_i, h] \leftarrow$ <$F$,$a$,$k$>;
    send <new$^\bullet$,$h$,$public$> to $\mathcal{F}_{\text{setup}}$
```

Listing 2: 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 Listing 3). 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. The definition of $\mathcal{F}_{\text{KM}}$ is parametric in the algorithms wrap, unwrap and $impl_{new}$ used to produce the wrapping. When a handle to a credential c is corrupted, the variable $key[c]$ stores the corresponding key, c.f. Listing 6. We use $\$^l$ to denote a bitstring of length l drawn from a uniform distribution.

```
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 3: 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, $\mathcal{F}_{\mathrm{KM}}$ 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. In this keys, the ⟨ inject ⟩ call imports cryptographic value of the key onto the KU functionality, which generates a new credential for this value.

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

```
unwrap[finish_setup]: accept <unwrap,$h_1$,$w$,$a_2$,$F_2$,$id$> from $\mathcal{F}_{\mathrm{setup}}$
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$^\bullet$,$h_2$> to $\mathcal{F}_{\mathrm{setup}}$
```

Listing 4: Unwrapping w created with attribute a_2, F_2 and id using the key h_1. $\exists! x.p(x)$ denotes that there exists exactly one x such that $p(x)$ holds ($\mathcal{F}_{\mathrm{KM}}$ above, ST_i below).

There is an improvement that became apparent during the emulation proof (see Section 4). When unwrapping with a corrupted key, $\mathcal{F}_{\mathrm{KM}}$ checks the attribute to be assigned to the (imported) key against the policy, instead of accepting that a corrupted wrapping-key might import any wrapping the attacker generated. This prevents, e.g., a corrupted wrapping-key of low security from *creating* a high-security wrapping-key by unwrapping dishonestly produced wrappings. This detail enforces a stronger implementation than the one in [5]: ST validates the attribute given with a wrapping, enforcing that it is 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 $\mathcal{F}_{\mathrm{setup}}$;
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 $\mathcal{F}_{\mathrm{setup}}$
```

Listing 5: Changing the attribute of h to a' ($\mathcal{F}_{\mathrm{KM}}$ 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 $\mathcal{F}_{\mathrm{KM}}$, 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 6. ST implements this corruption by outputting the actual key to the adversary.

```
procedure for corrupting a credential $c$:
$\mathcal{K}_{\mathrm{cor}} \leftarrow \mathcal{K}_{\mathrm{cor}} \cup \{c\}$
for any $Store[U,h] =< F, a, c >$
  if $F = \mathtt{KW}$
    $key[c] \leftarrow c$; 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 6: 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 $\mathcal{F}_{\mathrm{setup}}$;
  if $Store[U_i,h] =< F, a, k >$ send <corrupt$^\bullet$,$h$,$k$> to $A$
```

Listing 7: Corrupting h ($\mathcal{F}_{\mathrm{KM}}$ 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 $\mathcal{F}_{\mathrm{KM}}$), 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 8). The implementation ST_i does not implement this step, since U_i, U_i^{ext} compute $impl_C(public, m)$ themselves.

```
public_command: accept <$C$,$public$,$m$> from $U \in \mathcal{U} \cup \mathcal{U}^{\mathrm{ext}}$;
  if $C \in \mathcal{C}_{i,pub}$
    call $F_i$ with <$C$,$public$,$m$>; accept <$C^\bullet$,$r$> from $F_i$; send <$C^\bullet$,$r$> to $U$
```

Listing 8: Computing the public commands C using $public$ and m ($\mathcal{F}_{\mathrm{KM}}$, note that ST_i does not implement this step).

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 [6, § 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 6 **($\mathcal{F}_{\mathrm{KM}}$).** *Given the KU parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$, and polytime algorithms* wrap*,* unwrap *and $impl_{new}$, 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,$sid$> ($= \mathcal{F}_{\mathrm{KM}}$)
relay_to: accept <m> from <ideal,$sid$> ($= \mathcal{F}_{\mathrm{KM}}$)
  send <m> to <<reg,$\mathcal{F}_i$>,<$sid$,<prot$-\mathcal{F}_i$,<>>> ($= \mathcal{F}_i$)
relay_from: accept <m> from <<reg,$\mathcal{F}_i$>,<$sid$,<prot$-\mathcal{F}_i$,<>>>
  send <m> to <ideal,$sid$> ($= \mathcal{F}_{\mathrm{KM}}$)
```

The ideal party runs the logic for $\mathcal{F}_{\mathrm{KM}}$ described in Listings 2 to 7.

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.

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 7. *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', and additionally marked* `new` *if $(\mathcal{F}, \texttt{new}, a, a') \in \Pi$. An edge (a, a') is in $\mathcal{A}_{\Pi,\mathcal{F}}$ whenever $(\mathcal{F}, \texttt{attribute\_change}, a, a') \in \Pi$.*

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

The following theorem shows that *(i)* the set of attributes an uncorrupted key can have in $\mathcal{F}_{\text{KM}}$ is determined by the attribute policy graph, *(ii)* second, there are exactly three ways to corrupt a key, and *(iii)* KU-functionalities receive the `corrupt` message only if a key is corrupted. The proof of these claims can be found in the full version [15].

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 $\texttt{Store}[U, h] = \langle \mathcal{F}, a, c\rangle$ 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 $\texttt{Store}[U', h'] = \langle \mathcal{F}, a', c\rangle$ 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 $\langle \texttt{corrupt}, \texttt{h}\rangle$ from U while* `Store[`U`,`h`]`$= \langle \mathcal{F}, a, c\rangle$*, 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 $\langle\ \texttt{wrap}, h', h, id\rangle$ from U while* `Store[`U, h'`]`$= \langle \texttt{KW}, a', c'\rangle$, `Store[`$U$`,`$h$`]`$= \langle \mathcal{F}, a, c\rangle$*, 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 $\langle\ \texttt{unwrap}, h', w, a, F, id\rangle$ from U while* `Store[`U`,`h'`]`$= \langle \texttt{KW}, a', c'\rangle$ *and $c = \mathsf{unwrap}_{c'}^{\langle F,a,id\rangle}(w)$ 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 $\mathcal{F}_i = \langle \texttt{ideal}, \langle \texttt{sid}, \langle \mathcal{F}_i, F\rangle\rangle\rangle$, $F = \langle\langle \texttt{reg}, \mathcal{F}\rangle, \langle \texttt{sid}\rangle\rangle$, accepts the message $\langle \texttt{corrupt}, c\rangle$ 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 $\mathcal{F}_{\text{KM}}$.*

4 Proof overview

We show that, for arbitrary KU parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$, the network $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$, consisting of the set of users $\mathcal{U}$ connected to security tokens $\mathcal{ST}$, the set of external users $\mathcal{U}^{\text{ext}}$ and the functionality $\mathcal{F}_{\text{setup}}$, emulates the key-management functionality $\mathcal{F}_{\text{KM}}$. We will only give a proof sketch here, the complete proof can be found in the full version [15].

Let $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ (in the following: π) denote the network consisting of the programs $\pi(\texttt{prot-fkm})$ and $\pi(\texttt{prot-fsetup})$. $\pi(\texttt{prot-fkm})$ defines the behaviours for users in $\mathcal{U}, \mathcal{U}^{\text{ext}}$ and $\mathcal{ST}$. Parties in $\mathcal{U} \cup \mathcal{U}^{\text{ext}}$ will act according to the convention on machine corruption defined in [6, § 8.1], while parties in $\mathcal{ST}$ will ignore corruption requests (security tokens are assumed to be incorruptible). $\pi(\texttt{prot-fkm})$ 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 $\mathcal{F}_{\text{setup}}$-hybrid protocol.

The proof proceeds as follows: making use of the composition theorem, the last functionality $\mathcal{F}_l$ in $\mathcal{F}_{\text{KM}}$ can be substituted by its key-manageable implementation $\hat{I}_L$. Then, $\mathcal{F}_{\text{KM}}$ can simulate $\hat{I}$ instead of calling it. Let $\mathcal{F}_{\text{KM}}\{\mathcal{F}_l/\hat{I}_l\}$ 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 $\mathcal{F}_{\text{KM}}\{\mathcal{F}_l/\texttt{Impl}_{\mathcal{F}_l}\}$ saves keys rather than credentials (for $\mathcal{F}_l$). We repeat the previous steps until $\mathcal{F}_{\text{KM}}$ does not call any KU functionalities anymore, i. e., we have $\mathcal{F}_{\text{KM}}\{\mathcal{F}_1/\texttt{Impl}_{\mathcal{F}_1}, \ldots, \mathcal{F}_n/\texttt{Impl}_{\mathcal{F}_n}\}$. Then we show that the network of distributed token π emulates the monolithic block $\mathcal{F}_{\text{KM}}\{\mathcal{F}_1/\texttt{Impl}_{\mathcal{F}_1}, \ldots, \mathcal{F}_n/\texttt{Impl}_{\mathcal{F}_n}\}$ that does not call KU functionalities anymore, using a reduction to the security of the key-wrapping scheme. This last step requires restricting 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 [15]. The main result follows from the transitivity of emulation and two lemmas describing the steps we just mentioned.

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 $KW = (impl^{\texttt{KW}}_{\texttt{new}}, \mathsf{wrap}, \mathsf{unwrap})$ is a secure and correct key-wrapping scheme(See Definition 12 in Appendix D [15]), then $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ emulates $\mathcal{F}_{\text{KM}}$ 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 $\mathcal{F}_1 = \mathcal{F}_{\text{Rand}}$ and $\mathcal{F}_2 = \mathcal{F}_{\text{SIG}}$ described below. The resulting security token $ST^{\mathcal{F}_{\text{Rand}},\mathcal{F}_{\text{SIG}}}$ 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, $\mathcal{F}_{\text{Rand}}$, is unusual, but demonstrates what can be done within the design of $\mathcal{F}_{\text{KM}}$, as well as it's limitations. It models how random values can be stored as keys, with equality tests 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 $\mathcal{F}_{\text{Rand}}$ 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  $k \notin \mathcal{K}_{\text{cor}}$ send <equal$^\bullet$,false> to $p$
    else  if  $n = k$ send <equal$^\bullet$,true> to $p$
corrupt: accept <corrupt,$c$> from $p$;
  if  $(c,0) \in L$
    $k \leftarrow \{0,1\}^\eta$; $L \leftarrow (L \setminus \{(c,0)\}) \cup \{(c,k)\}$; $\mathcal{K}_{\text{cor}} = \mathcal{K}_{\text{cor}} \cup \{k\}$;
    send <corrupt$^\bullet$, $k$> to $A$
inject: accept <inject,$n$> from $P$;
  ($c$,<ignore>) $\leftarrow \{0,1\}^\eta$; $\mathcal{K}_{\mathrm{cor}} \leftarrow \mathcal{K}_{\mathrm{cor}} \cup \{n\}$; $L \leftarrow L \cup \{(c,n)\}$;
  send <inject$^\bullet$,$c$> to parentId
```

The two functions $impl_{\texttt{new}}$ and $impl_{\texttt{equal}}$ give the key-manageable implementation: $impl_{\texttt{new}}$ on input 1^η gives output $(n, \_)$ for $n \leftarrow \{0,1\}^\eta$; $impl_{\texttt{equal}}$ on input n, n' gives output $n = n'$.

Due to space restrictions, the signature functionality $\mathcal{F}_{\mathrm{SIG}}$ is presented in the full version [15]. 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. 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 <wrap, h', h, id> was emitted, for $c' \in \mathcal{K}_{\mathrm{cor}}$, or an unwrap query <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 cases 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 <corrupt, c>.

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

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 2. 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.

Acknowledgments. This work has been partially supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n^o 258865, project ProSecure, and by the Direction Générale de l'Armement, contact no 11810242, Secure Interfaces. The authors thank the anonymous reviewers as well as R. Küsters and M. Tuengerthal for their helpful comments.

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. 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
5. 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
6. Hofheinz, D., Shoup, V.: GNUC: A new universal composability framework. Cryptology ePrint Archive, Report 2011/303 (2011) `http://eprint.iacr.org/`.
7. Canetti, R.: Universally composable signature, certification, and authentication. In: Proc. 17th IEEE workshop on Computer Security Foundations (CSFW'04). CSFW '04, IEEE Computer Society (2004) 219–
8. Hofheinz, D.: Possibility and impossibility results for selective decommitments. J. Cryptology **24**(3) (2011) 470–516
9. Longley, D., Rigby, S.: An automatic search for security flaws in key management schemes. Computers and Security **11**(1) (March 1992) 75–89
10. Bond, M., Anderson, R.: API level attacks on embedded systems. IEEE Computer Magazine (October 2001) 67–75
11. 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
12. 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
13. Delaune, S., Kremer, S., Steel, G.: Formal analysis of PKCS#11 and proprietary extensions. Journal of Computer Security **18**(6) (November 2010) 1211–1245
14. 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
15. Kremer, S., Künnemann, R., Steel, G.: Universally composable key-management (2012) `http://eprint.iacr.org/2012/189`.
16. Backes, M., Dürmuth, M., Hofheinz, D., Küsters, R.: Conditional reactive simulatability. International Journal of Information Security (IJIS) (2007)
17. 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
18. Maurer, U., Renner, R.: Abstract cryptography. In: Proc. 2nd Symposium in Innovations in Computer Science (ICS'11), Tsinghua University Press (2011) 1–21