A Machine-Checked Proof of Security for AWS Key Management Service

José Bacelar Almeida[1], Manuel Barbosa[2], Gilles Barthe[3], Matthew Campagna[4], Ernie Cohen[4], Benjamin Gregoire[5], Vitor Pereira[2], Bernardo Portela[2], Pierre-Yves Strub[5], and Serdar Tasiran[4]

[1] University of Minho and INESC TEC
[2] University of Porto (FCUP) and INESC TEC
[3] IMDEA Software Institute
[4] Amazon Web Services
[5] INRIA Sophia Antipolis
[6] École Polytechnique

Abstract. We present a machine-checked proof of security for the domain management protocol of Amazon Web Services' KMS (Key Management Service) a critical security service used throughout AWS and by AWS customers. Domain management is at the core of AWS KMS; it governs the top-level keys that anchor the security of encryption services at AWS. We show that the protocol securely implements an ideal distributed encryption mechanism under standard cryptographic assumptions. The proof is machine-checked in the EasyCrypt proof assistant and is the largest EasyCrypt development to date.

1 Introduction

Today's cloud services use sophisticated distributed architectures and algorithms to make data highly available and durable. To improve security, data at rest is typically encrypted, and decrypted only when/where necessary. The encryption keys themselves must be similarly durable and available; however, directly providing *all* keys to whichever service needs to use them unnecessarily increases the attack surface. For the most sensitive keys, it is more prudent to encapsulate them within a separate distributed encryption service. Such a service allows the creation of new keys, and uses these keys to encrypt and decrypt data, but does not expose the keys themselves to clients.

The subject of this paper is the AWS domain management protocol (henceforth abbreviated DMP), a distributed encryption service underlying the Amazon Web Services (AWS) Key Management Service (KMS [5]). AWS KMS, a core component of the AWS cloud, lets AWS customers create and manage encryption keys, providing a consistent view of encryption/decryption operations across AWS services, and controlling their use through AWS Identity and Access Management (IAM).[7] The widespread usage of AWS KMS and the central role of the DMP justifies a high-assurance security proof, leveraging recent developments in computer-aided cryptography such as [7,4,3].

In this paper, we present a fully mechanized, concrete proof of security of the DMP. Informally, the proof shows that the DMP provides an idealized encryption service.

Security goal. The DMP is designed to protect the confidentiality of data encrypted under domain keys and guarantee the correct operation of the interface it provides, even in the presence of a malicious individual interfering with the inner workings of the system. In particular, we consider an adversary that can commission and decommission hosts and HSMs (Hardware Security Modules), assumed to be under adversarial control, and manipulate (insert, delete, modify) messages

[7] Within AWS KMS, the DMP is used only to encrypt and decrypt customer master keys, the roots of the customer key hierarchies. The use of these master keys, and the design of KMS (outside of the DMP itself) is described in [5].

exchanged between system entities. Our goal is to show that such an adversary cannot gain further advantage than possibly causing the system to go unresponsive.

Formally, this security goal is defined using an ideal functionality and the real-vs-ideal world paradigm, similarly to the Universal Composability [14] framework. We prove that the DMP is indistinguishable from an idealized encryption service to an arbitrary external environment that can collude with a malicious insider adversary. This formalization captures precisely the security that the rest of AWS KMS needs from the DMP.

Main Theorem. Our main theorem states that the DMP behaves like an ideal authenticated encryption service. The theorem rules out attacks from arbitrary computationally bounded adversaries, under standard cryptographic assumptions for digital signatures, hash-functions and encryption schemes. Formally, we prove that the probability of breaking the protocol is smaller than

$$2 \cdot ((q_{\mathsf{ops}} + q_{\mathsf{hid}}) \cdot \epsilon_{\mathsf{sig}} + q_{\mathsf{dom}} \cdot \epsilon_{\mathsf{aead}} + \epsilon_{\mathsf{cr}} + \epsilon_{\mathsf{mrpke}} + \epsilon_{\mathsf{coll}}) \ ,$$

where q_{ops} and q_{hid} are upper bounds on the number of human operators and HSMs in the system, respectively; q_{dom} upper-bounds the number of domain keys; ϵ_{sig}, ϵ_{aead} and ϵ_{cr} denote the maximum probabilities of breaking a standard signature, authenticated encryption and cryptographic hash function, respectively; $\epsilon_{\mathsf{mrpke}}$ denotes the maximum probability of breaking a multi-recipient variant of public-key encryption; and ϵ_{coll} is a small statistical term related to collisions of signature verification keys. The security of cryptographic signatures, hashes, and authenticated encryption implies that all of the epsilons above (and hence the total probability of breaking the protocol) are negligible. A more precise statement of the concrete cryptographic setting and bound can be found in Sections 4 and 5.

Formalization. The proof is fully machine-checked in EasyCrypt [6], a proof assistant for cryptographic proofs. The development is 15K lines of code (loc), of which 500 loc comprise the protocol specification. Besides being the largest EasyCrypt development to date, the proof combines game-hopping techniques that are standard in cryptographic proofs, and rich inductive reasoning that is standard in program verification. The machine-checked proof is novel for the following reasons:

- We formalize a notion of key secrecy for KMS DMP in the style of cryptographic APIs [23] and extend prior work in this area by i. addressing a substantially more complex (distributed) API; and ii. making explicit which assumptions on the behaviour of human operators are necessary (as otherwise trivial breaks would be possible), whilst excluding all non-trivial breaks as in prior work by reducing to standard cryptographic assumptions.
- We relate the above definition of security with a real-vs-ideal world security definition for encryption services, by proving a (reusable) general composition result for combining cryptographic key management APIs with AEAD schemes. Our resulting top-level security theorem establishes that KMS DMP is as good as an ideal authenticated encryption service in the specified trust model.
- The machine-checked proof follows best proof engineering practices and favors reusable components, breaking down the verification effort in three types of steps:
  i. reusable results that lift standard cryptographic assumptions on signatures and hash functions to idealized versions that permit reasoning symbolically about complex invariants on authenticated data structures;
  ii. use rich inductive reasoning to prove that intricate authentication invariants hold in the security experiments, and rewrite (slice) the code of the security games to make explicit the split between data which is under adversarial control (due to trivial strategies that do not contradict the security claim) and data which is outside of the adversary's reach; and

iii. build on the previous results to conduct a game hopping proof that, first, idealizes digital signatures and hash functions, accouting for concrete (negligible) security losses; then modularly uses the authentication invariants to perform security experiment slicing; and finally reduces the key-secrecy property to the security of multi-recipient encryption.

Paper Structure. In Section 2 we give a bird's eye view of our approach and provide a road-map for the paper, before moving on to more technical sections. In Section 3 we give a detailed description of the DMP and of its formalization in EasyCrypt. Then, in Section 4 we formalize the security model that we have adopted and in which we have proved security of the DMP. In Section 5 we describe the machine-checked security proof. Section 6 gives an overview of the improvements to EasyCrypt that were developed during the project. Section 8 contains a summary of related work, and Section 9 the concluding remarks.

2 Overview

In this section we present an overview of the DMP goals and interface, and then outline the structure and contents of the EasyCrypt model and proof (shown in Figure 1).

DMP Concepts. The fundamental unit of security in the DMP is a *domain*. Each domain provides an independent distributed encryption functionality using a combination of machines and people (collectively referred to as entities) which may change over time. Each entity can participate in multiple domains.

Concretely, a domain is given by its entities, the rules governing the domain, and a set of (symmetric) *domain keys*. The entities are of three types: HSMs, human *operators*, and front-end *hosts*. HSMs are the inner security boundary of the DMP, and have a limited web-based API and no other active physical interfaces to their operational state. Sensitive cryptographic materials of an HSM are stored only in volatile memory, and are erased when the HSM exits operational state, including shutdowns and resets. Domain keys likewise appear in the clear only in the volatile memory of HSMs in the domain.

The goal of the DMP is to govern the operations on domain keys and to manage membership of HSMs in a domain, as well as authorizing operators to act on a domain. HSMs do not communicate directly with each other. Thus, a central function of the DMP is to synchronize the domain state between domain participants. For this purpose, all information about a domain state, including its domain keys, is transferred and stored in a *domain token*. A domain token contains encryptions of the domain keys, and is authenticated in order to bind these encryptions to the domain state.

Domain state is modified through quorum-authenticated commands issued by authorized operators for that domain. Changes to domain state include modifying the list of trusted participants in the domain, modifying the set of quorum rules, and periodically rotating domain keys. Rules on quorum-signed commands are designed to mitigate attacks by colluding dishonest operators, namely attacks that might allow such operators to bypass the security protections provided by the HSMs. By requiring authorization from n operators from the domain, the security of operations that add new entities to a domain is anchored on the assumption that a quorum of n operators from the domain will always contain at least one honest operator that follows the protocol, where n is a security parameter for the domain. A more detailed description of the domain management operations is included in Section 3, along with their formalization in EasyCrypt.

DMP Implementation. Not counting the crypto libraries, the implementation of the DMP protocol is spread across some 16.5K lines of Java code. The conformance of this code to the protocol level design is checked via integration tests. Additionally, a formal code validation mechanism has

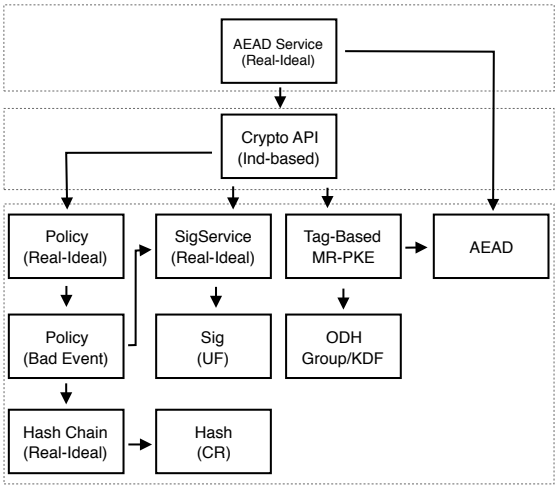

Fig. 1. Structure of the machine-checked proof

been built using an extension to a taint tracking type system (the Checker Framework, [19]). The checked property is a necessary condition for conformance to the protocol: a domain key must not be returned as part of the return value of an API call without first being encrypted by another key. This check is performed continuously, every time the KMS codebase changes, and it required only 323 manual annotations to the codebase.

DMP Functional Interface. The DMP provides an encryption functionality for each of its domains. Different domains can vary in the entities that they trust, their tolerance for dishonesty, and other security-related parameters. For each domain, the provided encryption functionality has the following interface[8] (formalized in Section 4):

- $\mathsf{New}(\mathsf{hdl})$ creates a new domain key within the domain and associates it to a key identifier hdl. The result indicates whether the operation was successful.
- $\mathsf{Enc}(\mathsf{hdl}, \mathsf{msg}, \mathsf{ad})$ uses the domain key associated with identifier hdl to encrypt the payload msg with associated data ad, returning the ciphertext.
- $\mathsf{Dec}(\mathsf{hdl}, \mathsf{cph}, \mathsf{ad})$ uses the domain key associated with identifier hdl to decrypt ciphertext cph with associated data ad and, if successful, returns the recovered plaintext.

The goal of the DMP security proof is to show that the DMP provides an idealized version of this interface (with a small probability of error). This ideal interface is close to that of standard Authenticated Encryption with Associated Data (AEAD), as detailed in Section 4, except that operations might fail (with no effect), as one might expect in a distributed system.

The EasyCrypt security proof consists of three layers.[9] The top layer gives a real-vs-ideal world security definition for the DMP and shows that security of a DMP domain in this model follows from the secrecy of its domain keys. The next layer shows that the DMP does indeed preserve the secrecy of domain keys of so-called "honest" domains (described in Section 3), assuming the secure implementation of the low-level cryptographic constructions used to create domain tokens. The bottom layer shows the security of these low-level constructions.

[8] This interface mentions only domain keys, so the functionality gives a simple way to separate the security provided by the DMP from its use in the rest of AWS KMS.

[9] Note that the scale of the proof does not increase the trusted base, as it is fully machine-checked by EasyCrypt – indeed, this is the main motivation for machine-checked provable security; the guarantee that the proof justifies the formalized security theorem requires only trust in EasyCrypt itself.

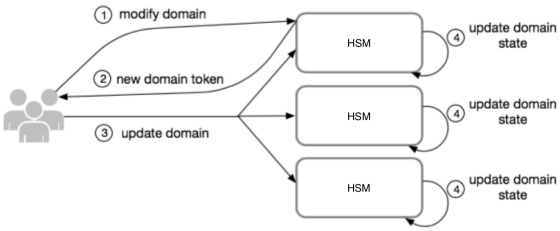

Fig. 2. High-level view of the DMP.

Proof: Real-vs-ideal World Security. At the top layer of the EasyCrypt proof lies a formal definition of security for encryption services supported by key management protocols such as the DMP (detailed in Section 5.1). The definition follows the real-vs-ideal paradigm of the UC framework (in fact, our proof can be seen as being carried out in a specific hybrid model in the UC framework, which we discuss in detail in Appendix A). Intuitively, the ideal functionality leaks nothing to the (adversarial) environment except the length of the data being encrypted, and implements decryption by maintaining a table mapping pairs $(\mathsf{cph}, \mathsf{ad})$ to messages. Ideal encryption always returns encryptions of 0^ℓ, where ℓ is the encrypted data length, and adds a new entry to this table; decryption simply does a lookup from the table (rather than calling the decryption function).

At this level we reduce the real-vs-ideal world security of the DMP to an indistinguishability-based security property that captures the secrecy of domain keys. This means that in the lower levels of the proof we do not need to reason about *how* domain keys are used; it suffices to prove that the DMP keeps domain keys hidden from the attacker's view.

Proof: Indistinguishability-based security. The second layer of results proves that the protocol hides all information about domain keys from the adversary' view. This is formalized as a cryptographic API [23] that guarantees domain key secrecy. The model captures the actions of a malicious insider adversary by allowing the domain management operations to consist of multiple adversarially orchestrated steps. The main challenge in this proof, formalized using the game-hopping technique, is to establish the invariants that govern the state of security experiments in each hop. These invariants combine properties that arise from standard cryptographic assumptions (e.g., absence of collisions and signature forgeries) with the inductive argument that justifies the soundness of the domain update operations carried out by honest operators, HSMs and hosts. It is by the joint action of these two types of guarantees that the domain management policy excludes dishonest entities from explicitly obtaining information on domain keys. The proof at this layer reduces the security of the API to the security of lower-level abstractions, all of which are formalized in the indistinguishability style, in order to facilitate the game-hopping technique. The details are discussed in Section 5.3.

Proof: Low-level abstractions. The lower layer of security results defines idealized versions of digital signature services, hash maps and certification of identity keys by human operators. It also contains proofs that these abstractions are indistinguishable from real-world instantiations down to standard cryptographic assumptions, which can then be used to make concrete the bounds in the theorems established at the higher layers in the development. At this level, we also formalize the specific flavor of (multi-recipient) public-key encryption that is used by the DMP.

This lower layer in the proof is meant to modularize various components, for three purposes: 1) lifting assumptions formalized as bad events, such as unforgeability and collision resistance, to indistinguishability definitions, allowing the higher-level parts of the proof to be solely based on indistinguishability game hops; 2) allowing for reuse of the abstractions across the project (e.g., we reuse the signature abstraction for both operator signatures and HSM signatures); and 3) allow for

multiple instantiations of the same underlying primitive (e.g., an encryption scheme with different constructions). This part of the proof is presented in Section 5.2.

3 KMS Domain Management Protocol

3.1 Detailed Description

A high-level operational view of the DMP is presented in Figure 2 (reproduced from [5]). Operators issue commands, HSMs manipulate the contents of domain tokens, and coordinator servers propagate updated domain tokens to each HSM in a domain to keep their domain states approximately synchronized (the latter are not shown and assumed for the purpose of the proof to be under adversarial control).

We now describe the core concepts and mechanisms involved in the DMP at the level of the mathematical model of the protocol that forms the basis of the formal proof of security. We begin by introducing the notion of a domain state, the different entities in the system and what assumptions we make about their behavior; we then explain the roles of these entities in domain state transitions, and conclude with an intuitive explanation of the security rationale underlying the design.

Protocol entities and assumed behavior. The DMP is implemented using three types of entities: *HSMs*, *hosts*, and *operators*. Each entity is identified with its identity (signature verification) key. A *genuine* entity is (the identity key of) an HSM/Host/Operator that behaves as specified by the DMP. A domain might include non genuine identity keys of any entity type, e.g. keys created by a malicious entity. HSMs perform the actual encryption and decryption operations,[10][11] and are the only entities allowed to manipulate domain keys in cleartext.

Operators are responsible for certifying identity keys: they sign statements claiming that a given identity key represents a genuine HSM, host or operator.[12] *Honest* operators only sign statements that are true, i.e. if an honest operator claims a key is that of a genuine HSM, the key is in fact genuine. Conversely, dishonest operators, while themselves genuine, might sign statements that are false. Note that we assume only that an honest operator can tell whether another operator is genuine, not whether he is honest. Genuine but dishonest operators model insider threats, possibly colluding with external adversaries. Non-genuine operators model arbitrary rogue identity keys that the adversary may also create in its attack. For the purpose of this paper, the quorum rule is defined by a security parameter n, which describes the minimum number of operators of the domain that must authorize an update over the domain state. Our security analysis is anchored on a global assumption that any set of n genuine operators contains at least one honest operator that follows the protocol. For example, a rule imposing that a quorum consists of a set of at least $n = 2$ operators from the domain guarantees that it requires at least two dishonest operators of the domain to break security.

[10] What we call HSMs are, in AWS KMS, running instances of FIPS 140-2 certified hardware security modules. They generate fresh identity and agreement key pairs when they boot, and store them only in volatile memory; the instance is effectively destroyed when power is lost. This simplifies physical protection — it suffices to guarantee that the machine cannot be physically attacked without losing power.

[11] In our protocol model, HSMs are conceptually stateless beyond their identity and agreement key pairs. In AWS KMS, HSMs maintain the current domain state for each domain they operate on, allowing their behavior to be more tightly controlled. This provides defense in depth, and potentially allows the proof of additional security properties not described here.

[12] In AWS KMS, these statements are actually commands to perform particular actions, such as a command requesting an HSM to add or remove HSMs or operators from a domain; such a command implicitly carries with it the certification from the command signers that the added entities are genuine. Note also that operators represent human operators, which play many additional security roles in the system; we describe only as much as is needed to justify the presented security proof.

Finally, hosts are the service endpoints. Although the actions of hosts in AWS KMS are more complex, our analysis focuses on the crucial role of honest hosts in directing cryptographic operations to honest domains:[13] as entry points in the system, hosts keep track of domain states and check that they are updated consistently with the domain management rules by HSMs. (Although we have not formalized this, in Appendix B we discuss how our proof implies security in a model where corrupt hosts are considered.)

Domain States, Tokens and Key Usage. A domain state logically consists of two parts: 1) a *domain trust* describing the administrative state of the domain—its entities and rules of operation—and 2) the actual domain keys. The trust components include a unique name of the domain, the set of entities of each type treated as genuine for the domain and the agreement keys for each HSM in the domain. The trust also includes a set of *quorum rules* defining what sets of operators (*quorums*) are considered trustworthy. Domain state modifications must be authorized by one of these quorums. Since the trust state of a domain can evolve over time (as operators, HSMs, and hosts are added and removed from the domain), the trust also includes the cryptographic fingerprint (hash) of the previous trust from which it was derived (by means of a trust update). Domain keys are kept secret from all entities other than the HSMs of the domain, whereas the domain trust only has to be authenticated. The concrete trust representation is signed by an HSM of the trust.

To provide the encryption functionality of a domain, its HSMs need access to the domain keys, which should not be exposed to other entities. To allow this, the domain state is concretely represented by a *domain token* signed by an HSM of the trust, which includes an encryption of the domain keys. This representation authenticates the state and binds the encrypted data to the trust; domain keys are decryptable only by the HSMs of the trust.

To allow encryptions between HSMs of a domain, each HSM has a long-term Diffie-Hellman *agreement* key pair. It certifies its (public) agreement key as its own by signing it with its identity key. To encrypt the domain keys for the other HSMs in a domain, an HSM first encrypts them under an ephemeral symmetric key. It then generates an ephemeral Diffie-Hellman key pair, uses DH agreement with this key pair and each of the agreement keys of the domain HSMs to compute a shared secret with each HSM, uses a key derivation function (KDF) on each shared secret to produce a set of symmetric keys, each of which is used to encrypt the ephemeral symmetric key. The result is a multi-recipient encryption of the domain keys to all the other HSMs in the domain based on DHIES [1]. This scheme has performance advantages: ciphertexts to all the HSMs in the domain take less bandwidth and can be batch-generated faster. The final domain token includes all of these encryptions, the public ephemeral Diffie-Hellman key, and the domain trust, all signed by the HSM creating the token. When an HSM receives a command (other than to create a new domain), it also gets a domain token on which to operate[14] (see Figure 2); it decrypts the domain token, and uses it to process the request.

Domain State Transitions. The security of a domain depends critically on all of the HSMs of a domain being genuine. For example, if an adversary could somehow introduce an identity into a domain for which he holds the private signing key, he could use this identity to sign an agreement key for which he holds the private key, have another member update the domain, and use his fake agreement key to decrypt the new domain token, breaking security. However, the HSMs performing

[13] Note that it is always safe to add domain-local secrets that appear only encrypted by keys of the domain. This can be done by allowing such secrets to only be created by an HSM of the domain (that immediately encrypts the new secret under a domain key), and to be re-encrypted from one domain key to another domain key of the same domain. For example, in AWS KMS, customer master keys are treated in this way, and hosts are also responsible for issuing the commands that manage the creation and usage of such keys.

[14] In AWS KMS, these domain states are typically cached within the HSMs, rather than being explicitly provided as part of a command. This provides slightly stronger security guarantees, e.g. wrt state rollback.

the domain update operations (adding and removing entities) do not carry with them a state that allows them to recognize the identity keys of genuine entities. Domain updates where HSMs modify trusts therefore rely on authorizations (attestations) of identity keys produced by operators, as follows.

An HSM will sign a new trust with any set of entities and quorum rules, if that trust is *initial* (i.e., has no predecessor fingerprint).[15] To modify an existing trust to create a new trust, an HSM checks that 1) it is an HSM of the existing trust, and 2) every entity in the new trust is either in the old trust or is certified (for the domain) by a quorum of operators of the existing trust. If these checks are successful, it creates and signs a new trust with the updated information, with the fingerprint of the existing trust as predecessor of the new one.

A host processing requests for a domain maintains in his state a trust for the domain; his commands can only be processed by HSMs in the trust.[16] As the trust of a domain evolves, hosts update their version of the trust. A host of a domain is initially given the initial trust of the domain. A host updates to a new trust only if 1) the predecessor fingerprint of new trust is the fingerprint of its current trust, and 2) the new trust is signed by an HSM of its current trust.

Invariant: an honest trust stays honest. A trust is *honest* if and only if its HSMs, hosts, and operators are all genuine, every quorum of operators (as defined by its quorum rules) contains an honest operator, and its predecessor trust (if any) is honest. A crucial property of the DMP is that, if a domain is initially created with an honest trust, then the domain will remain honest as updates are performed by HSMs. Note that this guarantee is enforced even though the HSM performing the updates keeps no state other than its own signing and agreement keys:[17] such an HSM has no way of distinguishing genuine operators from non-genuine ones, and it depends on attestation by operators to identify genuine HSMs.

Intuitively, the trust honesty property is preserved by the following inductive reasoning, which we formalize in our machine-checked proof. The base case is trivial. In the inductive step, an HSM is asked to update an honest existing trust to a new trust, and it performs checks (1-2) described above. By the quorum requirement on honest trusts, any entity certified by a quorum in the existing trust is guaranteed to be genuine. By check 2) above, the HSM will therefore guarantee that all entities in the new trust are genuine. To show that honesty is preserved it remains to prove that the quorum requirement is satisfied by the new trust. This is guaranteed by the global assumption on the security parameter n: since we just proved all operators in the new trust must be genuine, it must be the case that any subset of operators of size at least n contains at least one honest operator. Thus, if the current trust is honest, the successor trust (once signed by an HSM of the predecessor trust) is also honest. By the observation of the last paragraph, the actions of a host guarantee that if the host starts out with an honest trust, the host trust will remain honest.

Allowing Dishonest Domains. The DMP presented here allows dishonest domains to share entities with honest domains. In AWS KMS, this is prevented by having HSMs cache domain states, and using additional commands by which an HSM attests to its current domains. This allows operators to check that an HSM is loaded up with honest domain tokens, and so will never process requests using dishonest domain tokens. The current proof shows that locking down the HSMs in this way

[15] This corresponds to a domain creation request in AWS KMS.

[16] Commands are in fact issued by first obtaining an ephemeral symmetric key token generated by an HSM of the trust, with the plaintext key decryptable only using either a domain key or by the host requesting the token. We do not describe this communication mechanism in this paper.

[17] Again, AWS KMS caches domain tokens, rather than them being explicitly provided for updates; the DMP proof shows that this caching is not needed for the current security proof.

is not needed to achieve domain security. Note that the current proof nevertheless still applies to the security of AWS KMS, since the latter's stricter rules simply restrict adversarial action.

3.2 Formalization in EasyCrypt

Background on EasyCrypt. EasyCrypt is an interactive proof assistant for verifying the security of cryptographic constructions in the computational model. EasyCrypt adopts the code-based approach, in which primitives, security goals and hardness assumptions are expressed as probabilistic programs. EasyCrypt uses formal tools from program verification and programming languages to justify cryptographic reasoning, providing several program logics. We now describe the formalization of the DMP in EasyCrypt.[18]

```
type HId.    (* Identities of HSMs *)
type OpId.   (* Identities of Operators *)
type HstId.  (* Identities of Hosts *)
type DomId.  (* Identities of Domains *)

(* Trust data (genuine HSMs) and metadata *)
type Fpr.
type Quorum = OpId fset * int.
type Metadata = Quorum * Fpr option * HstId fset * DomId.
type Trust = (HId fset) * Metadata.

(* Domain keys in plaintext and encrypted form *)
type Keys = (Handle,Key) fmap.
type EKeys = MCTxt.

(* Unwrapped domain token *)
type TkData = { td_inst: bool; td_trust: Trust; td_skeys: Keys; }.

(* Wrapped domain token and trust *)
type TkWrapped = { tkw_ekeys: EKeys; tkw_signer: HId; tkw_sig: signature; }.
type Token = { tk_inst: bool; tk_trust: Trust; tk_wdata: TkWrapped; }.
```

Fig. 3. EasyCrypt Definitions for Domain Tokens

Trusts and Domain Tokens. Figure 3 shows the EasyCrypt declarations for domain tokens. The HId type is a pair holding the signing key of the HSM and its public agreement key. The OpId type is simply the signing key for the human operator. Type TkWrapped corresponds to signed data structures, which we reuse both for signed trusts and signed domain tokens. Technically, this simplifies the writing of invariants, as we only need to deal with one encoding function into the domain of signature schemes. We then syntactically distinguish (using bit tk_inst) *installable* signed trusts—only these can be installed in hosts—from signed domain tokens that also carry domain keys.

Wrapping and Unwrapping. Figure 4 shows how we formalize in EasyCrypt the operations carried out by HSMs to wrap (i.e., creating a data structure that is digitally signed and contains encrypted domain keys) and unwrap (verifying authenticity and recover cleartext domain keys) domain tokens. Operator checkToken performs consistency checks on tokens and performs signature verification; in particular, it also checks that the encrypted keys cph are encrypted to all HSM members of the trust (proj_pks(tr_mems old)=fdom cphs), and that the signer is a member of the trust (sid ∈ tr_mems old).[19]

[18] Notation: In EasyCrypt tup.1 denotes the first element of a tuple; notation is overloaded for fields in records, as in rec.field.

[19] When this operator is used in unwrap we have old=trust. The same operator is used for checking updates by both hosts and HSMs; in that case, the signer must be an HSM of the predecessor trust as well.

```
op proj_pks (hids: HId fset) : MPk = image snd hids.

(* Operator used for both unwrap and trust updates *)
op checkToken(inst : Installable, old new : Trust, tkw : TkWrapped) =
  let (cphs, sid, sig) = (tkw.tkw_ekeys, tkw.tkw_signer, tkw.tkw_sig) in
  let msg = encode_msg  (new,cphs,sid,inst)  in
  verify(sid.1, msg, sig) ∧ proj_pks (tr_mems old) = fdom cphs ∧
    sid ∈ tr_mems old.

proc wrap(hid : HId,td : TkData) : Token = {
  (inst, trust, keys) ← (td.td_inst, td.td_trust, td.td_skeys);
  mem ← tr_mems trust;
  tag ← encode_tag trust;
  ptxt ← encode_ptxt keys;
  ekeys ←$ mencrypt (proj_pks mem) tag ptxt;
  msg ← encode_msg (trust, ekeys, hid, inst);
  sig ← SigSch.sign(hid.1, msg);
  tkw ← { tkw_ekeys = ekeys; tkw_signer = hid; tkw_sig = sig };
  return { tk_inst = inst; tk_trust = trust; tk_wdata = tkw; };
}

proc unwrap(hid : HId, tk : Token) : TkData option = {
  rtd ← None;
  (inst, trust, data) ← (tk.tk_inst, tk.tk_trust, tk.tk_wdata);
  (* Signer must be in trust: so we call check with trust as old *)
  if (checkToken inst trust tk.tk_wdata ∧
        hid ∈ tr_mems trust ∧ hid_in hid benc_keys) {
    (ekey, dkey) ← benc_keys[hid.1];
    (tag, cphs) ← (encode_tag trust, data.tkw_ekeys);
    cph ← cphs[ekey];
    optxt ← decrypt dkey tag cph;
    if (optxt≠None)
      skeys ← decode_ptxt optxt;
      rtd ← Some { td_inst = inst; td_trust = trust; td_skeys = skeys; };
  }
  return rtd;
}
```

Fig. 4. HSM Operations for Domain Token (un)wrapping.

Encryption is formalized as tag-based multi-recipient public-key encryption [22,8]. Intuitively, this is a variant of public-key encryption where one can provide multiple agreement keys to the encryption algorithm, so that the resulting ciphertext can be decrypted independently by multiple recipients using only their individual private agreement keys. Encryption therefore takes a set of public keys (type MPk), and we model ciphertexts as a map from public keys to sub-ciphertexts that contain only the parts of the full ciphertext that each recipient needs to decrypt. This abstraction permits capturing the specific flavor of public-key encryption used in the DMP: the multi-recipient syntax permits modeling schemes that minimize bandwidth and encryption time via the reuse of randomness across multiple ciphertexts. The construction is tag-based, because it binds the ciphertext to the meta-data of the token (parameter tag) thereby preventing an adversary from porting such a ciphertext from a domain token associated to an honest trust to a domain token that is under its control. We describe how we formalized the proof of security for the encryption scheme used in AWS KMS in Section 5, and how we showed that the security model it satisfies is sufficient for the purposes of the DMP.

Trust management operations. Figure 5 shows the modeling of host operations. This consists of checking a signed trust (token) for consistency and its relation to its predecessor. The latter means checking that the signer is in the predecessor trust, and that the predecessor fingerprint contained in the new trust is the fingerprint of the predecessor trust (so the new trust can also not be an initial/root trust, and must actually have a fingerprint). Note that, for each host, we keep track of which trust is installed using a map hosts_tr.

```
op tr_initial(trust : Trust) = tfpr trust = None.
op checkTrustProgress(old new : Trust) = old = tfpr new.

proc installUpdatedTrust(hstid : HstId, tk : Token) : bool = {
  b ← false;
  if (hstid ∈ HstPolSrv.hosts_tr) {
    old ← hosts_tr[hstid];
    (* Signer must be in old trust and trust must be installable *)
    b ← tk.tk_inst ∧ checkToken tk.tk_inst old tk.tk_trust tk.tk_wdata ∧
          hstid ∈ hosts_tr ∧ ¬tr_initial (tr_data tk.tk_trust) ∧
          checkTrustProgress hosts_tr[hstid] tk.tk_trust;
    if (b) hosts_tr[hstid] ← tk.tk_trust;
  }
  return b;
}
```

Fig. 5. Host Trust Update Operation

Finally, Figure 6 shows the behavior of an HSM when checking a request for a trust update after *successful* unwrapping of the domain token containing the old trust. This ensures that an HSM member of the old trust has signed it. The update request is formalized by providing the new trust and a set of *authorizations*, which are just signatures by operators on the members of the new trust, who are not in the old trust. The check enforces that the minimum quorum size (tthr old) is preserved, and that a quorum of at least this size of operators in the old trust have signed the request. This is computed by a fold over the list of operators auth that signed the request.

4 Real-vs-Ideal security for KMS DMP

In this section we describe and justify the security definition we adopted for the DMP, corresponding to the AEAD-service layer as shown in Figure 1. We begin by defining a general syntax for (possibly

```
op checkTrustUpdate(old : Trust, new : Trust, auth : Authorizations) : bool =
  (* Check threshold n preserved *)
  tthr old = tthr new ∧
  (* Signers are a quorum *)
  fdom auth \subset tops old ∧
  (* Signers are a quorum of correct size *)
  (tthr old) ≤ card (fdom auth) ∧
  (* Check hash consistency *)
  tfpr new = hash old ∧
  (* Check all new members signed *)
  let newmems = tmems new \ tmems old in
  let newops = tops new \ tops old in
  (* Verify all signatures *)
  let msg = encode (newmems,newops) in
  all (fun o ⇒ o ∈ auth ∧ verify (o, msg, oget auth[o])) (fdom auth).
```

Fig. 6. HSM Trust Update Validation

distributed) cryptographic key management APIs, inspired by the work of Shrimpton, Stam and Warinschi [23]. This formalism abstracts away all the details of the protocol that are not directly related to key operations, and we believe it to be of independent interest to analyse the security of other key management APIs. This allows us to reason about different security models using simpler definitions, and later refine the results for the concrete case of the DMP. With this refinement in mind, we clarify how the introduced abstract notions map to DMP features in various remarks throughout the text.

4.1 Key Management APIs

Key management APIs store keys in *tokens*. In the real world, a token can be implemented by using trusted hardware (e.g., an HSM or a SIM card) to store and perform computations with the keys, in which case keys may never leave the trusted boundaries of the hardware device. In other settings, for example in the cases where the underlying hardware cannot store the key material in its internal state, tokens are implemented as cryptographically hardened data structures, which guarantee that only trusted devices can (temporarily) access the key material. Our formalization abstracts these details and applies to both cases.

A token tk is a data structure mapping handles (key identifiers) to keys key, together with some meta-data that is used by the API for management operations. The basic operations on tokens are:

- $\mathsf{TkManage}(\mathsf{tk}, \mathsf{cmd})$: a generic interface that captures all management operations that can be carried out on tokens, including creating a new empty token, changing meta-data, and all other operations that do not affect the keys stored in the token.[20] On input a token tk and a command description, it returns a (possibly updated) token.
- $\mathsf{TkNewKey}(\mathsf{tk}, \mathsf{hdl}, \mathsf{cmd})$: samples a new key from the appropriate distribution and adds it to the token under the relation $\mathsf{hdl} \to \mathsf{key}$ and following the API-specific instructions described in cmd. The success of this operation may depend on the consistency of the input token itself and on the command cmd, e.g. the command might violate a check on permissions for generating keys in a particular token.[21]
- $\mathsf{TkReveal}(\mathsf{tk}, \mathsf{hdl}, \mathsf{cmd})$: exposes the key associated with hdl within tk, following the API-specific instructions described in cmd. As before, the success of this operation may depend on the command details cmd.

[20] I.e., these commands may change the state of the API and even the meta-data associated with keys, but the handle-to-key map will not change.

[21] We do not model key deletion operations; including them in the model does not affect the proof rationale, but it adds complexity to invariants.

```
type KMS_Oracles = {
  proc newOp(badOp : bool) : OpId option * OpSk option
  proc requestAuthorization(request : Request, opid : OpId)
                                    : Authorization option
  proc newHst() : HstId
  proc installInitialTrust(hstid : HstId, tk : Token) : bool
  proc installUpdatedTrust(hstid : HstId, tk : Token) : bool
  proc newHSM() : HId
  proc newToken(hid : HId, new : Trust) : Token option
  proc updateTokenTrust(hid : HId, new_trust : Trust, auth : Authorizations,
                                tk : Token) : Token option
  proc addTokenKey(hid : HId, hdl : Handle, tk : Token) : Token option ··· }.
```

Fig. 7. KMS TkManage and TkNewKey Operations

The TkReveal operation is a modeling artifact. It is used to make explicit that any cryptographic API will contain as part of its internal mechanisms a procedure to recover keys contained within tokens, so that they can be used to provide cryptographic services. This serves two purposes: i. to define what correctness of an API means, and ii. to obtain keys managed by the API in the security games that define API security—in some cases these keys are simply given to the adversary in order to model security breaches and, in other cases, they are used to construct challenges for the adversary.

Relation to the DMP. We show in Figure 7 the EasyCrypt declarations matching the notions of TkNewKey (addTokenKey) and TkManage (the remaining *proc* declarations) for the DMP. addTokenKey simply takes the identity of the HSM that shall carry out the operation of adding a key to the token.

In token management (TkManage), we include procedures to create new genuine operators (the model allows both honest and dishonest ones, in which case the adversary gets the signing key) and genuine hosts and HSMs. The requestAuthorization procedure models the actions of honest operators signing attestation requests for genuine entity identities, in which case this management operation is really not operating on tokens, but only on the global state of the API. The body of this procedure just checks that all entities in the request are indeed genuine and signs the request with the key for honest operator OpId.

Two procedures model the operations on hosts: installing an initial trust in host hstid, and updating the installed trust. In the first case, the body of the procedure ensures that the installed trust is honest and initial. This captures the global assumption that we are focusing our analysis on hosts that were initially configured with honest trusts (there is nothing one can guarantee otherwise). In the latter case, the procedure executes the operations for hosts shown in Figure 5. Finally, the newToken and updateTokenTrust procedures model the actions of HSMs when they are called upon to create empty tokens, or to update a trust based on authorizations issued by operators.

Crucially our model enforces that, as in the DMP, the states of genuine HSMs, operators and hosts are totally disjoint, and that the only communication between the different entities in the model must be explicitly performed using calls to this API.

It remains to explain the semantics of the TkReveal operation within AWS KMS. Recall that the purpose of the operation in the syntax of key management APIs is to allow defining security experiments that explicitly have access to API-protected keys in order to express the goal of an adversary (e.g., to formalize that a key is indistinguishable from random, the experiment needs to have access to the key in order to construct a real-or-random oracle). In our model of the DMP, TkReveal is simply the operation that asks a genuine HSM to unwrap a domain token and return the key for a particular handle[22].

[22] TkReveal is of course not an actual operation of the DMP; it is only used as part of the proof.

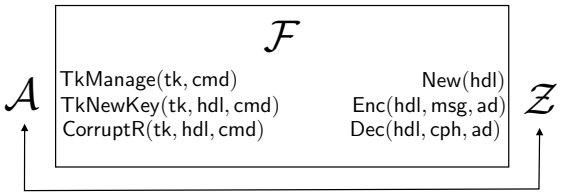

Fig. 8. Diagram of the UC-style ideal functionality.

Correctness. A natural requirement for key management APIs is that, subject to an API-specific set of restrictions over calls, they reliably store secret keys. To define correctness we introduce a predicate valid over traces of calls to TkManage and TkNewKey, which for a given token-handle-command input (tk, hdl, cmd) indicates whether a reveal operation should successfully return a key. In the AWS KMS model, valid simply requires that the command is placed on a genuine HSM, and the trust of the domain token is installed in a genuine host. Correctness requires that, if the HSM validates the domain token and a key with to hdl is stored within, then TkReveal must successfully recover it.

4.2 Defining Security of Encryption Services

Cryptographic key management APIs such as the DMP are used to build cryptographic services. In this section we follow the approach adopted in Universal Composability to formalizing this notion, by focusing on a service that provides authenticated encryption on client-chosen payloads, as described in Section 2.

The central component in our definition of correctness and security for such cryptographic service is an ideal functionality that describes how the service is expected to behave if it were provided as a monolithic self-contained block by a trusted-third-party. The ideal functionality provides both a functional specification of the service and a precise bound on the flow of information from the service to the (possibly adversarial) environment. We will define this functionality for the encryption service interface presented in Section 3.1, but the approach extends naturally to other cryptographic mechanisms providing confidentiality and authentication.

Our ideal functionality has two interfaces: the external interface accessible to an arbitrary environment $\mathcal{Z}$, and an adversarial interface that captures whatever influence a malicious insider adversary $\mathcal{A}$ is allowed with the underlying cryptographic API. The interface accessible to $\mathcal{A}$ is the same in the real and ideal worlds. We show this pictorially in Figure 8. As expected, we let the adversary $\mathcal{A}$ and the environment $\mathcal{Z}$ communicate freely. The execution is controlled by $\mathcal{Z}$, which may choose to interact with the outward facing encryption service interface or pass control to $\mathcal{A}$. In other words, the goal of $\mathcal{Z}$ is to distinguish the real world encryption service from an ideal authenticated encryption functionality. $\mathcal{A}$ is an insider colluding with the environment $\mathcal{Z}$, and helping it achieve this goal.

Our ideal AEAD functionality follows the standard UC approach: for each key hdl, calls to Enc(hdl, msg, ad) return encryptions of a fixed constant, rather than msg; and Dec(hdl, cph, ad) returns the original msg if and only if cph was previously given to $\mathcal{Z}$ as the result of a call to Enc(hdl, msg, ad). However, we need to consider the underlying distributed protocol that manages access to encryption keys, and the possibility that the adversary disrupts its operation, e.g., by not allowing the API to complete a request. Therefore, before answering requests placed by environment $\mathcal{Z}$, we let adversary $\mathcal{A}$ interact with the cryptographic API and lead it into a configuration of its choice.

We now describe in more detail the real-world and ideal-world experiments of our security model. The functionality keeps track of a list of corrupted key handles and a list of key handles that have been used in encryption and decryption.[23] In what follows, $\mathcal{E}$ and $\mathcal{D}$ represent the encryption and decryption operations of a standard AEAD scheme.

Real World. In the real world, the interface offered to environment $\mathcal{Z}$ behaves as follows:

- $\mathsf{New}(\mathsf{hdl})$ passes control to $\mathcal{A}$, indicating that $\mathcal{Z}$ requested the generation of a new domain key under handle hdl.
- $\mathsf{Enc}(\mathsf{hdl}, \mathsf{msg}, \mathsf{ad})$ passes control to $\mathcal{A}$, indicating that $\mathcal{Z}$ requested the AEAD encryption of payload msg with associated data ad, under the secret key corresponding to handle hdl. Adversary $\mathcal{A}$ is expected to eventually return a tuple $(\mathsf{tk}, \mathsf{cmd})$ and, if predicate $\mathsf{valid}(\mathsf{trace}, \mathsf{tk}, \mathsf{hdl}, \mathsf{cmd})$ holds and the key with hdl has not been corrupted by $\mathcal{A}$, the functionality computes $\mathsf{sk} \leftarrow \mathsf{TkReveal}(\mathsf{tk}, \mathsf{hdl}, \mathsf{cmd})$ and $\mathsf{cph} \leftarrow \mathcal{E}(\mathsf{sk}, \mathsf{msg}, \mathsf{ad})$, returning the result to $\mathcal{Z}$. Otherwise, an error symbol is returned.
- $\mathsf{Dec}(\mathsf{hdl}, \mathsf{cph}, \mathsf{ad})$ passes control to $\mathcal{A}$, indicating that $\mathcal{Z}$ requested the decryption of ciphertext cph with associated data ad, under the key corresponding to handle hdl. Adversary $\mathcal{A}$ is expected to eventually return a tuple $(\mathsf{tk}, \mathsf{cmd})$ and, if $\mathsf{valid}(\mathsf{trace}, \mathsf{tk}, \mathsf{hdl}, \mathsf{cmd})$ holds and the key with hdl has not been corrupted by $\mathcal{A}$, the functionality computes $\mathsf{sk} \leftarrow \mathsf{TkReveal}(\mathsf{tk}, \mathsf{hdl}, \mathsf{cmd})$ and $\mathsf{msg} \leftarrow \mathcal{D}(\mathsf{sk}, \mathsf{cph}, \mathsf{ad})$, returning the result to $\mathcal{Z}$. Otherwise, an error symbol is returned.

Note that, as in the previous section, our definition of security keeps track of calls placed by $\mathcal{A}$ to the TkManage and TkNewKey oracles and relies on the valid predicate to determine whether the service is required to produce a correct output to the environment $\mathcal{Z}$. Adversary $\mathcal{A}$ may therefore prevent the service from answering client requests by simply not executing the required API calls. Indeed, valid will naturally exclude sequences of API calls where a secret key with handle hdl is used before it is created, either because $\mathcal{Z}$ did not request its creation, or because $\mathcal{A}$ decided against carrying out this request. However, and crucially for our result, the valid predicate is totally oblivious of the honesty of trusts and tokens, so that any adversary that succeeds in leading the system into a dishonest configuration is not restricted in its actions: as shown below, it requires only that the selected host and HSM exist, and that the host is configured with the trust in the selected token.

```
op valid (t : Trace, tk : Token, hdl : Hdl, c : Cmd) : bool =
  (* ··· *)
  with c = Creveal hstid hid ⇒
    let (hstmap,hids,tklist) = t
    in ((hid ∈ hids) && (hstmap[hstid] = Some tk.`tk_trust)).
```

Ideal World. In the ideal world, the interface offered to environment $\mathcal{Z}$ behaves identically to what we presented for the real world, with the following exceptions. The ideal functionality keeps a table T associating handle-ciphertext-authenticated data tuples $(\mathsf{hdl}, \mathsf{cph}, \mathsf{ad})$ to payload (msg) values, which is initially empty. When $\mathcal{Z}$ places a call to one of its oracles, the following occurs:

- There is no change in oracle $\mathsf{New}(\mathsf{hdl})$.
- $\mathsf{Enc}(\mathsf{hdl}, \mathsf{msg}, \mathsf{ad})$ passes control to $\mathcal{A}$, indicating that $\mathcal{Z}$ requested the AEAD encryption of payload msg with associated data ad, under the secret key corresponding to handle hdl. Adversary $\mathcal{A}$ is expected to eventually return a tuple $(\mathsf{tk}, \mathsf{cmd})$ and, if predicate $\mathsf{valid}(\mathsf{trace}, \mathsf{tk}, \mathsf{hdl}, \mathsf{cmd})$

[23] These lists will always be disjoint, as we do not allow corrupted keys to be used for encryption operations. Fully removing this restriction would lead to a non-committing encryption level of security, not a goal for AWS KMS. However, in Appendix B we describe how our proof extends to a simple relaxation where corrupt keys can be used after corruption.

holds and the key with hdl has not been corrupted by $\mathcal{A}$, the functionality first computes $\mathsf{sk} \leftarrow \mathsf{TkReveal}(\mathsf{tk}, \mathsf{hdl}, \mathsf{cmd})$ and $\mathsf{cph} \leftarrow \mathcal{E}(\mathsf{sk}, \mathbf{0}^{|\mathsf{msg}|}, \mathsf{ad})$. It then updates the table with $(\mathsf{hdl}, \mathsf{cph}, \mathsf{ad}) \rightarrow \mathsf{msg}$ and returns cph to $\mathcal{Z}$. Otherwise, an error symbol is returned.
- $\mathsf{Dec}(\mathsf{hdl}, \mathsf{cph}, \mathsf{ad})$ passes control to $\mathcal{A}$, indicating that $\mathcal{Z}$ requested the decryption of ciphertext cph with associated data ad, under the key corresponding to handle hdl. Adversary $\mathcal{A}$ is expected to eventually return a tuple $(\mathsf{tk}, \mathsf{cmd})$ and, if $\mathsf{valid}(\mathsf{trace}, \mathsf{tk}, \mathsf{hdl}, \mathsf{cmd})$ holds and the key with hdl has not been corrupted by $\mathcal{A}$, then $\mathcal{Z}$ receives $T[(\mathsf{hdl}, \mathsf{cph}, \mathsf{ad})]$. Otherwise, an error symbol is returned.

This ends the presentation of the interface offered to $\mathcal{Z}$.

Adversarial interface. The oracles available to $\mathcal{A}$ reflect precisely the capabilities of a malicious insider adversary with direct access to the cryptographic API. As a worst case scenario, we let this adversary control the scheduling of API management and key generation operations, which in particular means that such an adversary may decide *not* to allow the API to answer the environment's requests.

We follow common practice in cryptographic API security definitions and allow $\mathcal{A}$ to corrupt keys using a $\mathsf{CorruptR}$ oracle, through which it can explicitly learn *any* secret key with handle hdl, even if such a key would otherwise be hidden from its view, i.e. even if such a key is meant to be protected by the API. Importantly, this captures a common real-world issue where keys are (maliciously or inadvertently) leaked, and ensures that non-leaked keys remain secret under those circumstances. To prevent the adversary from trivially distinguishing the real world from the ideal world, we never allow $\mathcal{A}$ to corrupt a key that is used by the environment $\mathcal{Z}$ for encryption/decryption operations.

Security goal. We say a cryptographic service is secure if, for all $(\mathcal{Z}, \mathcal{A})$, the views of $\mathcal{Z}$ in the real and ideal worlds are computationally indistinguishable. This means that the real-world system is not allowed to leak more than the ideal-world functionality, which reveals nothing about client payloads to insider adversaries interacting with the API.

5 Machine-checked Security Proof

We describe the machine-checked proof in EasyCrypt in three steps, corresponding to the three layers introduced in Section 2. We first describe the formalization of indistinguishability-based security, and show that this can be used as a convenient stepping stone in the analysis of the DMP, as it implies our real-vs-ideal world notion of security. This corresponds to the top layer in Figure 1. Then, we describe a set of low-level abstractions that we constructed in order to tame the complexity of the proof of indistinguishability-based security. We also discuss how we proved that the security guarantees that we modularly obtain from these abstractions follow from standard cryptographic assumptions, which allows us to state our main result in concrete terms. This corresponds to the lower-level layer in Figure 1. Finally we describe the core theorem that establishes the indistinguishability-based security of the DMP, and highlight the most interesting technical aspects of its proof. This corresponds to the intermediate layer shown in Figure 1.

5.1 From key hiding to real-vs-ideal security

The indistinguishability-based security of a cryptographic API [23] guarantees that cryptographic keys managed by the API remain hidden from the adversary. We adapt this notion to our formalism via a game where, in addition to interacting with the API to create and manage tokens, the

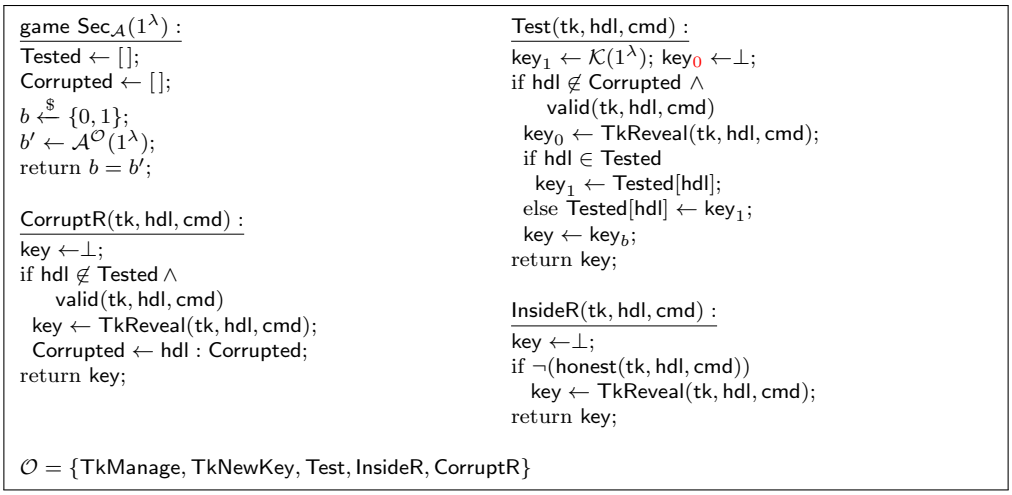

Fig. 9. Indistinguishability-based security.

adversary gets access to three oracles that capture the key hiding property. The Test oracle internally uses the TkReveal operator in order to model a real-or-random style challenge oracle on keys. The CorruptR and InsideR oracles use TkReveal to model explicit leakage of secret keys via corruption and execution traces recognized as dishonest by the security definition (note that such queries only strengthen the definition, as they state that domain keys will be protected even if one specific key is leaked by some external means). In particular, the CorruptR oracle allows the adversary to expose keys that it might otherwise be challenged on, and the InsideR oracle permits capturing CCA-style attacks. The game is shown in Figure 9. In this experiment, the adversary interacts with a set of oracles, which it can use to test valid, non-corrupt handles, receiving either the real key or a randomly generated one (depending on bit b). We require that no adversary can win this game other than with small probability over the random guess.

The following theorem, formalized and proved in EasyCrypt, implies that *any* cryptographic API that satisfies indistinguishability-based security will give rise to an encryption service that achieves our notion of UC-style security, when used together with an AEAD scheme satisfying the standard notions of correctness and security.

Theorem 1 (Informal). If a cryptographic API satisfies indistinguishability-based security, and $(\mathcal{E}, \mathcal{D})$ satisfy the standard notions of AEAD security then, for all adversaries $(\mathcal{Z}, \mathcal{A})$ the views of $\mathcal{Z}$ in the real world and ideal worlds are indistinguishable.

The proof proceeds as follows. One first uses the indistinguishability-based security of the API to show that Dec is effectively operating on a consistent secret key for each hdl, which are all hidden from the adversary. Note that consistency is implied by the indistinguishability definition, as the random branch $b = 1$ enforces that the same key is always returned for the same handle. Consistency is essential to ensure that standard AEAD security suffices in the next step of the proof, as otherwise one would need robust-encryption-level guarantees to ensure that operations with different keys on the same ciphertext do not leak information or allow forgeries.

The second step is to use AEAD correctness and unforgeability to switch to the table-like computation of Dec performed by the ideal functionality. Note that, when reducing to the security of the API, the validity predicate on traces enforced by the ideal functionality directly matches identical conditions in the oracles available in the indistinguishability-based definition. In particular, the corrupt oracles exactly match.

In EasyCrypt, the theorem is stated as follows. Notice that the reduction is tight, even though the bound includes the advantage of correctness and indistinguishability-security twice. This is because the proof requires symmetric game hops that first replace API managed keys with random ones using the IND-property and, after using AEAD security, restore the correct keys in order to match the ideal functionality exactly.

---

lemma IndImpliesUC:
$\mathbf{Adv}_{\mathrm{RI}}^{Z,A} \leq \mathbf{Adv}_{\mathrm{API}}^{B(Z,A)} + \mathbf{Adv}_{\mathrm{AEAD}}^{C(\mathrm{API},Z,A)} + \mathbf{Adv}_{\mathrm{API}}^{B_1(Z,A)}$.
where
$\mathbf{Adv}_{\mathrm{RI}}^{Z,A} =$
$|\mathbf{Pr}[\mathsf{Real}(Z, A, \mathrm{API}).\mathsf{main}():\mathbf{res}] - \mathbf{Pr}[\mathsf{Ideal}(Z, A, \mathrm{API}).\mathsf{main}():\mathbf{res}]|$
$\mathbf{Adv}_{\mathrm{API}}^{D} =$
$|\mathbf{Pr}[\mathsf{Ind}(D, \mathrm{API}).\mathsf{main}(\mathsf{false}):\neg\mathbf{res}] - \mathbf{Pr}[\mathsf{Ind}(D, \mathrm{API}).\mathsf{main}(\mathsf{true}):\mathbf{res}]|$
$\mathbf{Adv}_{\mathrm{AEAD}}^{D} =$
$|\mathbf{Pr}[\mathsf{AEAD\_LoRCondProb}(D).\mathsf{game}(\mathsf{false}):\neg\mathbf{res}]$
$- \mathbf{Pr}[\mathsf{AEAD\_LoRCondProb}(D).\mathsf{game}(\mathsf{true}):\mathbf{res}]|$

In the above statement, B, B_1 and C are constructed adversaries. This theorem allows us to focus on the main result proven in EasyCrypt in the rest of this section, namely that the KMS Domain Management API satisfies indistinguishability-based security.

5.2 Low-level abstractions

We now describe the lower layer in the EasyCrypt development, which defines and instantiates three reusable abstractions that are then used as black-box modules in the proof of indistinguishability-based security. The first abstraction is a generic signature service, which we use multiple times in the proof and can be reused in future EasyCrypt developments. The second abstraction is specific to AWS KMS, and it was created for managing the complexity of the proof by black-boxing the guarantees provided by the combined actions of HSMs and human operators in domain management. The third abstraction is the multi-recipient public-key encryption scheme, which is only used once in the main proof of security, but is meant for reuse in future EasyCrypt developments.

The ideal signature service abstraction. A central component in our modeling of the protocol and its proof of security (in both versions) is the signature service abstraction. We introduce a module called RealSignatureServ with an external interface that permits creating stateless signers, each with an independent signing key.

This service offers a signature verification procedure that works as a pure operator based on the public key and uses the signature verification algorithm for the underlying signature scheme. This means that any protocol using a digital signature with multiple signers and arbitrary verifiers that have access to the public keys can be described as a client to the real signature service. We then show that the standard property of unforgeability implies that this service is indistinguishable from an ideal one in which signature verification is now carried out by checking a list of signed messages.

The proof of security of the protocol relies on two instances of this abstraction, one for operator signatures and another one for HSM signatures. When using this abstraction, one first rewrites the description of the protocol as a function of the RealSignatureServ, which is always possible. Then we can use the fact that no adversary can distinguish this service from its ideal counterpart to modify the protocol into another one that uses a table-based idealized representation for signatures. From that point on, we can write invariants that refer to these idealized tables, which contain domain tokens/trusts (resp. identity attestations) if and only if they have been signed by genuine HSMs (resp. operators).

Domain Management Abstraction. We define a general notion of a domain management policy, for which we specify security in terms of distinguishing a real policy enforcement mechanism from an ideal policy enforcement mechanism.

Figure 10 details the module which captures the notion of a domain management policy based on the actions of hosts, operators and HSMs we have introduced in Section 3. This module maintains two data structures that keep track of the trusts manipulated by the system: protectedTrusts and parentTrust. Protected trusts are those that contain only genuine parties; this can happen because the trust is directly checked by operators in the isGoodInitialTrust operation to be a good initial trust, or because an HSM has checked that it is a valid descendant of a protected trust in checkTrustUpdate. The descendant relation is maintained using the parentTrust map.

The idealized version of the abstraction, which we omit for brevity, offers the same functionality as the real one, but ensures the following invariants:

i. All protected trusts contain only genuine HSM members and they descend from a protected trust.
ii. The descendant relation computed by HSMs behaves like an injective function—any trust has at most one valid parent throughout the lifetime of the system. This relation can be checked by hosts, if it has been computed by HSMs.

Intuitively this proof follows in the lines of the invariant described in Section 3. Note that this abstraction does not require genuine hosts or HSMs to be able to tell whether a trust was previously checked by a genuine HSM: it only speaks about trusts that have been tagged as protected. More precisely, the ideal policy says that, *if* a trust was previously tagged as protected, then the honest property is propagated *and* a genuine host will have the same view of the descendant relation; otherwise no guarantee is given.

In the main proof, which we discuss in the next subsection, we strengthen the security guarantee provided by this abstraction, relying on the authentication guarantees inherent to the signed trust data structure: looking ahead, we will use the fact that any trust for which the honesty property has been established must have been signed by a genuine HSM.

The following EasyCrypt theorem provides a concrete bound for any adversary distinguishing the real policy management module from its idealized version for the KMS Domain Management policy enforced by operators and HSMs. The bound is given by the collision resistance property of the hash function used to compute trust fingerprints, and the unforgeability advantage against the signature scheme used by operators to certify identity keys, scaled up by the maximum number of operators in the system q_ops.[24]

```
lemma domain_management:
  |Pr[TrustSecInd(A,IdealTrustService(OAR)).main():res]
    − Pr[TrustSecInd(A,RealTrustService(OAR)).main():res]| ≤
  Pr[CR(AdvCR(A)).main():res] + q_ops ∗ Pr[UF1(AdvUF1(A)).main():res].
where OAR=OA(RealSigServ)
```

Multi-recipient PKE Abstraction. The security proof of the main theorem relies on a tag-based multi-recipient public-key encryption abstraction. As a contribution of independent interest, we show that the variant of DHIES [2] used by the DMP to create domain tokes satisfies this notion of security. The IND-CCA security of this construction follows from the results in [2], together

[24] We note that our formalization relies on a unkeyed hash function. As we give concrete security reductions, our results are meaningful for unkeyed cryptographic hash functions used in practice, as discussed for example in [20]. Modifying the proof to support a keyed hash function would be straightforward, but would require the assumption that every entity in the system can be set-up with the same key.

```
module RealTrustService(OA : OperatorActions) : TrustService = {
  var protectedTrusts : Trust fset
  var parentTrust : (Trust,Trust) fmap
  proc newOp = OpPolSrv(OA).newOp
  proc addHId = OpPolSrv(OA).addHId
  proc requestAuthorization = OpPolSrv(OA).requestAuthorization
  proc newHst = HstPolSrv.newHst
  proc installInitialTrust = HstPolSrv.installInitialTrust
  proc installUpdatedTrust = HstPolSrv.installUpdatedTrust
  proc isInstalledTrust = HstPolSrv.isInstalledTrust

  proc isGoodInitialTrust(trust: Trust) = {
    b ← OpPolSrv(OA).isGoodInitialTrust(tr_data trust);
    if (b) protectedTrusts ← protectedTrusts `|` fset1 trust;
    return b; }

  proc checkTrustUpdate(old:Trust, new:Trust, auth:Authorizations) : bool = {
    c ← HSMPolSrv.checkTrustUpdate(old,new,auth);
    protected ← old ∈ protectedTrusts;
    if (c) {
      if (¬new ∈ parentTrust) parentTrust[new] ← old;
      if (protected) protectedTrusts ← protectedTrusts `|` fset1 new;
    }
    return c; }

  proc isProtectedTrust(trust : Trust) : bool = {
    return trust ∈ protectedTrusts;  }
   ...
}.
```

Fig. 10. Domain management policy abstraction.

with the general results on multi-recipient encryption in [8]. We also extend the result to the tag-based setting of Shoup [21], in which encryption takes a tag t and the decryption oracle permits decrypting any pair $(t, c) \neq (t^*, \mathsf{cph}^*)$, where (t^*, cph^*) was returned from the left-or-right oracle. This extension is crucial to show that a malicious HSM cannot modify an honest token to change the trust, in a way that decrypts successfully.

5.3 Main Theorem

The proof of indistinguishability security is carried out using the game-hopping technique. The first hop shows that the KMS Domain Management protocol can be re-expressed using the signature service and policy management abstractions introduced in the previous subsection. This hop is conservative, and introduces no additional terms in the security bound. The second and third hops consist of replacing the signature abstraction and the policy management abstraction with their ideal counterparts. These hops show that any adversary distinguishing the two games in the hop can be used to break the real-ideal indistinguishability guarantee for the low level abstractions, which we showed in the previous subsection can be, in turn, reduced to the security of standard cryptographic primitives.

At this point we perform a conservative hop that entails the most innovative part of the entire security proof. Here we combine two types of reasoning: 1) the inductive argument that establishes the propagation of trust honesty as discussed above; and 2) the global invariants guaranteeing the absence of collisions between trust fingerprints, and of signature forgeries. Together, these justify a game hop that slices the entire code of the indistinguishability game, isolating protected (honest) trusts from the remaining ones and enforces that the Test oracle can only be called by the adversary on protected trusts.

Furthermore, the game no longer relies on public-key decryption to recover domain-keys when dealing with protected trusts; instead, it keeps a table of domain keys to answer the adversary's challenge queries (i.e., at this point the domain keys are still encrypted in domain tokens, but domain tokens for protected trusts are never explicitly decrypted). To prove this hop we formalize an invariant (c.f. Appendix C) that establishes an equivalence between honest trusts and protected trusts occurring in the game, and further demonstrates that all trusts installed in genuine hosts are protected. This invariant also implies consistent management of the same key for each handle, which is necessary for indistinguishability-based security as mentioned above.[25]

At this point in the proof we can rely on the security of the multi-recipient encryption scheme to replace the domain keys encrypted in protected domain tokens with fixed constants. It is crucial that there is a strict separation between protected trusts and dishonest trusts, since the reduction to the security of the encryption scheme critically relies on the fact that one can use the left-or-right challenge oracle for public-key encryption to emulate the encryption of domain keys in protected domain tokens in both games. Intuitively, genuine HSMs map to the (honest) public keys in the multi-recipient public-key encryption game. CCA security of the underlying encryption scheme permits dealing with arbitrary InsideR queries, where one needs to decrypt ciphertexts contained in domain tokens mauled by the adversary.

The final step in the proof shows that, at this point, the adversary's view is independent of the domain key values and hence it has no advantage in winning the last game.

We conclude this section with the statement in EasyCrypt of the resulting security theorem. The indistinguishability advantage is upper-bounded by the probability that an adversary can break the domain management policy invariant (upper-bounded in lemma domain_management in the previous subsection), the probability that an adversary breaks the underlying signature scheme, and the probability that an adversary breaks the underlying multi-recipient public-key encryption scheme. Additional negligible terms account for the probability that the signing keys of honest parties collide with keys that an adversary generates itself and uses as identities for adversarially controlled operators and HSMs.

$$\begin{aligned}
&\Pr[\mathsf{KMS_RoR(A).main()}:\mathbf{res}] \leq \\
&\quad 1/2 + \mathbf{Adv}^{\mathrm{AdvMRPKE(A)}}_{\mathrm{MRPKE}} + \Pr[\mathsf{CR(AdvCR(A)).main()}:\mathbf{res}] + \\
&\qquad\qquad \mathsf{q_ops} * \Pr[\mathsf{UF1^{op}(AdvUF1^{op}(A)).main()}:\mathbf{res}] + \\
&\qquad\qquad \mathsf{q_hid} * \Pr[\mathsf{UF1^{hsm}(AdvUF1^{hsm}(A)).main()}:\mathbf{res}] + \epsilon
\end{aligned}$$

where

$$\begin{aligned}
&\mathbf{Adv}^{\mathrm{D}}_{\mathrm{MRPKE}} = \\
&\quad \Pr[\mathsf{MRPKE.Sec(D).main(false)}:\neg\mathbf{res}] - \Pr[\mathsf{MRPKE.Sec(D).main(true)}:\mathbf{res}] \\
&\epsilon = \mathsf{q_hid} * ((\mathsf{q_hid} + \mathsf{q_tkmng} + \mathsf{q_tkmng}) * \mathsf{max_size}) / \mathsf{n_keygen} + \\
&\qquad \mathsf{q_ops} * (((\mathsf{q_tkmng} + \mathsf{q_installinitial}) * 2 * \mathsf{max_size}) / \mathsf{n_keygen})
\end{aligned}$$

---

6 EasyCrypt usage and extensions

The EasyCrypt development consists of 15K lines of code (loc), where 500 loc correspond to the protocol specification. Additionally, 2.5K loc establish reusable definitions and supporting lemmas on standard cryptographic primitives and EasyCrypt data structures; 5.5K loc contain definitions and general results on KMS-specific security models; and 6.5K loc is the approximate size of the main security proof.

[25] To complete this hop we made explicit a property that is implied by the unforgeability of digital signatures, and which states that no adversary can guess the signing key of a genuine entity before it is generated. This allowed us to show that any trust that was declared by the game to be dishonest remains dishonest even after a new identity key is generated.

The core logics of EasyCrypt proved to be expressive enough and no exensions to these logics were needed to complete the proofs. However, for convenience during the development, we introduced a few new features that helped reduce the proof effort resulting from the unprecedented scale of this project. These new features do not enlarge the Trusted Computing Base (TCB) of EasyCrypt, which is composed of a set of base tactics defining the EasyCrypt core logics. Indeed, all the added features generate internal proof trees that only rely on the core tactics. Hence, no bug in the added features could lead to the acceptance of an invalid proof tree: this would be rejected by the TCB.

Management of pre- and post-conditions Several core tactics require users to provide intermediate assertions, e.g. loop invariants. When dealing with complex programs and specifications, writing such intermediate assertions becomes cumbersome and error-prone. However, in a majority of cases, these assertions can be expressed with little work from the current pre- and post-conditions (which tend to grow in size). We added the possibility to match sub-formulas that appear in the current proof goal and use these sub-formulas for new assertions. Doing this greatly decreases the proof writing effort and makes the proof script more robust, as the new assertion is given as a delta from the active proof goals. It also provides more readable proof goals.

Proof automation Several core tactics of EasyCrypt generate proof obligations that code is lossless, i.e. that it terminates with probability 1. We have implemented heuristics to deal with such goals automatically.

We have also improved existing automation to chain applications of core tactics, and tuned the implementation of some core tactics. In particular we have integrated an automatic version of the frame rule, which removes parts of the post-condition that are immediately implied by the pre-condition.

7 Lessons learned

In addition to producing the machine-checked specifications and security proofs for KMS DMP that we described in this paper, this project was also an oportunity to better understant the challenges posed by larger-scale developments to computer-aided cryptographic provable security, and particularly when using EasyCrypt. We now give an overview of the main take-away lessons we got from the project.

Imperative vs Functional specification One of the strong points of EasyCrypt is that there is a great deal of freedom in choosing how to formalize cryptographic primitives and security games. As security models become more intricate, a crucial decision is how to model the keeping of state, as there is usually a tension between game readability and the complexity of proof goals/invariants.

Favouring game readability means using EasyCrypt's imperative language as much as possible to describe the step-by-step actions that occur in each oracle call. This means using a dedicated EasyCrypt module to syntactically distinguish the behaviour of each entity (or type of entity) in the system, keeping each part of the game state as a separate local variable in the correct submodule, and using if and while statements to deal with control-flow. The consequence of this is that the top-level program that describes the security experiment displays a very complex control-flow, which makes proving the equivalences required for game-hopping harder (in particular because game hops typically require addressing particular cases that introduce additional branching points).

The alternative is to *flatten* the specificacions of security experiments by collapsing the module structure and moving as much of the detail as possible to EasyCrypt operators (functional

specifications of deterministic state transitions). This makes the specifications less readable, but it naturally provides a slicing between probabilistic statements, which model the cryptographic operations and associated control-flow (e.g., branching on a signature verification result) that are usually the crucial program actions in indistinguishability and bad-event hops, from branching and iteration statements that are only modeling state management operations (e.g., checking syntactic validity of a message and deciding where it should be dispatched).

In this project we have used both approaches and, in hindsight, the conclusion is that starting with a readable model that can be checked for soundness more easily and then flattening it as a first game hop is often the best choice.

Modular proofs using global invariants A related issue in managing proof complexity is identifying early on the abstraction boundaries that allow decomposing the proof into treatable self-contained intermediate goals. For some cases this is straightforward to do, namely for sub-components in the protocol that cryptographers naturally see as building-blocks (e.g., a signature service or a global hash function) that give rise to simple and well-understood global invariants in the security games. However, in this project we encountered protocol-specific semantically rich global invariants that permit (once correctly identified and formalized) not only to break the proof down into manageable subgoals, but also to simplify the top-level proof. Intuitively, we achieved this by: i. first specifying invariant $I = I_1 \wedge I_2 \wedge I_3 \ldots$ globally in, say game G_i; then ii. reducing the preservation of I in game G_i to a bad event occurring in a simpler *flattened* game, where the probability of bad is bounded at much lower cost; and iii. jumping to game G_{i+1} where the preservation of I is hardwired as checks of (and sometimes branching on) some sub-formula I_k directly in the code, where needed. The challenge here is to pin down the minimal use of I_k in the new game, so that the invariant does not need to be reproved, while keeping G_{i+1} as simple as possible in order to complete the proof more easily. As a side result of this design pattern, which we believe it is interesting to generalize in future work, game G_{i+1} now syntactically displays only the relevant parts of the established invariant in its code, which makes it easier to understand the context for each proof goal.

Proof effort Overall, from a rough analysis of the proof effort involved in this project our intuition is that the resources required to complete game hopping proofs by experienced EasyCrypt users, once the games and security invariants are correctly specified (and following good practices, some of which were highlighted above) scales "linearly with the complexity" of the programs/games, similarly to functional equivalence proofs between programs with a reasonably close, if not identical, control-flow.

8 Related Work

There has been significant work on the formal verification of cryptographic API, and in particular on PKCS tokens.

Delaune, Kremer, and Steel [16] model tokens as security protocols with a sole party, and apply Dolev-Yao verification methods to analyze their security. Bortolozzo, Centenaro, Focardi and Steel [12] build an automated tool for model checking the security and functionality of tokens, and evaluate commercially available PKCS tokens. They discover several security issues and validate patches.

Cachin and Chandran [13] formalize computational security of cryptographic APIs, and show security of PKCS tokens in their model. Kremer, Steel and Warinschi [18] define another model designed to provide more genericity and to support more powerful corruptions. Kremer, Künnemann and Steel [17] define a UC functionality for key management, and use their model for proving the

security of a minimal example. The model imposes constraints on the interactions between key management and key usage.

Shrimpton, Stam and Warinschi [23] also show that the indistinguishability definition of security for cryptographic APIs composes with a natural class of symmetric-key primitives, such as AEAD and MAC in order to provide cryptographic services. The security of these cryptographic services was defined by extending the attack vectors of standard AEAD and MAC security with all the oracles available to the cryptographic API adversary. We depart from this approach and adopt an alternative formulation for the security of cryptographic services, which is inspired by the Universal Composability framework (UC). As discussed in Appendix A, from UC we inherit two important advantages: i. clear composition guarantees of cryptographic services with other systems; and ii. a clean and intuitive view of correctness and security guarantees for the clients of cryptographic services.

Blanchet and Chaudhuri [11] use ProVerif to analyze a protocol for secure file sharing, which includes distributed key management features. Their analysis is conducted in the symbolic model of cryptography, which provides weaker guarantees, but allows for a higher degree of automation.

CryptoVerif [9] was among the first tools to support cryptographic security proofs in the computational model. It uses probabilistic process algebra as a modelling language and leverages different notions of equivalence to support proofs of equivalence, equivalence up to failure events, as well as simple forms of hybrid arguments. CryptoVerif has been used for verifying primitives, as well as protocols; it was recently applied to TLS 1.3 [10].

9 Conclusion

We proved a concrete security bound for the DMP of AWS KMS, down to standard cryptographic assumptions. The bound is tight, increasing linearly with the number of entities in the system, and it is machine-checked in EasyCrypt. For practical purposes, our work gives strong evidence that the DMP is as good as an ideal encryption service, under the assumption that any quorum of AWS operators that authorizes a domain update operation includes at least one honest operator.

Access to EasyCrypt code

The EasyCrypt code is available from `gitlab.com/kmsver/kmsdmp`.

Acknowledgements

Manuel Barbosa was supported by grant SFRH/BSAB/143018/2018 awarded by the Portuguese Foundation for Science and Technology (FCT). Vitor Pereira was supported by grant FCT-PD/BD/113967/201 awarded by FCT. This work was partially funded by national funds via FCT in the context of project PTDC/CCI-INF/31698/2017.

References

1. Michel Abdalla, Mihir Bellare, and Phillip Rogaway. DHIES: An encryption scheme based on the Diffie-Hellman problem. Contributions to IEEE P1363a, September 1998.
2. Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In David Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 143–158. Springer, Heidelberg, April 2001.

3. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time MEE-CBC. In Thomas Peyrin, editor, *FSE 2016*, volume 9783 of *LNCS*, pages 163–184. Springer, Heidelberg, March 2016.
4. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. A fast and verified software stack for secure function evaluation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1989–2006. ACM, 2017.
5. Amazon Web Services (AWS). AWS Key Management Service Cryptographic Details, August 2018. `https://d1.awsstatic.com/whitepapers/KMS-Cryptographic-Details.pdf`.
6. Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In Alessandro Aldini, Javier López, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.
7. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 71–90. Springer, Heidelberg, August 2011.
8. Mihir Bellare, Alexandra Boldyreva, and Jessica Staddon. Randomness re-use in multi-recipient encryption schemeas. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 85–99. Springer, Heidelberg, January 2003.
9. Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *2006 IEEE Symposium on Security and Privacy*, pages 140–154. IEEE Computer Society Press, May 2006.
10. Bruno Blanchet. Composition theorems for cryptoverif and application to TLS 1.3. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 16–30. IEEE Computer Society, 2018.
11. Bruno Blanchet and Avik Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pages 417–431. IEEE Computer Society, 2008.
12. Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing PKCS#11 security tokens. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 10*, pages 260–269. ACM Press, October 2010.
13. Christian Cachin and Nishanth Chandran. A secure cryptographic token interface. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 141–153. IEEE Computer Society, 2009.
14. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
15. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.
16. Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal analysis of pkcs#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*, pages 331–344. IEEE Computer Society, 2008.
17. Steve Kremer, Robert Künnemann, and Graham Steel. Universally composable key-management. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 327–344. Springer, Heidelberg, September 2013.
18. Steve Kremer, Graham Steel, and Bogdan Warinschi. Security for key management interfaces. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 266–280. IEEE Computer Society, 2011.
19. Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM.
20. Phillip Rogaway. Formalizing human ignorance. In Phong Q. Nguyen, editor, *Progress in Cryptology - VIETCRYPT 06*, volume 4341 of *LNCS*, pages 211–228. Springer, Heidelberg, September 2006.
21. Victor Shoup. A Proposal for an ISO Standard for Public Key Encryption (version 2.1), 2001. `https://www.shoup.net/papers/iso-2_1.pdf`.
22. Victor Shoup. A proposal for an ISO standard for public key encryption. Cryptology ePrint Archive, Report 2001/112, 2001. `http://eprint.iacr.org/2001/112`.
23. Thomas Shrimpton, Martijn Stam, and Bogdan Warinschi. A modular treatment of cryptographic APIs: The symmetric-key case. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 277–307. Springer, Heidelberg, August 2016.

A Relation to Universal Composability

Although we have not formalized it (syntactically) in this way, our security model and proof can be recast in terms of Universal Composability with global setup [15]. This is a framework that extends Universal Composability to enable set-up assumptions such as common reference strings, or public-key infrastructures (PKI). In this setting we would consider three types of parties.

- Hosts, which provide the environment $\mathcal{Z}$ with the high-level interface of AEAD encryption and decryption and enable $\mathcal{A}$ to install trusts.
- HSMs, which enable $\mathcal{A}$ to update trusts and manage keys, as well as perform encryption and decryption operations based on requests placed by hosts.
- Operators, which enable $\mathcal{Z}$ to generate key attestation statements for operators and HSMs.

The proof assumes static-corruptions, whereby some parties are known to be corrupt from the beginning of computation.

The proof of security in the UC framework would be carried out in a hybrid model. The first hybrid functionality is a confidential and authenticated channel through which hosts place encryption and decryption requests to HSMs. We emphasise that this hybrid functionality is not used by the KMS DMP protocol in any of the domain management operations, which are the focus of our analysis; it captures only the architectural choice that KMS hosts collect end-user encryption/decryption requests and forward them to HSMs (see footnote 13). Future work will extend the proof to consider the specific secure channel used by KMS for this purpose, which builds on top of the DMP itself.

Additionally the hybrid model includes two set-up assumptions. The first set-up assumption is a standard PKI functionality that is used only by operators, and which registers the identity keys of genuine entities in the system; this models the real world operation of KMS DMP in which operators are assumed to know the set of genuine entities. A crucial aspect of the security guarantees we prove for KMS DMP is precisely that this resource is restricted to operators, and still it suffices to guarantee that trust honesty is preserved by host and HSM actions.

The second set-up assumption captures the global quorum invariant that in any group of n genuine operators, there exists at least one non-corrupt operator. This assumption can be modeled via a simple hybrid functionality that is used only by hosts and HSMs. This functionality takes a set of operator identities and returns a single bit; the bit will be true if either the set contains a non-genuine operator or, if this is not the case, if the invariant is satisfied. By accessing this functionality, the description of the DMP protocol executed by hosts and HSMs can be instrumented (as a modeling artefact) to truncate execution traces where the global invariant is violated.

While the PKI set-up assumption is standard for many real-world systems, the second set-up assumption reflects a different kind of trust assumption that arises in modeling the actions of human operators in the UC setting.

The implication of recasting our result in the gUC framework is that we obtain composability: any higher-level protocol that can be proved secure by relying on the ideal encryption functionality we defined in the main body of the paper will still be secure when this functionality is replaced by the DMP protocol, assuming of course that the set-up assumptions described above are satisfied and that the deployed cryptographic schemes are secure.

B Extensions to the security proof

B.1 Access to keys in dishonest tokens

The security model in the main body of the paper underspecifies what happens if an attacker is able to maul a non-genuine identity into an honest trust, or convince an honest host to accept a dishonest

trust: what it states is that, *even if that were possible*, the attacker should not be able to break the secrecy of domain keys associated with that trust. This is somewhat counterintuitive and, indeed, our security proof establishes the stronger result that no honest trust will ever be successfully updated to another one inhabitted by non-genuine identities. For this reason, our machine-checked proof is actually carried out in a stronger security model.

In this model we extend the adversarial interface in the ideal functionality to make it explicit that, whenever the adversary succeeds in causing any of the events above, it can successfully launch a distinguishing attack separating the real world from the ideal world. We model this by extending the ideal functionality with two oracles (BadEnc and BadDec) that check if a trace satisfies a predicate honest(trace, tk, hdl, cmd) and, if this is *not* the case, they execute TkReveal to obtain secret key key and answer the adversary's request by encrypting or decrypting the provided payload.

Note that, if the honest predicate is trivially true, then the BadEnc and BadDec oracles are useless to an adversary. Conversely, excluding traces from the honest predicate explicitly rules out protocols that allow non-honest traces to occur, as the attacker can trivially distinguish the world from the ideal world by decrypting the ciphertext provided to the environment by the ideal functionality.

Our strenghthening maps to the indistinguishability definition, where we introduced the InsideR oracle; this works as a CCA-style oracle on dishonest tokens: it allows an attacker to reveal all keys in tokens associated with dishonest trusts and, technically, it maps to the CCA security of the multi-recipient encryption scheme used by the KMS DMP. In the reduction from the UC-style definition to the indistinguishability definition, the BadEnc and BadDec oracles can be simulated using InsideR to obtain the key and then perform the encryption and decryption operations.

B.2 Stronger corruption models

Although we have not formalized these extensions, it is easy to see that our proof can be easily adapted to deal with two (apparently) stronger corruption models. The first strengthening refers to the possibility of using corrupt keys, i.e., allowing an attacker to request encryptions and decryptions on keys it has corrupted (prior to legitimate usage, of course). In this case, the ideal functionality would provide an encryption/decryption of the actual payload/ciphertext provided by the attacker. This has no impact in the proof of security, as the reduction to indistinguishability can be extended to cache corrupted keys and use them to simulate these extra calls. The second strenghtening refers to dealing with corrupted hosts, which might be accepting dishonest trusts and/or revealing client payloads to the attacker; this case is similar to the previous one, in that the ideal functionality would need to be extended to keep track of whether these corrupt hosts were dealing with honest or dishonest trusts, and provide ideal or real encryption accordingly.

C Example invariant in DMP proof

We present here the core invariant that supports the intermediate step in the proof of KMS DMP security, where global invariants stemming from cryptographic security guarantees provided by low-level components are combined with inductive properties related to trust honesty. This invariant permits separating honest (protected) trusts from those who may be under adversarial control and proving that the adversary cannot gain control over domain keys that are initially associated to an honest trust.

The invariant relies on signedTr and signedTk predicates that allow leveraging the authentication guarantee provided by the signature scheme used by HSMs. Similarly, the properties of the parentTrust

relation permit relying on the injectivity of trust fingerprints, which is (computationally) guaranteed by a collision-resistant hash function.

```
(* if a trust is installed, then it is protected *)
(∀ hstid, hstid ∈ HstPolSrv.hosts_tr ⇒
   oget HstPolSrv.hosts_tr[hstid] ∈
        RealTrustService.protectedTrusts){2} ∧
(* if a trust is protected, all its members are genuine *)
(∀ t, t ∈ RealTrustService.protectedTrusts ⇒
      all_genuine OpPolSrv.genuine t){2} ∧
(* if a trust has a parent, then it is not initial *)
(∀ t, t ∈ RealTrustService.parentTrust ⇒
       ¬tr_initial t){2} ∧
(* all signed installable non initial tokens have a parent *)
(∀ (tk : Token), ¬tr_initial tk.`tk_trust ⇒
   tk.`tk_inst ⇒ signedTk tk RealSigServ.qs ⇒
     tk.`tk_trust ∈ RealTrustService.parentTrust){2} ∧
(* all signed tokens with genuine trusts have signed trusts *)
(∀ tk, signedTk tk RealSigServ.qs ⇒
      all_genuine OpPolSrv.genuine tk.`tk_trust ⇒
        signedTr tk.`tk_trust RealSigServ.qs){2} ∧
(* if a trust has a parent that is all genuine, then the parent
   was signed in an installable token *)
(∀ t, t ∈ RealTrustService.parentTrust ⇒
  all_genuine OpPolSrv.genuine
    (oget RealTrustService.parentTrust[t]) ⇒
      signedTr (oget RealTrustService.parentTrust[t])
           RealSigServ.qs){2} ∧
(* if a trust is protected, then it was signed
   in an installable token *)
(∀ t, t ∈ RealTrustService.protectedTrusts ⇒
   signedTr t RealSigServ.qs){2} ∧
(* if a trust was signed in an installable token,
     all its members are genuine and has
   an initial trust, then it is protected *)
(∀ t, tr_initial t ⇒ signedTr t RealSigServ.qs ⇒
      all_genuine OpPolSrv.genuine t ⇒
      t ∈ IOperatorActions.trusts ⇒
      oget IOperatorActions.trusts[t] ⇒
      t ∈ RealTrustService.protectedTrusts){2} ∧
(* if a token is signed with a trust where all its members
   are genuine and has an initial trust, then this trust
   was installable signed. *)
(∀ (tk : Token), tr_initial tk.`tk_trust ⇒
     signedTk tk RealSigServ.qs ⇒
      all_genuine OpPolSrv.genuine tk.`tk_trust ⇒
      tk.`tk_trust ∈ IOperatorActions.trusts ⇒
      oget IOperatorActions.trusts[tk.`tk_trust] ⇒
      signedTr tk.`tk_trust RealSigServ.qs){2} ∧
(* if a trust is protected and is not initial then it has
   a parent and this protected as well *)
(∀ t, ¬tr_initial t ⇒
  t ∈ RealTrustService.protectedTrusts ⇒
      (t ∈ RealTrustService.parentTrust ∧
        oget RealTrustService.parentTrust[t] ∈
            RealTrustService.protectedTrusts)){2} ∧
(* if a trust has a protected parent then it is protected as well *)
(∀ t, t ∈ RealTrustService.parentTrust ⇒
  oget RealTrustService.parentTrust[t] ∈
    RealTrustService.protectedTrusts ⇒
      t ∈ RealTrustService.protectedTrusts){2} ∧
(* Encryptions *)
(∀ (tk : Token), signedTk tk RealSigServ.qs{2} ⇒
    tk.`tk_trust ∈ RealTrustService.protectedTrusts{2} ⇒
 tk ∈ KMS_Procedures3.wrapped_keys{2} ∧
 tk.`tk_wdata.`tkw_ekeys ∈
   mencrypt (proj_pks (tr_mems tk.`tk_trust)) (encode_tag tk.`tk_trust)
     (encode_ptxt (oget KMS_Procedures3.wrapped_keys{2}[tk])))
```