Cryptographically Verified Design and Implementation of a Distributed Key Manager

Tolga Acar[*],*Cédric Fournet*[†],*Dan Shumow*[*]
[*] *Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA*
{tolga,danshu}@microsoft.com
[†] *Microsoft Research, 7 J J Thomson Ave, Cambridge CB3 0FB, UK*
fournet@microsoft.com

Abstract

We present DKM, a distributed key management system with a cryptographically verified code base.

DKM implements a new data protection API. It manages keys and policies on behalf of groups of users that share data. To ensure long-term protection, DKM supports cryptographic agility: algorithms, keys, and policies can evolve for protecting fresh data while preserving access to old data. DKM is written in C# and currently used by several large data-center applications.

To verify our design and implementation, we also write a lightweight reference implementation of DKM in F#. This code closes the gap between formal cryptographic models and production code:

- Formally, the F# code is a very precise model of DKM: we automatically verify its security against active adversaries, using a refinement type-checker coupled with an SMT solver and new symbolic libraries for cryptographic agility.
- Concretely, the F# code closely mirrors our production code, and we automatically test that the corresponding C# and F# fragments can be swapped without affecting the runtime behavior of DKM.

To the best of our knowledge, this is the largest cryptographically-verified implementation to date. We also describe several problems we uncovered and fixed as part of this joint design, implementation, and verification process.

I. Introduction

Securing data at rest is a challenging problem that has become increasingly important. While numerous protocols are routinely deployed to protect data in transit over networks, such as SSL/TLS, IPsec, and SSH [15, 19, 20, 29], security for stored data, especially shared stored data in large and distributed systems, is much less advanced.

Enterprise IT departments used to assume that data on the company intranet was secure as long as it was hidden behind firewalls and VPNs, Cloud computing clearly invalidates this naive assumption: in outsourced infrastructures, data security necessarily involves cryptography. However, encrypted storage requires users to manage their keys. Thus, cryptography shifts the burden from protecting data to protecting keys in distributed systems. Managing cryptographic keys remains one of the hard problems in applied cryptography. Without proper key management, an otherwise theoretically secure system is in reality quite vulnerable. For example, TLS and IPsec are insecure without a proper PKI [15, 19, 20], and SSH is vulnerable to man-in-the-middle attacks without trusted public keys [29]. Similarly, NIST identified several difficulties for storage encryption [28]: managing access to keys in storage, cryptographic algorithm and key length selection, and notably, the ease of solution updates as stronger algorithms and keys become available.

To address the increasing demand for secure persisted data in distributed environments, we develop a Distributed Key Manager (DKM). DKM operations rely on cryptographic keys, policies, and access control information, stored in a distributed repository. Our goal is not to create a new authentication mechanism or a secure key repository. On the contrary, we build on existing authentication, authorization, and storage methods, such as Active Directory and SQL.

The verified DKM implementation is far from being an academic exercise. DKM has been integrated into several large products and services of a large software company; some of them are data center applications that implement underpinnings of the now-pervasive "cloud" where flexible cryptographic policies are required to service the diverse needs of customers. Multiple business groups have shipped DKM in boxed products and deployed them in large data centers in various hosted service environments. We describe three such real-life

scenarios in Section II.

Data Protection APIs. The Windows operating system features a 10-year-old interface (DPAPI) providing local data protection services to user and system processes [26]. DPAPI does not share keys across users or machines—a severe restriction for data centers. We design DKM to provide a similar service to multiple users across multiple machines.

DKM is not a cryptographic API, such as CAPI [24], CNG [25], PKCS#11 [27] or KeyCzar [14]. CAPI, CNG, and PKCS#11 are APIs exposing cryptographic primitives through well-defined interfaces. KeyCzar tackles cryptographic misuse by hiding details such as key lengths and algorithms, and by providing safe defaults through a simple interface. Unlike DKM, KeyCzar does not automatically manage keys or provide agility. By design, DKM does not expose cryptography: it provides a *data protection* interface. Various cryptographic parameter decisions, key management and sharing across users and machines, and key lifecycle management are abstracted away from the user [2]. We describe the DKM architecture in Sections II and V.

Cryptographic Agility. Over time, cryptographic algorithms become obsolete and keys become too short for a desired security level. For example, DES [17] was the primary encryption algorithm a couple of decades ago. It was later replaced by 3DES and eventually AES. MD5 and SHA1 were the primary hash algorithms until major vulnerabilities were disclosed [32, 31], and an industry-wide effort started to replace them with SHA2. However, retiring a cryptographic algorithm is easier said than done. Disabling weak algorithms may improve security in theory, but may also cause data loss and renders real-life systems unusable, an unacceptable trade-off for many products.

We design DKM to be cryptographically agile: insecure algorithms can be replaced easily while maintaining backwards compatibility. DKM can generate and distribute fresh keys upon each algorithm change, and provides support for data re-encryption; this is recommended, but often not tenable in practice: keys may be derived from unalterable hardware-provided secrets or from re-usable passwords; key distribution may be slow or expensive; besides, operational oversights may lead to accidental key re-use. Thus, DKM embeds defensive key-derivation mechanisms to support (apparent) key re-use across algorithms, even broken algorithms. We present our cryptographic constructions in Section III.

Verifying DKM design and implementation. Despite their technical merits, formal tools for cryptography are seldom applied to the design and implementation of new protocols and APIs. Our formal work shows how to close the gap between cryptographic models and concrete production code, by applying verification tools on a large scale, early in the design and implementation process. Our method relies on developing *verified reference implementations* [6, 18, 11], also known as executable specifications, using general-purpose programming languages (rather than, say, rewriting rules or process calculi). Thus, we obtain a very precise formal specification of the product, shared between developers, security experts, and automated verification tools.

Compared with abstract models distilled from informal specifications, reference implementations also benefit from existing software tools, such as programming environments, compilers, typecheckers, debuggers, and test harnesses. Pragmatically, the functional properties of the model can be automatically tested against production code. Conversely, when there is an ambiguity, one can quickly assemble new security tests from the reference code, run them against production code, observe the result, and fix one of the two.

We implement DKM in C# and F# (a variant of ML). The production code is entirely written in C#. Additional F# code covers the security-critical parts of the implementation, rewritten in a more functional style to enable its automated cryptographic verification using F7 [8], a refinement typechecker coupled with Z3 [13], an SMT solver. We obtain in particular a full-fledged, verified reference implementation of DKM. (Other software verification tools independently apply to C# production code; they check simple safety properties, for instance excluding out-of-range access to arrays, or dereference on null objects; However, they do not handle the security properties verified in this paper, which involve active adversaries, concurrency, and cryptography.) We describe our C# and F# code in Sections V and VI.

Very few large cryptographically-verified reference implementations exist, notably for subsets of TLS [7] and CardSpace [9]. Their code was developed with specific verification tools in mind, independently of the original protocol design and implementation, so they treat prior production code as black boxes for basic interoperability testing. In contrast, we advocate a close collaboration between designers, developers and security reviewers, and we use a single code base for the two implementations. We take advantage of the fine-grained integration of C# and F# within the .NET framework to compose them at the level of individual methods: we precisely align each fragment of (formally verified) F# code to the corresponding fragment of (production) C# code. Both implementations share the same structure, along with the same typed interfaces. This complicates verification, but also yields stronger guarantees for the production code. Indeed, this task involves a careful code review, driven from the elaboration of a verified

model, with numerous automated verifications and tests between the two. In our experience, it led to the discovery of ambiguities in the DKM design, of delicate low-level issues in its implementation, and of a few serious vulnerabilities. Overall, we improved DKM and gained more confidence in its security, with a reasonable verification overhead: we wrote 5% additional F# code, and automated verification runs faster than our test suite.

Main Contributions. To our knowledge, this is the first time automated cryptographic verification is integrated in the design and production of a new security component deployed in large-scale commercial products. Also, to the best of our knowledge, this is the largest cryptographically-verified implementation to date.

We verify the design and implementations of DKM for a threat model that covers (in particular) our key deployed scenarios. We obtain formal security guarantees for F# reference code tightly linked to our C# production code. We discuss problems discovered as part of this hybrid verification process, as well as their fixes. We highlight that our efforts have a direct impact in the security of real products.

Contents. Section II presents DKM using a few deployment scenarios. Section III specifies its main cryptographic constructions and discusses cryptographic agility. Section IV defines its security goals (precisely, but informally). Section V gives selected C# implementation details. Section VI presents the reference F# implementation. Section VII evaluates our approach, discusses several weaknesses we identified and fixed, and concludes. For readers interested in the details of our formal verification we have provided appendices. Appendix A reviews our verification method. Appendix A gives selected modeling details. This paper presents only selected code excerpts and omits many details. The entire F# and F7 source code used for verification is available from http://research.microsoft.com/~fournet/dkm.

II. Distributed Key Manager

DKM involves three roles: *users*, a trusted *repository*, and untrusted *stores*; it operates on data organized into named groups of users that share the data. DKM is a library that runs on the users' machines, on their behalf, with their authenticated identity. As it reads and writes data, each user may call the DKM API to carry out *Protect* and *Unprotect* operations, of the form below:

blob := *Protect* (*group*, *data*)
data := *Unprotect* (*group*, *blob*)

Intuitively, *Protect* applies some authenticated encryption algorithm to the user data. The resulting ciphertext '*blob*' can then be persisted in untrusted storage. Conversely, *Unprotect* verifies and decrypts a given blob, and then returns the plaintext data to the user, or fails with a 'corrupted data' error. If the user does not belong to the named *group*, both calls fail with an 'access denied' error.

Each DKM group relies on a small amount of trusted data kept in the repository, essentially a collection of keys and a cryptographic policy. Repository access, effective cryptographic policy determination, and all cryptographic operations are hidden behind the DKM interface. The repository has client (requestor) and server (responder) components; it is used as a black-box by DKM on behalf of each user. Authorization is at group granularity and is enforced by the repository. Some users, called group administrators, can also add members or change its policy. This requires write access to the repository. Conversely, the untrusted stores are directly managed by the users; they may consist of file systems, SQL servers, or any other storage medium.

We describe two cloud scenarios with data-center servers acting as DKM users. These scenarios are simplified from real life deployed products that use DKM.

Scenario 1. Hosted E-Mail Server. Consider a cloud e-mail service hosted on multiple servers. The e-mail servers share the same set of keys for protecting mailbox contents. They offer offer *message aggregation*, fetching mail from third-party service providers on behalf of their users, and storing them in user mail boxes. Third-party services require a user name and password for authentication, so it is necessary for the e-mail servers to have access to these credentials. The data center hosts e-mail for multiple tenants. Each tenant has an e-mail administrator to remotely manage that tenant's e-mail settings. One threat is that these e-mail administrators may attempt to steal a user's plaintext credentials. Another threat is the data center personnel who may access a user's plaintext credentials.

To address these threats, a data center administrator creates a DKM group to protect e-mail credentials and grants read/write access only to the e-mail servers. The DKM repository denies e-mail users, e-mail administrators, and data center operations personnel access to DKM groups with the exception of a few trusted data center administrators. The typical e-mail fetch workflow starts with an e-mail server reading a user's encrypted credentials (labeled (1) on Figure 1). The e-mail server sends a decryption request to DKM and receives the user's plaintext credentials. In step (3), the e-mail server authenticates to the external server.

Scenario 2. Offloading Web Server State. In a large scale deployment with thousands of servers, sometimes spread across multiple data centers, any server affinity to session states is a huge impediment to scalability

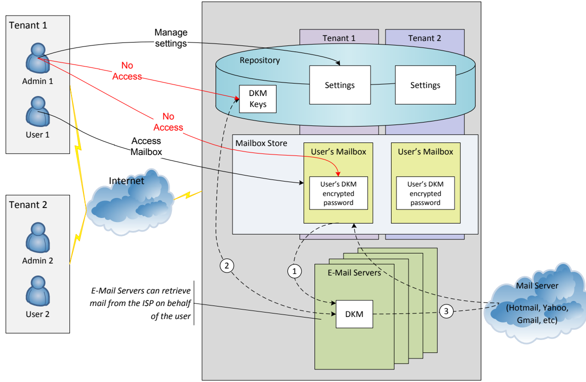

Fig. 1. DKM hosted e-mail scenario.

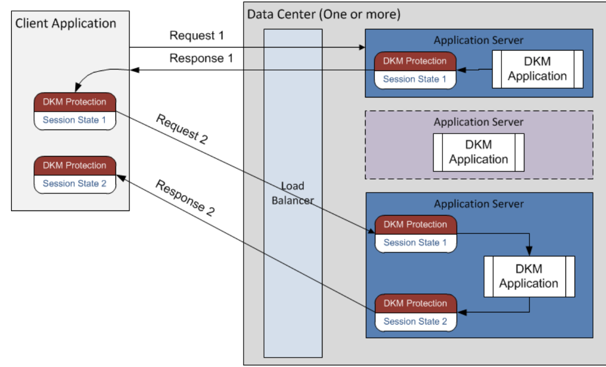

Fig. 2. Web server state offloading scenario.

and reliability. A common approach is to make the servers stateless, and instead pass any session state back and forth between the client and the server. The session state may contain user credentials, such as forms-based authentication data, as well as data-center-specific sensitive information. This state must also be preserved across multiple requests serviced by different servers. The threat is that a client may read and modify his session state, or even the session state of other clients. Figure 2 depicts the data flow with two requests where session state 1 is created on one server, and is read and modified to session state 2 on another server.

TLS provides adequate security to session state transmitted over the network. However, the session state must not be disclosed to the user and must not be modified by the user. Naive attempts to use CBC encryption have been proved vulnerable in theory and in real life [30, 21]. Our solution uses DKM on data center servers to protect session states. Thus, any server that belongs to the DKM group associated with the service can authenticate, encrypt, and decrypt the state, while any client (presumably not member of the group) are unable to read or modifiy it. This approach has been successfully implemented in a large hosted services offering and contributed to better server utilization.

III. Cryptographic Agility

We define cryptographic agility as the capability to securely use different sets of algorithms with the same keys. Securing persisted data highlights the need for agility, as the data may outlive the actual security of any cryptographic algorithms and keys. Cryptographic agility in DKM keeps the long-term key secure even if one or more of the algorithms that used the key are compromised: a critical property we implement and formally verify. Our constructions are based on those initially proposed by Acar et al. [1].

Cryptographic Algorithms. DKM policies are parameterized by two kinds of symmetric-key algorithms.

- We let E range over encryption algorithms, and write D for the corresponding decryption algorithm. We currently use AES in CBC mode. Similarly, we let E_{ae} range over authenticated encryption algorithms and a proper D_{ae} for an authenticated decryption algorithm. We currently use AES-GCM [23].
- We let H and H_d independently range over keyed hash algorithms. We currently use HMAC [4, 3] with either SHA-256 or SHA-512.

Each algorithm has a fixed key length, written $|E|$, $|E_{ae}|$, $|H|$, and $|H_d|$, respectively. Each algorithm also has a global, unambiguous algorithm identifier, written $\langle E\rangle$, $\langle E_{ae}\rangle$, $\langle H\rangle$, and $\langle H_d\rangle$. In our implementation, we use DER (Distinguished Encoding Rules) encoded object identifiers.

To provide secure and flexible cryptographic algorithm support, DKM is parameterized by a whitelist of cryptographic algorithms that meet target security properties. The algorithm list can be adapted in future implementations to allow new or custom algorithms.

DKM Construction. Let $\mathsf{es} = (\mathsf{Kg}_{\mathsf{es}}, \mathsf{Kd}_{\mathsf{es}}, \mathsf{Enc}_{\mathsf{es}}, \mathsf{Dec}_{\mathsf{es}})$ represent an authenticated encryption scheme that consists of a key generator $\mathsf{Kg}_{\mathsf{es}}$, a key derivation function $\mathsf{Kd}_{\mathsf{es}}$, and authenticated encryption and decryption algorithms $\mathsf{Enc}_{\mathsf{es}}$ and $\mathsf{Dec}_{\mathsf{es}}$.

The algorithms $\mathsf{Enc}_{\mathsf{es}}$ and $\mathsf{Dec}_{\mathsf{es}}$ provide authenticated encryption. We use the term *authentication method* to describe a particular construction DKM supports: Encrypt-then-Mac, Mac-Then-Encrypt, or an authenticated encryption mode AES-GCM. While we prefer the less frequently fielded AES-GCM as an authenticated encryption algorithm, the CBC and HMAC based methods are necessary to attain a much larger deployment base. The scheme is parameterized by a *cryptographic policy* p that determines the method to use and its

algorithms: either an authenticated encryption algorithm E_{ae}, or a block cipher in CBC mode composed with HMAC, using two algorithms E and H. Additionally, the cryptographic policy indicates which hash algorithm H_d to use with the key derivation function.

The key derivation function KDF uses a pseudo-random function in a prescribed construction [22]. In this implementation, KDF is the SP800-108 KDF in counter mode [12] and the pseudo-random function is an HMAC, specified as algorithm H_d in the policy and keyed with a group key k.

We write k for a DKM key generated as $k = \mathsf{Kg}_{\mathsf{es}}(L_k)$, where L_k indicates the length of k and is a function of the policy:

$$L_k = \begin{cases} \max(|H_d|, |E_{ae}|) & \text{when using } E_{ae} \\ \max(|H_d|, |E|, |H|) & \text{when using } E \text{ and } H \end{cases} \tag{1}$$

When using E_{ae}, we write $k_{es} = k_{ae}$ for the authenticated encryption key. When using E and H, we write k_e and k_m for the encryption and MAC keys, respectively, and let $k_{es} = k_e | k_m$ be their concatenation. The authenticated encryption key length L_{es} is a function of the policy:

$$L_{es} = \begin{cases} |E_{ae}| & \text{when using } E_{ae} \\ |E| + |H| & \text{when using } E \text{ and } H \end{cases} \tag{2}$$

The key derivation function KDF yields a key k_{es} of length L_{es}

$$k_{es} = KDF(H_d, L_{es}, k, R, M_{es} | I_{es}) \tag{3}$$

where k is the DKM key, R is a fresh random nonce, M_{es} is the authentication method, and I_{es} is an unambiguous DER encoding of the algorithm identifiers in the policy: I_{es} is either $\langle E_{ae} \rangle$ or $\langle E \rangle | \langle H \rangle$. (We will show the motivation for including M_{es} in the key derivation.)

Achieving Agility. To achieve cryptographic agility, the finite set of algorithms must "fit together", a notion formally introduced in [1]. One example of a condition "fitting together" is that the encryption scheme es must have a unique encoding. We "fit together" our scheme with a small set of authenticated encryption algorithms chosen from an agile set and a fixed PRF (Pseudo Random Function) constructed from HMAC with SHA-256 and SHA-512. Since there are more authenticated encryption algorithms than PRFs in practice, this is not a severe restriction.

Agile Encoding. How do we represent the set of algorithms and parameters in an agile implementation? To provide cryptographic agility at encryption time, our implementation relies on a cryptographic policy and a key stored in the repository. This allows an administrator to configure the cryptographic policy and update the key as needed. The *Protect* operation unambiguously encodes the cryptographic policy and the key identifier in the ciphertext. We use DER-encoded OIDs to represent algorithms and Guid to represent the key identifier in the encoded ciphertext blob. The *Unprotect* operation first decodes and validates the algorithms and parameters from the ciphertext, then fetches the authenticated-decryption key from the repository. This decoded policy may differ from the latest policy stored in the repository. Section VII-A describes a problem with agile encoding found by this verification effort.

Crypto-Agile DKM. With these necessary building blocks, we present the DKM functions for authenticated encryption (called *Protect*) and decryption (called *Unprotect*):

- *Protect.* If the user has at least read access to the DKM group, the current group key k and policy p are read from the repository. An L_{es} bits long authenticated encryption key k_{es} is derived from the key k. Figure 3 gives the encryption steps to produce C. Finally, the formatted ciphertext is given by the equation

$$\mathit{blob} = V | \langle p \rangle | G_k | R | IV | L_H | L_C | C \tag{4}$$

  where $|$ is concatenation, V is a 32-bit version number, $\langle p \rangle = M_{es} | \langle H_d \rangle | I_{es}$ is the unambiguously encoded policy, $\langle H_d \rangle$ is the key-derivation PRF algorithm identifier, G_k is the 128-bit DKM key identifier, R is the random nonce, IV is the initialization vector for encryption, L_H is the MAC length, and L_C is the length of $|C|$ in Figure 3. The values G_k, R, and IV are DER-encoded to allow variable lengths.
- *Unprotect.* The protected blob is parsed as *blob* (Equation (4)) and validated: in addition to various length and buffer overflow length checks that include L_H and L_C, the algorithms H_d, E, and H are validated against the list of accepted algorithms. The G, R, and IV lengths are also validated: $|IV|$ must match the blocksize of E, and $|R|$ must match $|H_d|$. The decoded policy may differ from the current policy (e.g. when unprotecting an old ciphertext) and is *a priori* untrusted. Only after a successful validation, the decryption is carried out and the decrypted message is returned if the MAC verifies. The key identifier G_k is used to fetch the DKM group key k from the repository. If a key is found, it is fed to the key derivation and decryption algorithms to calculate the plaintext message M and return it to the user.

Security Challenge. There are two main paths to verify as regards cryptographic agility: encryption and decryption. The encryption uses algorithms in the policy and

Policy p	Derivation from key k and R	Operations
EncryptThenMAC(H_d, E, H)	$k_e \| k_m = KDF(H_d, L_{es}, k, R, M_{es} \| I_{es})$	$C_e = E(k_e, IV, \mathit{text})$ $\mathit{mac} = H(k_m, IV \| C_e)$ $C = C_e \| \mathit{mac}$
MACThenEncrypt(H_d, E, H)	$k_e \| k_m = KDF(H_d, L_{es}, k, R, M_{es} \| I_{es})$	$\mathit{mac} = H(k_m, \mathit{text})$ $C = E(k_e, IV, \mathit{text} \| \mathit{mac})$
GCM(H_d, E_{ae})	$k_{ae} = KDF(H_d, L_{es}, k, R, M_{es} \| I_{es})$	$C = E_{ae}(k_{ae}, IV, \mathit{text})$

Fig. 3. DKM key derivations and encryption methods.

must derive a new k_{es} with KDF from k, a fresh R, and I_{es}. At decryption time, I_{es} and R are decoded from the ciphertext. In both cases, I_{es} is compared against known set of algorithms (the white list) to prevent an insecure algorithm from being used. Verification must thus account for algorithmic choices. To this end, our formal model defines two logical predicates on policies, *Authentic*(*p*) and *Confidential*(*p*), that explicitly state sufficient security conditions on their algorithms (within our symbolic model) to ensure authenticity and confidentiality, respectively, for any data protected with a correctly generated group key that remains secret. (The next sections also deals with key compromises.)

For example, paraphrasing its logical definition, the predicate *Authentic*(*p*) says that, in policy p, H_d is a pseudo-random function; and, if p selects authenticated encryption, then E_{ae} is a secure authenticated encryption algorithm, otherwise, H is a hash function resistant to existential forgery chosen-message-attack and, moreover, if p selects Encrypt-then-MAC, then E is at least functionally correct (that is, it correctly decrypts any correctly-encrypted message, to prevent data corruption after authentication). We refer to Section A for additional modelling details.

IV. Security

We give a precise but informal descriptions of our verified security properties and attacker model for DKM. The formal counterpart is discussed in Section A.

Goals and Non-goals. DKM provides data integrity, data secrecy, and authentication for the group name and the key policy. On the other hand, DKM does not attempt to protect privacy information associated with group membership or policies, or with traffic analysis (e.g. the number and size of encrypted blobs).

Our verification effort concerns only security properties. For instance, we do not verify that *Unprotect* always succeeds on the output of *Protect* when both users have access to the group (although we believe that our model could be extended accordingly).

Access Levels. DKM distinguishes four access levels and provides an interface to manipulate them. These levels are enforced by the repository implementation. For each group, they control whether a user can read and write keys and policies, or manage other users, as detailed below.

1) NO ACCESS. This is the default for all users.
2) READ ONLY. Users with read-only access to the DKM group in the repository can read its cryptography policy and its keys; thus, they can use DKM both for reading (unprotect) and writing (protect) group data.
3) READ/WRITE. Users that also have write access to the group can additionally update the cryptographic policy, create new keys, delete existing keys, and delete the group.
4) OWNER. Users that own a DKM group (also known as group administrators) have full access to the group. They can additionally change user access levels.

DKM also provides an interface for managing keys and policies. For policies, for instance, a user may read and write a DKM group policy using calls of the form

policy := *ReadPolicy*(*group*)
WritePolicy(*group*, *policy*)

In addition, after any successful call to *Unprotect*, a user can read the policy that is part of the blob and used for unprotecting it. We call this policy the *unprotect policy*. As explained below, the user can inspect this policy to obtain finer authentication guarantees in case weak algorithms may have been used.

Adversary Model. We now define our active adversaries for DKM, to reflect our threat model for the purpose of verification. Our adversaries can:

- Read and write any protected blob. (We assume that a protected blob is stored in untrusted media accessible by adversaries.)
- Perform arbitrary computations using our (symbolic) cryptographic interface.
- Control users, asking them to protect plaintexts of their choice, to unprotect blobs of its choice, to create new groups, to manage groups, etc; to renew a key; or to change a policy.
- Corrupt users, thereby gaining their repository access and their capabilities (that is, depending on group membership, read any group key, or even

inject arbitrary values for group keys and policies into the repository).

In our model, "control" accounts for the unknown behavior of the service built on top of DKM. In the properties below, if the adversary asks a user to protect some text, this is formally recorded as a genuine plaintext for the group. In contrast, conversely, "corrupt" accounts for the potential partial compromise of the group users. If the adversary directly reads a key and runs the protect algorithm on some plaintext on its own, then the resulting blob is treated as a forgery.

Conversely, our adversaries cannot:

- Corrupt the repository: they have to comply with its access control policy. We thus entirely trust the repository and its secure channel facilities.
- Introduce their own broken crypto primitives—however, our model features a fixed collection of "broken" primitives to formally let the adversary build insecure policies.

Authentication. We now express the precise logical authentication guarantee we verified on DKM. (This guarantee can be simplified by making additional assumptions, for instance by excluding the compromise of any group member, or assuming that all cryptographic algorithms are secure, to obtain corollaries of our main verification results.)

For any run of DKM with any number of users and groups, and for any adversary, if a user a successfully unprotects a blob for a given group, then we have all the properties below.

1) user a had read access to the group;
2) the unprotect policy uses algorithms in the DKM whitelist;
3) one of the following holds: either (a) the resulting plaintext was protected by a user who had read access to the group, using this unprotect policy; or (b) a corrupted user had read access to the group; or (c) the unprotect policy does not have the "Authentic" property.

The "Authentic" property is defined from the security assumptions on the three security algorithms in the policy. Moreover, if a user a successfully protects a plaintext for a given group, then we have

1) user a had read access to the group.
2) the protect policy use algorithms in the DKM whitelist;
3) either the protect policy and key k have been proposed and correctly generated by a user who had write access to the group, or there is a corrupted user who had write access to the group.

Secrecy. If a user protects a plaintext for a given group, then one of the following holds: (a) the plaintext remains secret; or (b) the current group policy does not have the "Confidential" property; or (c) there is a corrupted user who had read access to that group, or (d) there is a user who had read access to that group that leaks plaintexts.

Example. Our property statements cover partial- compromise scenarios with dynamic group memberships and policies. As a simple common case for illustrating our results, consider a group whose owner uses the default policy and remains the single user. Our security guarantees boil down to: Unless the owner is compromised, any unprotected value is a previously-protected value, and their secrecy is preserved.

Limitations (Modeling). We do not capture some temporal properties of access control: our properties are conservatively stated in terms of users having access in the past, so for example they do not state sufficient conditions to render a group secure again once corrupt users have been expelled and new keys generated. We do not model group deletion, and may not entirely account for group creation—in our model, whoever first creates a group owns that group. Thus, we differentiate between initial deployment and run-time operations and focus on run-time aspects. However, we observe that the creation of a DKM group requires write access to the DKM container in the repository, which in practice is typically restricted to a small set of trusted network administrators.

V. Modular Software Architecture

Next, we outline the structure of the DKM implementation, with a selection of details relevant for the rest of the paper. Our production code is written in C# and distributed as a single DLL deployed with user applications. It was coded partly before our verification effort—we have released 5 minor revisions of DKM so far.

Figure 4 describes the modular structure of DKM as a UML class diagram. The API consists of a single public interface, named *IDKM*. DKM also uses internal interfaces, such as *IRepository* and *IAuthEncrypt*, and internal classes, such as *KeyPolicy* and *Key*. It relies on system libraries for all cryptographic algorithms [25] (omitted in the figure).

IDKM. The two main methods of the DKM API, outlined in Section II, are declared as follows:

```
public interface IDKM {
  (...)
  MemoryStream Protect(MemoryStream plaintext);
  MemoryStream Unprotect(MemoryStream ciphertext);
  KeyPolicy DecodedPolicy { get; } }
```

To access data, the user first creates an instance of DKM for a given group name. To store sensitive data, the

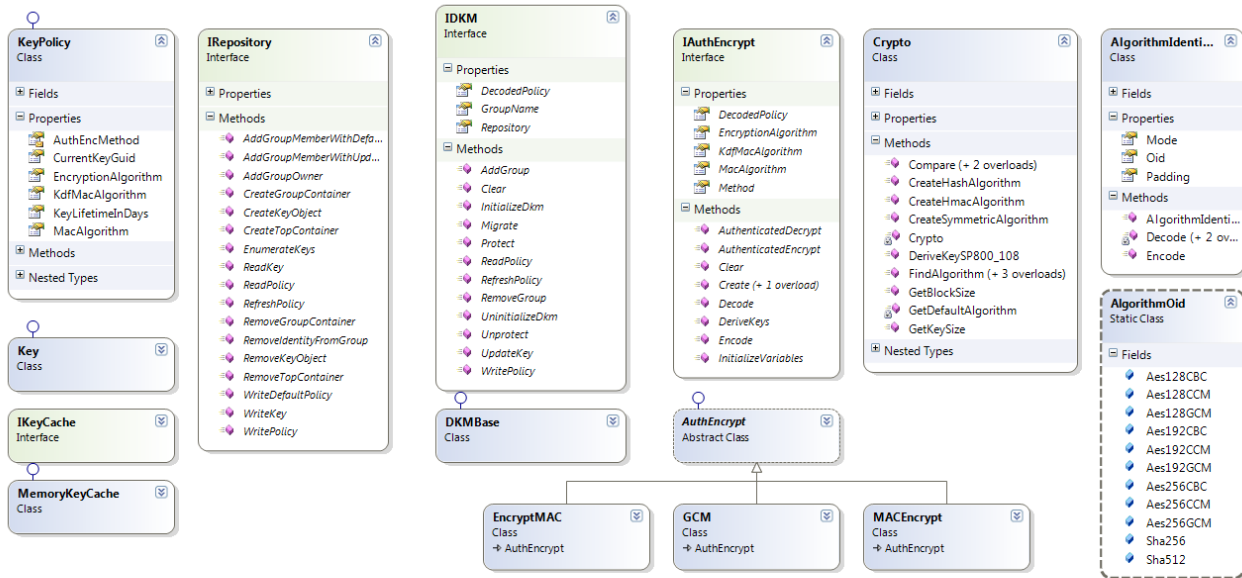

Fig. 4. DKM main interfaces (in green) and classes (in blue)

user writes to a plaintext stream, calls *Protect* to obtain a protected ciphertext stream, and then directs this stream to untrusted storage. (Streams are the primary I/O mechanisms in the .NET framework; they generalize the C++ ios class. The *MemoryStream* used in DKM is just a stream implemented over memory buffers.) Conversely, to retrieve data, the user opens a ciphertext stream from untrusted storage, calls *Unprotect* to obtain a plaintext stream, then reads data from it. In addition, the user can read the *DecodedPolicy* property to review the policy last used for unprotecting data. We omit a description of the other methods and properties of *IDKM* listed in the figure, which support group, policy, and key management, as well as data migration (an unprotect followed by a protect with the current group key and policy).

IRepository. The *IRepository* interface decouples the choice of a particular trusted repository from the rest of DKM. It has methods for accessing a secure container associated with each group name. The container maintains a small amount of trusted data for the group: an access control list for its users, its current protect policy, and a set of keys. (Keys used with earlier protect policies are usually retained to enable unprotection of old data.)

In addition, an internal *MemoryKeyCache* class provides a per-process in-memory cache for keys retrieved from the repository. These cached keys are kept encrypted using a per-process ephemeral key the Windows OS provides. The cache is consulted first on each key read request, and is cleared before the process exits.

DKM currently supports four implementations of *IRepository*, based respectively on Microsoft Active Directory, on Microsoft SQL Server, on NTFS/SMB, and on a simple in-memory database. All implementations rely on the current user's credentials for authentication and authorization purposes. For verification, we model only the in-memory repository, as we are mostly interested in the sequences of calls through the repository interface, rather than its implementation details.

IAuthEncrypt. The *IAuthEncrypt* interface encapsulates crypto-agile authenticated encryption and its state, which consists of the the policy to use, the random nonce, an IV for encryption, and the derived keys. Three subclasses, named *EncryptMAC*, *MACEncrypt*, and *GCM* (one for each authentication method) provide different implementations of the methods *AuthenticatedEncrypt*, and *AuthenticatedDecrypt* but share code for key derivation (*DeriveKey*) and for parsing and formatting the protected blobs (*Decode* and *Encode*).

Cryptographic Algorithms and Materials. The *Crypto* class provides all primitive cryptographic operations. such as block encryption and decryption and random generation. In addition, DKM defines a few auxiliary classes:

- *AlgorithmIdentifier* objects uniquely identify algorithms, using the *AlgorithmOid* constants.
- *Key* objects encapsulate cryptographic keys; their *KeyValue* property yields their (otherwise in-memory encrypted) key bytes; auxiliary properties record the key identifier and its intended algorithm.
- *KeyPolicy* objects state a particular choice of authentication method, of cryptographic algorithms (for key derivation, encryption, and authentication)

and of key identifier.

VI. Functional Implementation

Our verified reference-implementation code takes advantage of the modular structure of DKM and of the fine-grain integration between C# and F# within the .NET platform. Thus, our two implementations can call one another and share interfaces, classes, and objects.

We do not implement primitive cryptographic operations in F#; instead, we just call their C# implementations from F# and reflect their properties using type annotations and logical refinement at their interface. We end up with a hybrid C#—F# code base, with two implementations for the main classes and interfaces, and the ability to build executable code by selecting implementations for each of the three main interfaces.

The main purpose of the F# code is to bridge the gap between an imperative implementation and a functional model. Hence, our F# implementation avoids mutable fields and object oriented-features, whenever possible: some C# objects are coded as F# records, C# properties are coded as F# fields, and some data representations are simplified. Each F# implementation file is structured in two parts: (1) mostly-functional code, in the subset of F# verifiable by the F7 typechecker, followed by (2) a small object-oriented stub that implements the target C# interface, with implementation methods that mostly call plain F# functions. (These stubs are used, for instance, to leverage the existing test suites developed for the C# implementation.) Next, we give selected F# implementation details for the three main DKM interfaces. Section A provides the corresponding F7 declarations verified by typing.

DKM. This module re-implements the core logic of the DKM base class. It uses a record to store the group name and the last unprotect policy (instead of C# properties). It also turns the (implicit) authenticated user identity into an explicit parameter, so that it can be used in specification events. We give an outline of the whole F# implementation, showing code only for *Unprotect*. As explained above, part (1), above the **#if** F7 conditional-compilation flag, is formally verified by typing; and part (2) is a stub implementing *IDKM* by calling part (1).

```
module DKM
(...)
type Dkm = { GroupName: string;
             DecodedPolicy : KeyPolicy Var.t }
let DkmUnprotect (user:principal) (dkm:Dkm) cipher plain =
  let n = dkm.GroupName
  let ae = DkmDecodeProtectedBlob cipher
  let p = Var.get ae.Policy
  let keyId = KeyPolicyGetCurrentKeyGuid p
  let k0 = ReadKey user n keyId
  let k = checkKeyLength p k0
  AuthEncDeriveKeys ae k
  Var.set dkm.DecodedPolicy p
  AuthenticatedDecrypt ae cipher plain

#if F7
#else
type public DKMFsharp (repository:IRepository,
                       keyCache:IKeyCache,
                       authEncrypt:IAuthEncrypt) =
  let user = WindowsIdentity.GetCurrent().User.ToString()
  let dkm = CreateDkm user repository.GroupName
  (...)
  interface IDKM with
    member r.GroupName with get() = dkm.GroupName
    member r.Unprotect(cipher:MemoryStream) =
      let plainText = new MemoryStream()
      DkmUnprotect user dkm cipherText plainText
      plainText
  (...)
#endif
```

As explained in Section III, *DkmUnprotect* first parses the blob and decodes the unprotect policy (p). This may fail, for instance if the policy proposes algorithms outside the set supported by DKM. Otherwise, this returns an initialized instance (ae) of *IAuthEncrypt*. The code then reads the key ($k0$) referenced in the policy from the repository, checks that this key is appropriate for use with p, and finally performs authenticated decryption on ae (listed below). The stub for *IAuthEncrypt* implements the method *IDKM.Unprotect*; it creates the plaintext stream then just calls *DkmUnprotect*.

Authenticated Encryption. Module *AuthEncModel* implements crypto-agile protection depending on a given policy p and its fields: *p.AuthEncMethod*, *p.KdfMacAlgorithm*, *p.MacAlgorithm*, and *p.EncryptionAlgorithm*. We give the code of the two main functions for encryption and decryption, as described in Figure 3. These functions operate on byte arrays and are parameterized by policy (p), random nonce (r), encryption IV (iv), and group key (k); they each have three branches, depending on the authentication method. Decryption is also parameterized by the expected lengths for ciphertext and mac, previously obtained when parsing the ciphertext. This code is verified by typing, as explained in Section A.

```
let authenticatedEncrypt p r iv k plaintext =
  let ke,kh = deriveKeys p r k
  match AuthEncMethod p with
  | AuthEncryptCM →
      encrypt p.EncryptionAlgorithm iv ke plaintext
  | EncryptThenMac →
      let e = encrypt p.EncryptionAlgorithm iv ke plaintext
      let v = concat iv e
      assert(IsEncryptThenMac(v,k,p,r,plaintext,ke,iv,e))
      let m = mac p.MacAlgorithm kh v
```

```
    concat e m
  | MacThenEncrypt →
    let m = mac p.MacAlgorithm kh plaintext // but not iv
    let v = concat plaintext m
    assert(IsMacThenEncrypt(v,k,p,r,plaintext,kh,m))
    encrypt p.EncryptionAlgorithm iv ke v

let authenticatedDecrypt p r iv k ciphertext ctLn macLn =
  let ke,kh = deriveKeys p r k
  match AuthEncMethod p with
  | AuthEncryptCM →
    let pt = decrypt p.EncryptionAlgorithm iv ke
         ciphertext
    pt
  | MacThenEncrypt →
    let d = decrypt p.EncryptionAlgorithm iv ke
         ciphertext
    let pt,ptm = split d (ctLn−macLn) macLn
    macVerify p.MacAlgorithm kh pt ptm
    pt
  | EncryptThenMac →
    let ct,ctm = split ciphertext (ctLn−macLn) macLn
    let v = concat iv ct
    macVerify p.MacAlgorithm kh v ctm
    let pt = decrypt p.EncryptionAlgorithm iv ke ct
    pt
```

In-Memory Repository. Module *RepositoryModel* provides a simple implementation of the *IRepository* interface. We represent the persisted state of each DKM group as a tuple consisting of the group name, a mutable list of key policies, an in-memory database of keys indexed by key identifiers (Guid), and an in-memory database of access rights indexed by user names. Their F# type is

```
type StoredGroup =
  groupname ∗
  KeyPolicy list ref ∗
  Db.t<guid,Key> ∗
  Db.t<principal, right list>
```

We list sample code for setting and getting key policies. Similarly to the *Protect* and *Unprotect* functions, the Get and Set policy functions accept a user name string to enforce access rights.

```
let consPolicy n p ps = p::ps
let insertPolicy n p ps = ps := consPolicy n p !ps

let GetKeyPolicy (id:principal) (n:groupname) =
  let (_,ps,_,_) = Access id n Read
  match !ps with
  | p::_ → assert(GroupPolicy(n,p)); p
  | _ →failwith "no policy yet"

let SetKeyPolicy (id:principal) (n:groupname) (p:KeyPolicy)
     =
  let (_,ps,_,_) = Access id n Write
  assume (Assigned(id,n,p)) // security event
  insertPolicy n p ps
```

Additional Modules. For completeness, we mention the other F# modules of the reference implementation:

- *Acls* implements an abstraction of access control list to represent access rights;
- *Db* implements an abstraction of an in-memory database;
- *Var* implements a write-once variable (see Section A);
- *KeyPolicyModel* implements key policies as records, rather than objects.
- *CryptoModel* provides functional wrappers around .NET cryptographic primitives, and is the basis of their formal model.

Testing. We validate our F# code by writing tests, both in F# and in C#. These tests cover a number of DKM scenarios as well as interoperability between the two implementations. In addition to C# manual code reviews, this gives us high confidence that our verified code closely corresponds to our production code, and also enables us to automatically detect any discrepancy that may appear as the result of code changes. (So far 9 minor revisions of the C# DLL have been released.)

The C# tests make sure that the F# code interoperates with the C# code. To this end, we wrote a simple in-memory repository also in C#, and ported the 28 existing C# tests. Those tests exercise scenarios such as initializing *IDKM* and *IRepository* implementations, adding groups to a repository, adding users to these groups, and protecting and unprotecting various series of data.

The F# tests take the existing 28 tests for the C# in-memory repository, and replace instances of C# DKM components with their F# counterparts. Additionally, we also test using the C# *IDKM* implementation configured to use the F# in memory repository, to make sure that this repository operates correctly.

Finally, we write explicit interoperability tests, making sure that data protected with F# can be unprotected with C# and vice-versa, for various choices of policies and algorithms. There are 56 such tests written in C# exercising F# DKM.

Hybrid Codebase. Figure 5 provides lines of code in C#, F#, and F7 for the verified codebase, following the structure of the F# code, as well as its verification time. Both for C# and F#, we give separate sizes for interfaces and their implementations. Overall, DKM consists of 21,000 lines of code; it contains cryptographic primitives and data structures, and several repository implementations. Thus, the programming overhead for building the reference F# implementation is small as compared to the whole DKM project. For F7, we give the size of the corresponding annotated interfaces we use for automated verification, and also the size of the query script passed to the SMT solver. Hence, the logical annotations (855 lines in total) are almost

```
// privileged administrator functions
let admin: principal = "DKM administrator"
let AddServer group s = AddUser admin group s Acls.Read
let CreateServerGroup (servers:principal list) group =
  let dkm = CreateDkm admin group
  DkmAddGroup admin dkm
  List.iter (fun s → AddServer group s) servers

// server functions
let Read id group (u:username) (x:provider) =
  let dkm = CreateDkm id group
  let cipher = FileToStream (u,x)
  let plain = newMemoryStream ()
  DkmUnprotect id dkm cipher (write_only plain)
  let (u',x',pwd) = StreamToData (read_only plain)
  if (u',x') = (u,x) then pwd
  else failwith "stored record mismatch"

let Store id group (u:username) (x:provider) pwd =
  let dkm = CreateDkm id group
  let plain = DataToStream id group (u,x,pwd)
  let cipher = newMemoryStream()
  DkmProtect id dkm plain (write_only cipher)
  StreamToFile (u,x) (read_only cipher)
```

Fig. 6. F# Sample Code for Stored Passwords

as large as the verified F# code, partly because we systematically re-used C# code that did not require any verification rather than re-implement it in F#. Most of the verification effort focused on developing these annotations. Still, once in place, automated verification is fast and modular: the total runtime for F7 is smaller than the time it takes to build or test DKM.

Sample User Code. We complete our outline of our implementations with sample F# code for the two scenarios of Section II. For simplicity, our code directly calls our verified functions rather than the methods defined in the main DKM interface.

The code given in Figure 6 first defines functions for the administrator to create and populate groups. We may use this code to write a test with six servers split in two groups; for instance the commands for creating the groups are:

```
CreateServerGroup
  ["srv1";"srv2";"srv3";"srv4" ]
  "Stored E-Mail Credential Group"
CreateServerGroup
  ["srv5";"srv6" ]
  "Third-Party Access Group"
```

The code then defines two functions for data center e-mail servers granted access to stored credentials, to read and store the protected user credentials on untrusted storage. The first parameter *id* represents the authenticated caller id. The auxiliary functions *FileToStream* and *StreamToFile* associate the username with some untrusted storage location and perform actual I/O.

VII. Evaluation

A. Problems we found and fixed in DKM

Our modeling effort uncovered subtle problems in preliminary versions of DKM, and more generally led us to a better understanding of the properties provided by DKM and some restructuring of its codebase. For modularity, we added abstract interfaces, so that we could easily pick and mix between C# and F# implementations.

In particular, modeling lead us to provide access to the 'unprotect policy' effectively used to unprotect a blob, by adding a read-only property to *IDKM*, so that a suspicious user can inspect the policy and its algorithms before processing the unprotected data. Technically, this enables us to verify a more precise authentication property.

We also wrote new unit tests to check corner cases for some security properties and prevent regression after fixing some vulnerabilities. In the paragraphs below, we enumerate the identified problems and verified resolutions that address these vulnerabilities.

Group attribution. In a preliminary version of DKM, although the key and decoded policy were cryptographically authenticated in *Unprotect*, the unprotected data could be attributed to the wrong group, potentially leading to confusion and misuse of data. The "attack" went as follows:

1) assume we have two groups, A and B, used to store different sorts of data (e.g. credentials for different services);
2) an adversary with access to the untrusted storage reads a blob for group A and writes it at the location of a blob for group B;
3) a user who is a member of both groups A and B reads that blob and attempts to unprotect it as data belonging to group B.

Although the user can derive the correct keys to unprotect that blob (as data belonging to group A), this particular attempt should fail—otherwise, the user would treat A's credentials as if they were B's credentials (e.g. pass them to a compromised service).

To prevent the attack, DKM keys must be fetched from a container that includes only the keys of the target group of the unprotect, here the keys for group B. This property was broken in case the DKM keys were retrieved from the local key cache, which was indexed only by key identifiers: if the user had recently accessed group A, then the DKM key for A would sit in its cache, so the unprotect with target group B above would fetch it and silently succeed. We fixed this problem by indexing the cache by group name and key identifier, and also added specific tests for this attack.

Module	C# itf (.cs)	C# impl (.cs)	F# itf (.fsi)	F# impl (.fs)	F7 itf (.fs7)	Verification queries (.smp)	Verification time (S)
Var	–	–	10	38	13	185	2.10
Db	–	–	10	23	16	0	1.94
CryptoModel	–	–	100	328	322	–	–
Acls	–	–	–	51	26	–	–
KeyPolicy	–	390	24	120	44	0	2.25
AuthEnc	169	523	36	244	143	1112	4.07
Repository	206	508	–	368	172	865	2.18
DKM	367	1125	–	275	119	880	3.88
total for *DKM*	742	2546	180	1447	855	3042	18.48
SampleServer	–	–	53	330	39	16	3.23
Tests	–	7192	–	197	–	–	–

Fig. 5. Lines of code in C#, F#, and F7, and verification time. *itf* is interface, *impl* is implementation.

Although this kind of logical attack is trivial to fix once identified, it is unlikely to be found by undirected testing. On the other hand, models written for traditional formal analyses may miss implementation details such as local key caches, and thus also miss this attack. Our verification method involves systematically building (and checking) a correspondence between the production code and the model. Thus, it is well adapted to detect this kind of logical attack.

Implicit Authentication of the Authentication Method. Our verification effort discovered a construction problem in the input of the key derivation function. When using CBC encryption and MAC, the key derivation algorithm *KDF* did not differentiate between the encrypt-then-MAC and MAC-then-encrypt authentication methods, violating the unambiguous encoding promise stated in Section III. In other words, the algorithms but not the method were included as parameters for key derivation, allowing the adversary to change this method and cause the same keys to be shared between different methods using the same algorithms. For example, an adversary would have been able to inject blobs (apparently) protected with MAC-then-encrypt and, assuming it could break this method, recover their keys and then also unprotect blobs protected with encrypt-then-MAC. In contrast, our formal model requires resistance against chosen-ciphertext attacks only when the method is MAC-then-encrypt, which is fine as long as DKM uses separate keys for blobs (apparently) encrypted with distinct methods. We caught this vulnerability using our new model for cryptographic agility.

To remove the vulnerability, we redefined the input of *KDF* to include the method M_{es} as well as its algorithms I_{es} (see Equation (3) in Section III). We let M_{es} be 1 for GCM (encoded in one byte), 2 for MAC-then-encrypt (encoded in one byte), and the empty bytestring for Encrypt-then-MAC. One might ask why we use this ad hoc encoding instead of 3. There is a very practical issue at work: backward compatibility with deployed software. The default policy uses Encrypt-then-MAC. So, we inquired with the early users of DKM to make sure that none of them changed the default M_{es} setting in their DKM configuration. Thus, an empty, but still unambiguous encoding lets us be backward compatible with all protected blobs in deployed environments, while protecting new deployments with a modified mechanism. In practice, we do not foresee a default mechanism anything other than Encrypt-then-MAC or AES-GCM.

Setting Key Lengths & Policy updates. We also identified several inconsistencies in the choice of appropriate key lengths for the various algorithms, to ensure that each key has enough entropy for all algorithms it is used for. To compute the length of the DKM key k defined in Section III, we initially included the encryption E and MAC algorithm H key lengths but not the PRF $|H_d|$ key length in the key derivation function. Also, we did not systematically re-compute and re-check that key length when using the DKM key with a decoded policy, so that a key allocated for a given policy could be used (in principle) for another policy, even if the later policy would mandate a longer key. Formally, these discrepancies yield typechecking errors on the refinements of the key values. We fixed these issues before DKM was integrated into a product.

Cryptographic Policy in Unprotect. The DKM unprotect operation does not consult the policy in the repository. Instead, DKM parses out the effective unprotect policy from an protected blob, giving the adversary various knobs to play with. Assuming that parsing is done properly, we discovered that not all key lengths were verified against the decoded policy. In particular, the length of the DKM key k was not checked against the authenticated encryption and key derivation algorithms despite the checks to make sure that the derived key were of proper lengths.

The fix was straightforward: every DKM unprotect operation checked that the DKM key k was at least

as long as the key length L_k computed from the key derivation and authenticated encryption algorithms in (1).

B. Limitations of F7

Our case study also reveals limitations of the verification tools we used. Although F7 can handle complex logical models at the level of details of production code, it lacks support for many imperative and object-oriented programming patterns. In principle, our model could be checked directly against the C# codebase, but this would involve engineering beyond the scope of our case study. Our verification effort also stress the need for better error messages for logical debugging and for a local refinement-type inference while developing the model.

In terms of cryptographic verification, our method relies on symbolic assumptions on the underlying primitives. In future work, it would be revealing to formally validate these assumptions against more concrete cryptographic hypotheses, such as those for agility [1].

C. Conclusions

In modern distributed systems persisted data requires flexible key management and cryptographically agile policies to achieve long-term security and availability. This need is not only theoretical; it is identified in numerous actual software products and forms a basis for our effort. As a response, we design, implement, and verify DKM, a new security component that solves this problem and keeps cryptographic complexity behind a simple, high-level API for services that require data protection.

As part of the development cycle of this new security component we build, verify, and maintain a reference implementation. This yields security guarantees beyond those obtainable by independent verification efforts on simplified formal models. In our experience with DKM, this approach significantly improved our design and understanding, and helped us identify and fix several vulnerabilities early in its development cycle, avoiding costly post-deployment updates. Taking advantage of recent advances in automated tools for cryptography, this was achieved at a reasonable cost: a 15% overhead in code size including new verification libraries.

References

[1] T. Acar, M. Belenkiy, M. Bellare, and D. Cash. Cryptographic agility and its relation to circular encryption. In *EUROCRYPT 2010*, LNCS, pages 403–422. Springer, Berlin, Germany, May 2010.

[2] T. Acar, M. Belenkiy, L. Nguyen, and C. Ellison. Key management in distributed systems. Technical Report MSR–TR–2010–78, Microsoft Research, June 2010.

[3] M. Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In *CRYPTO 2006*, volume 4117 of *LNCS*, pages 602–619. Springer, Berlin, Germany, Aug. 20–24, 2006.

[4] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 1–15, Santa Barbara, CA, USA, Aug. 18–22, 1996. Springer, Berlin, Germany.

[5] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. Technical Report MSR–TR–2008–118, Microsoft Research, 2008. A preliminary, abridged version appears in the proceedings of CSF'08.

[6] K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. In *ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 198–209, 2004. An extended version appears as Technical Report MSR–TR–2003–83.

[7] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Cryptographically verified implementations for TLS. In *ACM Conference on Computer and Communications Security*, pages 459–468, Alexandria, VA, USA, 2008. ACM Press.

[8] K. Bhargavan, C. Fournet, and A. D. Gordon. F7: refinement types for F#, Sept. 2008. Available from http://research.microsoft.com/F7/.

[9] K. Bhargavan, C. Fournet, A. D. Gordon, and N. Swamy. Verified implementations of the Information Card federated identity-management protocol. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 123–135, Tokyo, Japan, 2008. ACM Press.

[10] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *ACM Symposium on Principles of Programming Languages (POPL'10)*, Jan. 2010.

[11] S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *IEEE Computer Security Foundations Symposium*, pages 172–185, Port Jefferson, NY, USA, 2009. IEEE Computer Society.

[12] L. Chen. *NIST Special Publication 800-108: Recommendation for Key Derivation Using Pseudorandom Functions*. Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-8900, Oct. 2009.

[13] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[14] A. Dey and S. Weis. *KeyCzar: A Cryptographic Toolkit*. Google, Inc., Aug. 2008. http://keyczar.googlecode.com/files/keyczar05b.pdf.

[15] T. Dierks and E. Rescorla. *RFC 5246, The Transport Layer Security (TLS) Protocol Version 1.2*. IETF, Aug. 2008. http://www.ietf.org/rfc/rfc5246.txt.

[16] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.

[17] W. Ehrsam, C. Meyer, R. Powers, J. Smith, and W. Tuchman. *Product Block Cipher System for Data Security*. International Business Machines, Feb. 1975.

[18] J. Goubault-Larrecq and F. Parrennes. Cryptographic

protocol analysis on real C code. In *VMCAI'05*, pages 363–379, 2005.

[19] D. Harkins and D. Carrel. *RFC 2409, The Internet Key Exchange (IKE)*. IETF, Nov. 1998. http://www.ietf.org/rfc/rfc2409.txt.

[20] *RFC 4306, The Internet Key Exchange (IKEv2)*. IETF, Dec. 2005. http://www.ietf.org/rfc/rfc4306.txt.

[21] J.Rizzo and T.Duong. Practical padding oracle attacks. In *Black Hat Europe*, Barcelona, Spain, Apr. 2010.

[22] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648, Santa Barbara, CA, USA, Aug. 15–19, 2010. Springer, Berlin, Germany.

[23] D. A. McGrew and J. Viega. The security and performance of the Galois/counter mode (GCM) of operation. In A. Canteaut and K. Viswanathan, editors, *INDOCRYPT 2004*, volume 3348 of *LNCS*, pages 343–355, Chennai, India, Dec. 20–22, 2004. Springer, Berlin, Germany.

[24] *Cryptography*. Microsoft, Mar. 2010. http://msdn.microsoft.com/en-us/library/aa380255(VS.85).aspx.

[25] *Cryptography API: Next Generation*. Microsoft, Mar. 2010. http://msdn.microsoft.com/en-us/library/aa376210(VS.85).aspx.

[26] NAI Labs, Network Associates, Inc. *Windows Data Protection*. Microsoft, Oct. 2001. http://msdn.microsoft.com/en-us/library/ms995355.aspx.

[27] *PKCS#11 v2.30: Cryptographic Token Interface Standard*. RSA Laboratories, Apr. 2009. http://www.rsa.com/rsalabs/node.asp?id=2133.

[28] K. Scarfone, M. Souppaya, and M. Sexton. *NIST Special Publication 800-111: Guide to Storage Encryption Technologies for End User Devices*. Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-8900, Nov. 2007.

[29] T.Ylonen and C.Lonvick. *RFC 4251, The Secure Shell (SSH) Protocol Architecture*. IETF, Jan. 2006. http://www.ietf.org/rfc/rfc4251.txt.

[30] S. Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In L. R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 534–546, Amsterdam, The Netherlands, Apr. 28 – May 2, 2002. Springer, Berlin, Germany.

[31] X. Wang and H. Yu. How to break MD5 and other hash functions. In R. Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 19–35, Aarhus, Denmark, May 22–26, 2005. Springer, Berlin, Germany.

[32] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 17–36, Santa Barbara, CA, USA, Aug. 14–18, 2005. Springer, Berlin, Germany.

Appendix

We first review our method for verifying the security of code written in F#, relying on logical models typechecked by F7. Section A explains how we apply this method to DKM.

Symbolic Cryptography. Formal verification relies on a series of security assumptions, expressed as hypotheses on the interface of the cryptographic-algorithm libraries. In the absence of formal security proofs on the concrete design and implementation of the core primitives, we rely instead on dual, non-standard implementation of these libraries, which embody the target security properties in some more abstract programming model, as initially proposed by Dolev and Yao [16]. Hence, we obtain theorems about the security of protocol implementations linked against symbolic cryptographic libraries, against all adversaries with access to selected protocol and library interfaces, and we can independently review whether those symbolic libraries adequately reflect the strengths and known weaknesses of the underlying cryptography. We refer to Bengtson et al. [5] and Bhargavan et al. [10] for a detailed description and security analysis of two large cryptographic libraries using F# and F7. Section A also illustrates our approach on the interface for keyed cryptographic hashes.

Refinement Typechecking. Our main verification tool is F7, a refinement type checker developed for the purpose of verifying protocol implementations [5]. F7 supplements F# with much-richer types carrying first-order-logic formulas that can specify properties of the values that have those types. These formulas are verified during type-checking, by calling an automated theorem prover. (In our case, we rely on Z3, an SMT solver.) After typechecking, these formulas are erased, and we are left with ordinary F# programs with plain .NET types.

Sample Refinements. For example, let 'bytes' be the type of byte arrays (byte[] in C#) and suppose we use bytes to represent cryptographic key values. For verification purposes, instead of *bytes*, we may use a more precise *refined type* for keys, declared in F7 as

type key = *b*:bytes { *Length*(*b*) = 32 ∧ *SymmetricKey*(*b*) }

This type is inhabited by values b of type 'bytes', its base type at runtime, that also meet the *logical refinement* specified by the formula between braces. Hence, keys are bytes such that two facts hold: (1) the byte array has length 32, and (2) the predicate '*SymmetricKey(b)*' holds, which, in our model, may indicate that the key was correctly generated by a particular algorithm.

Suppose our encryption function takes a formal key parameter with type 'key', rather than just 'bytes'. Then, typechecking an encryption involves proving that these two facts hold for the actual key passed as an argument. (Refinement formulas on arguments act as logical preconditions.) Moreover, if the key generation function is the only function with a result type refined with the fact '*SymmetricKey(b)*', then typechecking prevents

any other bytes from being treated as keys, and thus well-typed programs use only correctly-generated keys for encryption. (Refinement formulas on results act as logical post-conditions.)

Programs typechecked with F7 can use two special specification statements:

- The statement '**assert** C' asks the typechecker to verify that formula C holds in the current context, that is, can be deduced from the current typing environment. Otherwise, typechecking fails.
- Conversely, '**assume** C' tells the typechecker that formula C holds. For instance, we may use a statement '**assume***SymmetricKey(b)*' within the implementation of our key-generation function, to promote 32 freshly-generated random bytes and give its result the type 'key'.

These statements have no effect at runtime (since all formulas are erased after typechecking) but they must be carefully reviewed to understand the logical properties established by typechecking. The main type-safety theorem for F7 states that, whenever a well-typed program executes an **assert** C, the formula C logically follows from the conjunction of all previously-assumed formulas.

We now outline our verified model, highlighting some of its logical properties. The main novelties are (1) the handling of imperative programming features; (2) the first symbolic model for cryptographic agility—all crypto functions are parameterized by their algorithm and the protocol retains security properties even when some of those algorithms are broken; and (3) a precise account of access-control and group management.

We describe verification in three categories: program verification, cryptographic verification, and API verification.

A. Program Verification

Untrusted functions. Many functions implemented in C# are irrelevant for the security of DKM. Technically, we verify this by declaring their types in F7 without any refinement; typechecking code that uses them under those plain types (that is, in particular, without making any assumption on the values returned by those functions); and implementing them in F# using calls to C#. This approach enables us to safely limit our modeling effort. For instance, the format for encoding policies into protected blobs is irrelevant for security: their model only ensures that encoding and decoding functions operate on public bytes.

Mutable Objects. Our executable model is more concrete and imperative than those previously verified with F7, so we extend its verification libraries accordingly.

For instance, a new F7 module '*Var*' provides a simple abstraction to keep track of the values of variables and fields that are mutable (as usual in C#) but should be written at most once during the lifetime of the object (as often in DKM code, for instance for each of the private fields of *AuthEncrypt* for keys, IV, etc, which are gradually filled as the blob is processed). A variable r of type *Var.t* can be written at most once and read any number of times; both operations are typed in F7 with a logical postcondition *Val(r,v)* stating that v is the current value stored in r. Writing a second time to a variable triggers a run-time error (detected by testing). As a benefit, '*Var*' can includes an F7 assumption stating that this type of variable stores at most one value (logically, $!r,v,v'.\ Val(r,v) \wedge Val(r,v') \Rightarrow v = v'$ where ! is universal quantification); this assumption is used during typechecking and enables efficient verification of the code that depends on such variables, essentially as if they were immutable.

Stream-based I/O. As shown in Section V, our C# code systematically relies on I/O streams for incrementally reading and writing both (authentic, confidential) source bytes to be protected and (raw, public) blob bytes after protection.

For verification, we need to distinguish between readers and writers, and to precisely keep track of logical refinements on the stream contents. To this end, we decompose the single '*System.IO.MemoryStream*' type into several, more precise types in F7, and we introduce functions for casting streams into input-only streams and output-only streams:

val *newMemoryStream*: unit $\rightarrow \alpha$ *Stream*
val *read_only*: α *Stream* $\rightarrow \alpha$ *InStream*
val *write_only*: α *Stream* $\rightarrow \alpha$ *OutStream*

(In F#, all these types are plain memory streams, and we define **let** *read_only s = s* and **let** *write_only s = s* .) Input streams can be read, but not written, so they are contravariant; conversely output streams are covariant. Hence, for example, in a typing context such that we have a logical implication $C \Rightarrow C'$, a stream of type $(x:\text{bytes}\{C\})$*InStream* can safely be passed as a stream of type $(x:\text{bytes}\{C'\})$ *InStream* (since the latter refinement is less demanding on the values read off the stream). This kind of subtyping on polarized streams is used for instance to give different, precise types to the content of the plaintext stream, as it is passed from *DKM* to *AuthEncrypt*.

B. Cryptographic Verification

Core Cryptographic Model. We refer to Bhargavan et al. [10] for a general description of modeling cryptographic libraries as refined F7 modules. We develop our own libraries to account for cryptographic agility: all

primitives take an extra 'algorithm identifier' parameter, and all symbolic security properties are conditioned by special predicates that specify our cryptographic assumptions on these algorithms. (This is illustrated below for keyed hash functions.)

Our formal model includes a few sample algorithms for each kind of cryptographic primitive. In contrast with production code, it also deliberately includes one "broken" algorithm of each kind to accurately reflect potential attacks when using poor policies. Formally, we do not assume anything about those algorithms. Thus, when those algorithms are run, we can pessimistically give full control of their behavior to the adversary: the adversary receives all their arguments (including the key!) and sends back any bytes to be used as the result of the algorithm. This enable us to verify, for instance, that plaintexts protected with strong algorithms remain secure, even if some other plaintexts within the same group are (poorly) protected using broken algorithms.

```
type oid =
| Aes128CBC (* sample encryption algorithm *)
| Aes128GCM (* sample authenticated−encryption
    algorithm *)
| WeakEncryption (* sample bad encryption algorithm *)
| Sha256 (* sample hash algorithm *)
| WeakHash (* sample bad hash algorithm *)
| NoAlgo (* no algorithm (null in C#) *)

type security_assumptions =
| PRF of oid (* ok for key derivation *)
| CPA of oid (* ok for encrypting authenticated data *)
| CCA of oid (* ok for encrypting any data *)
| CMA of oid (* ok for authenticating data (MACs) *)
| CCM of oid (* ok for authenticated encryption *)
```

We also use different refinements on the raw "bytes" datatype to check the consistency of the algorithms used, for instance, for generating keys and IVs versus those used later for encryption.

Sample primitive: authentication. We illustrate our approach for authentication primitives, which have the following plain types in F#:

```
val mac: oid → Key → bytes → bytes
val macVerify: oid → Key → bytes → bytes → unit
```

Both functions take as parameters an algorithm identifier *a*, a key *k*, and the bytes value *v* to be authenticated; the *mac* function returns a MAC *m*, whereas *macVerify* takes a MAC as fourth argument, and either returns nothing (if verification succeeds) or raises an exception (if verification fails). In F7, these two functions are given the more precise refinement types below:

```
val mac: a:oid → k:Key → v:bytes
  { ((IsMACKey(k) ∧ MACSays(k,v))
      ∨ (Pub(k.value) ∧ Pub(v)) ) ∧ k.oid = a } →
  m:bytes { (Pub(v) ⇒ Pub(m)) ∧ IsMAC(m,k,v) }

val macVerify: a:oid → k:Key
  { (IsMACKey(k) ∨ Pub(k.value)) ∧ k.oid = a }
  → v:bytes → m:bytes → unit { IsMAC(m,k,v) }

assume !m,k,v.
  IsMAC(m,k,v) ∧ CMA(k.oid) ⇒ MACSays(k,v) ∨ Pub(k.
      value)
```

In these refinement types,

- the preconditions for both functions demand that *k* be a MAC key, using the fact *IsMACKey(k)* whose algorithm identifier matches the one passed as parameter (using the equality $k.oid = a$). Thus, for instance, any erroneous code that may potentially call *macVerify* with a key derived for another algorithm would trigger a type error during its verification.
- The predicate *MACSays* logically specifies the meaning of an "authentic" value for a given key. It is a precondition of *mac*, and a (conditional) postcondition of *macVerify*.
- The predicate *Pub* tracks the knowledge of the adversary, that is, public and potentially tainted values. For instance, $Pub(v) \Rightarrow Pub(m)$ in the *mac* postcondition says that the MAC of m is public, and can thus be written to untrusted storage, provided that the MACed value v is also public; otherwise the MAC may leak information on v.
- The fact *CMA(k.oid)* records a security assumption stating that the algorithm with identifier *k.oid* protects integrity against chosen-message attacks, that is, even if the adversary can influence the values being MACed. This fact is *not* a precondition for calling our two functions, but is required to deduce *MACSays* after calling *macVerify*.
- The fact *Pub*(*k.value*) models the potential use of compromised keys. Its use here reflects the possibility that the group key has been leaked, or provided by the adversary.
- The predicate *IsMAC* is the postcondition of a successful MAC verification; otherwise *macVerify* throws a security exception. The assumption above gives it logical meaning.

Thus, following the logical definition of *IsMAC*, after cryptographic verification, we only know that, if the algorithm is CMA, then *either* the verified value *v* is authentic *or* the adversary knows the authentication key. This logical property will need to be combined with properties of the key derivation to rule out the second alternative.

Sample primitive: key derivation. We give below the type of our key-derivation function, and the definition of its first postcondition *IsDerivedMACKey*. We omit the definition of its counterpart for encryption *IsDerivedEncryptionKey*.

```
val deriveKeys:
```

```
p:KeyPolicy → n:bytes →
s:Key { s.oid = p.KdfMacAlgorithm } →
ke:Key ∗ ka:Key { IsDerivedEncryptionKey(ke,p,n,s)
                  ∧ IsDerivedMACKey(ka,p,n,s) }
```

```
assume !ka,p,n,s.
  IsDerivedMACKey(ka,p,n,s) ⇒
  ( ka.oid = p.MacAlgorithm
  ∧ ( PRF(p.KdfMacAlgorithm) ∧ CMA(p.MacAlgorithm)
        ∧ Pub(ka.value) ⇒ Pub(s.value))
  ∧ ( p.AuthEncMethod = MacThenEncrypt ⇒
       ( !x. MACSays(ka,x) ⇔ Protect(s,p,x) ))
  ∧ ( p.AuthEncMethod = EncryptThenMac ⇒
       ( !x. MACSays(ka,x) ⇔
          (∃plain,ke,iv,e. IsEncryptThenMac(x,s,p,n,plain,ke,iv
              ,e)))))
```

Intuitively, the assumed formula defines the specific logical properties of each kind of keys that may be derived from group key *s* according to policy *p*. For example, if we use the MAC-then-encrypt method, then the post-condition of *deriveKeys* tells us that the key *ka* matches the MAC algorithm in the policy, and that the logical meaning of authentication with *ka*, defined by the predicate *MACSays*, is that the authenticated value is a DKM plaintext protected using the group key *s* and policy *p*, as recorded by the fact *Protect(s,p,plain)*.

Authenticated Encryptions. The declared types of the two main functions for authenticated encryptions are:

```
val authenticatedEncrypt:
  p:KeyPolicy →
  r:bytes → iv:bytes { IsIV(iv,p.EncryptionAlgorithm) } →
  s:Key { s.oid = p.KdfMacAlgorithm } →
  text:bytes { Protect(s,p,text) } → blob:bytes
  { Pub(iv) ∧ ( Confidential(p) ∨ Pub(text) ⇒ Pub(blob) ) }
```

```
val authenticatedDecrypt:
  p:KeyPolicy →
  r:pubs → iv:pubs →
  k:Key { k.oid = p.KdfMacAlgorithm } →
  blob:pubs → int → int → text:bytes
  { Authentic(p) ⇒ (Protect(k,p,text) ∨ Pub(k.value)) }
```

Their typechecked implementation is given in Section VI; it has specialized code for each of the three possible methods prescribed by the policy *p*. Notice that the security postconditions depend on the cryptographic properties of the policy: for instance, to deduce that the decrypted text is authentic after calling *authenticatedDecrypt*, we need *Authentic(p)* to hold, a predicate defined from the security assumptions on the three security algorithms of *p*.

C. API Verification

Repository. Our implementation of the *IRepository* interface maintains mutable state for each group using an in-memory database with entries of the following refined type:

```
type StoredGroup =
  n: groupname ∗
  (p:KeyPolicy{GroupPolicy(n,p)}) list ref ∗
  (guid,k:Key{GroupKey(n,k)}) Db.t ∗
  (principal, right list) Db.t { ∃o. Created(o,n) }
```

Each entry includes a list of past and present policies; a list of all keys for the group; and an ACL that records the access level for each user. We maintain a list of the successive policies that have been stored for the group, rather than just the current policy, to account for weak synchronization and the possibility of protecting a plaintext with an out-of-date policy.

The refinement formulas in the type *StoredGroup* above are essential to structure our model: to typecheck code that stores a new policy, for instance, one has to prove that this policy is a valid policy for that group (this is tracked by the predicate *GroupPolicy*). Conversely, when typechecking code that fetches a policy from the repository, we know that this policy was valid at some point in the past.

We give below the logical definitions of the predicates that give a formal meaning to the statements "*x* had *a* access to group *n*" (with e.g *a* = *Read*, or *a* = *Write*) and "the policy *p* was valid at some point for group *n*". These predicates are recursively defined to keep track of delegation chains for group access (from runtime events *Created*, *Granted*, and *Corrupt*) and of the origin of any stored policy (from runtime events *Default* or *Assigned*), respectively.

```
assume !x,n,a. HadAccess(x,n,a) ⇔
  Created(x,n) ∨
  ∃z.( HadAccess(z,n,Owner) ∧ (Granted(z,n,a,x)∨ Corrupt(z
       )))
```

```
assume !n,p. GroupPolicy(n,p) ⇔
 ∃o. Created(o,n) ∧
 ( Default(p) ∨
 ( ∃z. HadAccess(z,n,Write) ∧ (Assigned(z,n,p)∨ Corrupt(z)))
     )
```

Main DKM Interface. Relying on logical definitions from all refined modules, we arrive at the following types for the two main functions of DKM:

```
val DkmProtect:
  id:principal → dkm:Dkm →
  plaintext:
  (t:bytes {CanProtect(dkm.GroupName, id, t)}) InStream
       →
  protectedData: pubs OutStream → unit
```

```
val DkmUnprotect:
  id:principal → dkm:Dkm →
  ciphertext: pubs InStream →
  plaintext:
  (t: bytes { ∃p. Var.Val(dkm.DecodedPolicy,p)
     ∧ Unprotected(id,dkm.GroupName,p,t) }) OutStream →
        unit
```

Instead of just returning a pair of a plaintext and a decoded policy, *Unprotect* stores the decoded policy in a variable and sends the plaintext bytes on an output stream. Accordingly, the effective postcondition of *Unprotect* is attached as a refinement of any bytes sent on that stream, with an existential quantifier on the content of the decoded-policy variable.

The most precise property verified as a postcondition of *Unprotect* is defined as follows:

```
assume !x,n,p,t. Unprotected(x,n,p,t) ⇔
  ∃k,ae. (
      Var.Val(ae.Policy,p) ∧ Var.Val(ae.Key,k)
    ∧ HadAccess(x,n,Read) ∧ GroupKey(n,k)
    ∧ APP(k,p,t) (* needed just as a logic hint *)
    ∧ ( Authentic(p) ⇒
          (GroupPolicy(n,p) ∧ ∃y. Protected(y,n,p,t))
        ∨ (∃z. HadAccess(z,n,Read) ∧ Corrupt(z))))
```

where the application-specific predicate *CanProtect* records data protected in the group.

Typechecking these properties involves delicate subtyping, as the conditions passed from one function to another evolve from properties on users and groups to properties on keys and algorithms. (In the definition, the *APP* predicate is included as a hint for Z3, our underlying prover; it is logically redundant with the rest of the formula but it helps Z3 instantiate variables in the proof search; its definition is omitted.)