Policy-based Secure Deletion

Christian Cachin* Kristiyan Haralambiev* Hsu-Chun Hsiao†
Alessandro Sorniotti*

August 26, 2013

Abstract

Securely deleting data from storage systems has become difficult today. Most storage space is provided as a virtual resource and traverses many layers between the user and the actual physical storage medium. Operations to properly erase data and wipe out all its traces are typically not foreseen, particularly not in networked and cloud-storage systems. This paper introduces a general cryptographic model for policy-based secure deletion of data in storage systems, whose security relies on the proper erasure of cryptographic keys. Deletion operations are expressed in terms of a policy that describes data destruction through deletion attributes and protection classes. The policy links attributes as specified in deletion operations to the protection class(es) that must be erased accordingly. A cryptographic construction is presented for deletion policies given by directed acyclic graphs; it is built in a modular way from exploiting that secure deletion schemes may be composed with each other. The model and the construction unify and generalize all previous encryption-based techniques for secure deletion. Finally, the paper describes a prototype implementation of a Linux filesystem with policy-based secure deletion.

1 Introduction

Modern storage systems do not include operations to reliably destroy stored information. Common deletion operations simply mark the occupied space as free and remove an entry from the directory, but some of the stored data may remain accessible for much longer. A technically knowledgeable user with low-level access to the storage system can still obtain the data. This description applies to simple magnetic storage devices like disks or tapes, but holds as well for networked storage services, such as storage controllers in a data center, file servers, or cloud storage. Storage systems nowadays contain many layers of virtualization and perform aggressive caching for increased performance. They leave around traces of stored data beyond the control of the users, because such data cannot be securely wiped out through the usual service interface.

However, users would like to control the deletion of their information because supposedly deleted data that reappears later may have undesirable consequences. Many companies have installed detailed polices for retaining data and for deleting expired data; also the Electronic Frontier Foundation recommends controlled data deletion as a means to maintaining user privacy [9]. The European Data Protection Directive mandates that personal data must be erased upon request of the data subject [11].

With the advent of cloud computing, many clients who outsource storage want to take control over the shredding of their data themselves. They would like to retain an element of control that lets them erase their outsourced data from the cloud, without relying on the cooperation of the storage service.

*IBM Research - Zurich, CH-8803 Rüschlikon, Switzerland. `(cca|kha|aso)@zurich.ibm.com`.
†Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, USA. `hchsiao@cmu.edu`.

At the same time, providers of storage services are also interested to offer guaranteed destruction as a feature to their customers. For instance, providers would like to erase data so that no trace of it reappears later, not even during a forensic investigation.

In this paper, we explore the use of *encryption* and *key management* for securely deleting data. When data is stored encrypted, only the corresponding key has to be destroyed for erasing the data. Deleting (remote) data becomes a problem of managing and deleting (local) keys. We introduce the concept of *policy-based secure deletion*, where the stored data is grouped into *protection classes*, and *attributes* control the selective erasure of data through a *policy*. The design relies on a master key stored in a controlled and erasable memory, so that operations to delete data manipulate the master key, e.g., by updating it or erasing parts of it. No bulk data is ever re-encrypted.

This paper makes the following contributions.

- We introduce the notion of a *secure deletion scheme* and provide first formal model and a security definition for encryption-based secure deletion. It uses a coercive adversary that obtains the master key at the time of attack. The notion is formulated in the secret-key setting, but it can be extended to a public-key model.
- We construct a secure deletion scheme from encryption and threshold secret sharing; it supports arbitrary policies that are modeled as a circuit with AND, OR, and threshold gates.
- We show how secure deletion schemes can be composed in a modular way. Our approach unifies and generalizes all existing constructions for cryptographic secure deletion.
- We present a prototype implementation of a Linux filesystem with policy-based secure deletion.

Our encryption-based deletion methods apply to all kinds of storage systems, regardless of their physical storage media, and they can be integrated into existing systems with minimal effort. In contrast, solutions based on overwriting at the physical level [16, 24] only work in close connection with the media properties. This work aims at erasing data from large networked storage systems and assumes a small, controlled, and erasable keystore. Thus, it leverages physical (local) deletion to achieve secure deletion on (remote) large-scale data stores.

1.1 Related work

Secure deletion. Many systems have been proposed which essentially overwrite the data in order to delete it [16, 24]. Some methods are very flexible and can be integrated with arbitrary filesystems [17], provided their source code is available. Recent work has addressed solid-state storage, which requires completely different approaches than magnetic disks for destroying data [29, 23]. However, all solutions using overwriting depend heavily on the properties of the underlying physical storage. With cloud computing and the virtualized storage models that are widely used today, physical control over data-storage locations is no longer feasible. Therefore we do not further consider secure deletion mechanisms based on overwriting or other physical properties.

Employing encryption for the explicit goal of erasing information goes back to the work of Boneh and Lipton [4]. Di Crescenzo et al. [8] introduce a tree construction for efficient secure deletion of arbitrary files among a group of files. The master key at the root of the tree is kept in erasable memory, and every key in the tree encrypts several keys below, until the keys at the leaves encrypt the files themselves.

Mitra and Winslett [20] describe a method for creating an inverted index of keywords found in stored data records. The method uses encryption and allows to selectively delete a data record and the corresponding keywords in the index by assuming the encryption keys can be destroyed.

Perlman's Ephemerizer [21] employs a temporal sequence of keys modeling different expiration times for encrypted data. The FADE system [28] uses public-key cryptography and introduces some simple policies with Boolean operators governing deletion. FADE comes closest to our approach among the existing work. The policies of FADE are restricted to one- or two-level Boolean expressions, though, and its policies are intertwined with an implementation from a particular public-key cryptosystem. In contrast, our work permits general policies using Boolean expressions with threshold operators, may use generic cryptosystems, including secret-key systems, and supplies security proofs for all constructions.

Peterson et al. [22] use all-or-nothing transforms (AONT) at the block level for secure deletion, in combination with overwriting. The idea is to store every block through an AONT and then to overwrite only a part of it, which will render the whole block inaccessible.

Vanish [13] is a practical system for publishing content online with an expiration date, e.g., providing secure deletion for user data published in social networks. It encrypts the content and splits the key using secret sharing. The shares are then maintained by a peer-to-peer distributed system that gradually forgets the stored items unless they are refreshed. This gives the user some control over the expiration of his content.

Key-assignment schemes. Our approach to secure deletion is related to monotone secret sharing schemes and to key-assignment schemes for hierarchical access control. The survey by Crampton et al. [6] presents a summary of the literature on key assignment. Key assignment considers a publisher and multiple users. The publisher distributes one key to each user; every user can later derive suitable keys that allow the user to access information according to a hierarchical policy. The constructions may use public storage as well [1, 7]. Many constructions and improvements are available in the literature [6, 1, 2, 5]; they may be applied to the policy formulation and to the implementation of secure deletion schemes described here.

Attribute-based encryption. Our work also relates to the attribute-based encryption (ABE) schemes developed in the last few years [25], especially to ciphertext-policy ABE (CP-ABE) schemes [3]. Similar to CP-ABE, our policy-based secure deletion construction maintains keys for different attributes and the deletion policy is linked to the protected files. Many existing CP-ABE constructions suffer from creating large ciphertexts, though this deficiency has been removed recently [15]. While conceptually related to secure deletion in the sense that data is encrypted in a policy-specific way, CP-ABE does not already yield secure deletion. CP-ABE maintains a master key from which it produces attribute-specific decryption keys. Without knowing the attributes relevant in future decryption operations, either all such possible decryption keys must be generated in advance so the master key can be deleted or it must be kept around for generating decryption keys. However, this either is not efficient or contradicts the goal of secure deletion under coercion, when the master key must be revealed.

1.2 Organization

Section 2 introduces our notion of policy-based secure deletion. Multiple implementations of secure deletion schemes and the composition operation appear in Section 3, and Section 4 discusses their efficiency and other properties. Finally, Section 5 presents a filesystem with policy-based secure deletion.

2 Model

This section defines policy-based secure deletion schemes using a deletion policy represented by a graph.

2.1 Selective secure deletion

The goal of a policy-based secure deletion scheme is to maintain, on a permanent storage medium, a collection of *files* and to selectively delete some of them. Each file consists of a bit string of arbitrary length and is protected under a *protection class* from a set $\mathcal{P}$, as specified by a *deletion policy*; the formal definition of a deletion policy will be presented in the next section. A protection class is a logical grouping of files governed by an identical deletion rule. The universe of protection classes is denoted by $\mathcal{P} = \{p_1, p_2, \dots\}$.

The scheme provides operations for *protecting* a file, for *accessing* a file, and for *securely deleting* files. Secure deletion schemes in our model represent specialized encryption schemes and provide cryptographic security. We model only a secret-key secure deletion scheme, where the same key serves for protection of files and access to files; our model applies also to public-key schemes that may be defined analogously.

At the beginning, files of all protection classes are protected under an initial *master key*. The master key is stored in a closely guarded *erasable memory*, which is kept secret from an adversary. The master key will be changed later as a result of deletion operations. In contrast, all other data produced by the scheme is called *ciphertext* and stored in immutable *non-erasable memory*, which is public and exposed to the adversary at all times.

Each protection class is defined by means of *attributes* from a set $\mathcal{A}$ of strings over a fixed alphabet. *Secure deletion* operates on a subset of attributes, by ensuring that protection classes subject to those attributes become inaccessible. When a secure deletion operation is executed, a new master key is computed and stored in the erasable memory; the master key stored there previously is erased. Secure deletion may also change the ciphertext, i.e., add new ciphertext to the non-erasable memory.

2.2 Policy graph

A *deletion policy graph G* suitable for a secure deletion scheme is given by a pair (V, E) with $V = \mathcal{A} \cup \mathcal{P}$ such that (V, E) is a *directed acyclic graph (DAG)*. It has two kinds of nodes, sources and interior nodes. Nodes with no incoming edges (indegree zero) are *sources* and correspond one-to-one to the attributes in $\mathcal{A}$. All other nodes are called *interior nodes*; each of them is associated with a *threshold parameter m*, which is a positive integer less than or equal to the indegree of the node. Every interior node is labeled by a distinct protection class in $\mathcal{P}$. A policy graph must contain at least one source and one interior node, hence, the minimum policy graph has two nodes and one edge from the source node to the interior node.

Every node and every edge of the graph is associated to a Boolean value. All outgoing edges from a node take the same value as the node. The source nodes are assigned a value through the secure deletion scheme. An interior node with threshold m and n incoming edges corresponds to a Boolean threshold gate with threshold m: the node is TRUE whenever at least m among the n incoming edges are TRUE. Notice that G has a natural interpretation as a Boolean circuit whose sources correspond to $\mathcal{A}$. Threshold nodes subsume AND and OR gates as special cases.

2.3 Operation

Intuitively, deletion operations can be triggered by setting a subset of attributes to TRUE. The corresponding source nodes in G are then set to TRUE, which may cause some protection classes of G to become TRUE. This means that all files protected under these classes are deleted. Multiple secure deletion steps may follow each other.

More precisely, secure deletion works as follows. All protection classes are initially accessible, i.e., all source nodes and all their outgoing edges are FALSE at the start. As the circuit is monotone,

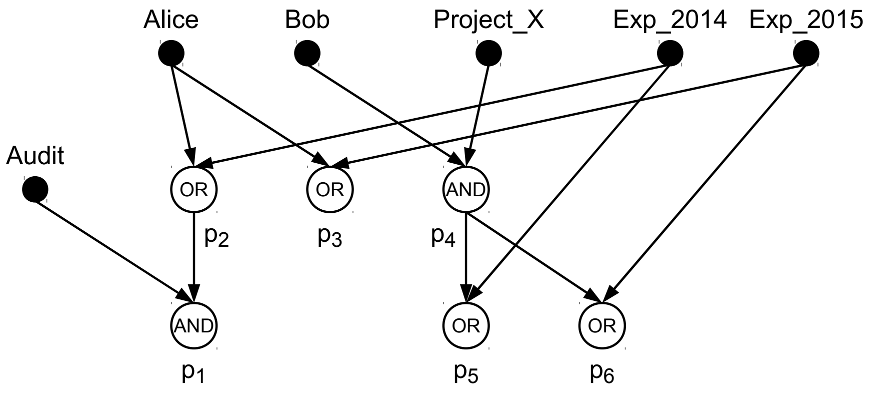

Figure 1. A sample policy graph.

this means that also all other nodes initially evaluate to FALSE according to G. Hence, none of the protection classes specified has been deleted and all files are accessible in the sense that their plaintext can be obtained from the ciphertext with the master key. A secure deletion operation takes a subset of the attributes as parameter and changes the associated source nodes, and, consequently, some edges in G to TRUE. Those files governed by protection classes that change their value to TRUE are no longer accessible after the master key has been updated. This model allows secure deletion of many files according to the policy and the given attributes. The attributes to delete can be specified independently of each other in arbitrary order.

Example 1. The policy graph of Figure 1 has six attributes (resp., source nodes) named *Alice*, *Bob*, *Project_X*, *Exp_2014*, *Exp_2015*, and *Audit* and six protection classes (resp., interior nodes) named $p_1, \ldots, p_6$. The latter are threshold nodes that implement binary AND and OR gates. The attributes model users that own files with different protection needs, a project category, expiration dates for stored data, and a special audit need for deleting stored data.

For example, protection class p_3 is governed by a policy *Alice* OR *Exp_2015*; thus data in p_3 becomes inaccessible as soon as a delete operation for protection classes owned by *Alice* is executed or a delete operation for the files with expiration *Exp_2015* takes place.

Protection class p_1 has policy $($*Alice* OR *Exp_2014*$)$ AND *Audit*. Data protected under this class only disappears after the attribute *Alice* or the attribute *Exp_2014* has been deleted, and, furthermore, after a secure deletion operation for *Audit* has been executed. This might apply when data protected under p_1 is more important for retention than data under p_2, as data in class p_1 can be destroyed only after an auditor has given consent to its erasure.

User *Bob* owns the data protected under p_4, p_5, and p_6. The policy dictates that classes p_5 and p_6 become inaccessible after securely deleting the attributes *Exp_2014* or *Exp_2015*, respectively, or when data owned by *Bob* and data labeled by *Project_X* is securely deleted. For instance, *Bob* might be a temporary user working on project *X*, and regardless of whether *Bob* leaves the organization, his data must be retained until the end of the project, i.e., until it is erased explicitly by specifying *Project_X* for secure deletion.

Consider the initial state and suppose a secure deletion operation with attribute *Exp_2014* is invoked. Then p_2 and p_5 become inaccessible, but p_1 remains present as *Audit* has not been specified for deletion. If data protected with owner *Alice* securely deleted later, then p_3 becomes inaccessible but p_1 can still be retrieved.

2.4 System model

In a practical system supporting policy-based secure deletion, there are two distinct kinds of storage space: erasable and permanent memory. A small *erasable memory* forms the root of trust and must be under close control of the system operators. It is provided, for instance, by key-management systems, hardware-security modules (HSM), or local filesystems which support physical secure deletion. Content that has been deleted from the erasable memory is impossible to retrieve for both legitimate and malicious parties. On the other hand, *permanent memory* is readily available with large capacity, but its content can neither be erased nor hidden from an adversary. Many forms of storage encountered in practice fall in this category, ranging from the complex storage hierarchies of a data center to the mobile end-user devices attached to storage back-ends in the cloud.

In this work, we consider a user with access to erasable memory and permanent memory. Her goal is to store a potentially large number of files and to selectively delete files according to a deletion policy. Information to be stored is protected, resulting in ciphertext being written to permanent memory. The ciphertext is available later for accessing non-deleted files. Secure deletion operations make it impossible for an adversary to retrieve the deleted files, even if the adversary uses coercion and obtains all keys in the erasable memory. The constructions exploit the capability to remove data from erasable memory. (In other words, we assume a "bounded peek-a-boo adversary" according to Reardon et al. [24].)

We assume the adversary is passive and cannot modify data stored on either type of memory. In practice, one can ensure this easily through orthogonal data-authentication methods.

2.5 Secure deletion schemes

We now introduce the formal notion of a policy-based secure deletion scheme. The model is cryptographic and formulated as a secret-key scheme for simplicity.

We define a predicate $\mathsf{deleted}(G, D, p)$ for a deletion policy with attributes $\mathcal{A}$, policy graph G, and protection classes $\mathcal{P}$ that denotes whether deleting all attributes in a set $D \subseteq \mathcal{A}$ implies that the protection class $p \in \mathcal{P}$ should become inaccessible. In terms of the Boolean circuit interpretation of G, suppose those source nodes of G corresponding to the attributes in D are set to TRUE; then $\mathsf{deleted}(G, D, p)$ denotes the value of node p in G. The notation $[a, b]$ for two integers a and b denotes the set of integers $\{a, \ldots, b\}$; the expression $[a]$ is short for $[1, a]$.

Definition 1. A *policy-based deletion scheme* $\mathcal{E}$ is a tuple (Init, $\mathsf{Protect}$, Access, Delete), consisting of four probabilistic polynomial-time algorithms (in terms of a security parameter κ) with the following properties:

- $\mathsf{Init}(\kappa, G) \to (M_0, S_0)$

  The *initialization* algorithm takes as inputs the security parameter κ and a policy graph G. It outputs an initial master key M_0 and initial auxiliary state S_0.

- $\mathsf{Delete}(M_t, S_t, A_t) \to (M_{t+1}, S_{t+1})$

  The *secure deletion* algorithm takes as inputs a master key M_t, auxiliary state S_t, and a set of attributes $A_t \subseteq \mathcal{A}$, and outputs a new master key/auxiliary state pair reflecting the deletion of the supplied attributes.

  Throughout the operation of the scheme, a set $D \subseteq \mathcal{A}$ that contains the union of all attributes deleted so far is implicitly maintained. That is, after $t + 1$ calls to Delete, it holds $D = \cup_{i=0}^{t} A_i$.

- $\mathsf{Protect}(M_t, S_t, p, f) \to c$

The *protect* algorithm takes as inputs a master key M_t, auxiliary state S_t, a protection class $p \in \mathcal{P}$, and a file f which is a binary string of any length, and outputs a ciphertext c. If $\mathsf{deleted}(G, D, p) = \text{TRUE}$, i.e., the protection class has already been deleted, then $c = \bot$. Otherwise, the protection class is still accessible and $c \in \{0,1\}^*$ is a protected version of the file f.

- $\mathsf{Access}(M_t, S_t, p, c) \to f$

  The *access* algorithm takes as inputs a master key M_t, auxiliary state S_t, a protection class p, and a ciphertext c, and outputs a string $f \in \{0,1\}^*$ or $\bot$.

Whenever a master key/auxiliary state tuple (M_t, S_t) appears here, we assume that M_t and S_t are *well-formed* and result from a call to Init and a number of subsequent repeated calls to Delete. In other words, for some $\mathsf{Init}(\kappa, G) = (M_0, S_0)$ and a sequence $A_0, A_1, \ldots, A_{t-1}$ of subsets of $\mathcal{A}$, it holds $\mathsf{Delete}(M_i, S_i, A_i) = (M_{i+1}, S_{i+1})$, for $i \in [t-1]$. Note that this assumption incurs no loss of generality in the adversarial model considered here.

All four algorithms except for Access are usually probabilistic; they output a random variable induced by their internal random choices. In statements about particular output values of an algorithm, such as in the preceding paragraphs, it is implied that these outputs may occur with non-zero probability.

Next, we discuss the completeness and security properties of a policy-based deletion scheme $\mathcal{E}$.

Definition 2. A policy-based deletion scheme defined as above is said to be *complete* if any protected file can be accessed at a later time unless it has been deleted. That is, for any t and $j \leq t$, for all $p \in \mathcal{P}$, for all $f \in \{0,1\}^*$, for all $\{A_i\}_{i=0}^{t-1}$, where $A_i \subseteq \mathcal{A}$, and all key/state tuples (M_i, S_i), for $i \in [t-1]$, such that $\mathsf{Init}(\kappa, G) = (M_0, S_0)$ and $\mathsf{Delete}(M_i, S_i, A_i) = (M_{i+1}, S_{i+1})$, it holds that

$$\mathsf{Access}\big(M_t, S_t, p, \mathsf{Protect}(M_j, S_j, p, f)\big) = f.$$

conditioned on $\mathsf{deleted}(G, \cup_{i=0}^{t-1} A_i, p) = \text{FALSE}$.

The security of a policy-based deletion scheme is defined using the following experiment for an adversary A and a security parameter κ.

Secure deletion experiment $\mathsf{Secdel}_{\mathsf{A},\mathcal{E}}(\kappa)$ **:**

1. The adversary A is given κ and outputs a policy graph G with corresponding sets $\mathcal{A}$ and $\mathcal{P}$ for the attributes and the protection classes, respectively. Also, the adversary outputs a set $\mathcal{D} \subseteq \mathcal{A}$ of all attributes to be deleted at the end.
   Then, algorithm $\mathsf{Init}(\kappa, G) \to (M_0, S_0)$ is executed and S_0 is given to A.
2. A is given oracle access to protection and deletion operations. In particular, the set D of deleted attributes, the index t, and the current master key/auxiliary state pair (M_t, S_t) are maintained; A may choose inputs (p, f) for protection and receives the output of $\mathsf{Protect}(M_t, S_t, p, f)$; A may also specify $A_t \subseteq \mathcal{D}$, which causes algorithm $\mathsf{Delete}(M_t, S_t, A_t) \to (M_{t+1}, S_{t+1})$ to be invoked, then A receives S_{t+1}.
3. The adversary A outputs some $p^* \in \mathcal{P}$ such that $\mathsf{deleted}(G, D, p^*) = \text{FALSE}$ and two strings $f_0, f_1 \in \{0,1\}^*$ of the same length.
4. After a random bit $b \leftarrow \{0,1\}$ is chosen, a ciphertext $c^* \leftarrow \mathsf{Protect}(M_t, S_t, p^*, f_b)$ is computed and given to A.
5. The adversary is given further oracle access to protection and deletion operations, continued from step 2, until A stops under the condition that $\mathsf{deleted}(G, D, p^*) = \text{TRUE}$ and $D = \mathcal{D}$, i.e., p^* is inaccessible for the current set D of deleted attributes and that set is the same as the one defined in step 1.

6. A receives the current value of M_t and outputs a bit $\hat{b}$. The experiment returns 1 if $\hat{b} = b$ and 0 otherwise.

Definition 3. A *policy-based deletion scheme* $\mathcal{E}$ is called *secure* when for all probabilistic polynomial-time adversaries A, there exists a negligible function ε such that

$$\Pr\Big[\mathsf{Secdel}_{\mathsf{A},\mathcal{E}}(\kappa) = 1\Big] \;\leq\; \frac{1}{2} + \varepsilon(\kappa).$$

Remark. A secure deletion scheme maintains the secrecy of the protected and deleted content. According to the security definition, in the last step of Secdel, the adversary receives the master key. At this point, the files protected under all classes that have not yet been deleted are obviously exposed to A. However, any protected file that has already been deleted is guaranteed to remain confidential, even after the master key has been leaked.

Note that the security model requires the adversary to declare the attributes to delete upfront, before it can observe any output produced by the scheme and adaptively choose in what order to delete these attributes. Our security notion is similar to "selective security" for attribute-based encryption [25, 14].

The security notion may readily be extended to cover adversarial modifications to permanent memory, analogous to chosen-ciphertext attacks against encryption schemes. As storage systems typically protect data integrity through different means (that additionally prevent replay attacks), we omit this extension for clarity.

2.6 Measuring efficiency

We will characterize implementations of secure deletion schemes in terms of the cost incurred for executing their operations. We define the *deletion cost*, *protection cost*, and *access cost* as the complexities of running the deletion, protection, and access algorithms, respectively. Complexities are expressed in terms of computation steps or, more usually, through the number of calls to an encryption primitive made by the algorithm.

We consider deletion schemes with *constant* deletion cost, independent of the number of protected files, to be the most interesting. For the constructions considered in this work, protection and access cost do not differ, hence we are mainly interested in access cost.

Furthermore, the size of the erasable memory for storing the master key and the permanent memory for storing the auxiliary state are important parameters. We quantify them as the *master-key size* and *auxiliary-state size*, respectively. Note the protected files must be maintained outside the secure deletion scheme.

3 Constructions

3.1 Prerequisites

In the constructions described below, we assume for simplicity that master keys and auxiliary state values M_t and S_t, returned by Init and Delete operations, are associative arrays indexed by nodes and edges of G. Thus, the values $S_t[v]$ and $S_t[e]$ denote the auxiliary data, if any, associated to $v \in V$ and $e \in E$, respectively. The notation $S_t|^{V',E'}$ denotes only the collection of entries of S_t restricted to $v \in V' \subseteq V$ and $e \in E' \subseteq E$; of course, we require $E' \subseteq V' \times V'$ for such restrictions.

Secret-key encryption schemes. A *secret-key encryption scheme* $\mathcal{S}$ consists of three algorithms Keygen, $\mathsf{Encrypt}$, and $\mathsf{Decrypt}$. The probabilistic key generation algorithm $\mathsf{Keygen}(\kappa)$ outputs a key K; algorithm $\mathsf{Encrypt}(K, m)$ takes a key K and a plaintext m as inputs and returns a ciphertext c; algorithm $\mathsf{Decrypt}(K, c)$ takes a key K and a ciphertext c as inputs and returns a plaintext m. We assume $\mathcal{S}$ is complete and IND-CPA secure according to the standard notions [18].

Secret-sharing schemes. A $(m + 1)$-out-of-n secret-sharing scheme denotes a method to split a *secret* s into n *shares* $s_1, \ldots, s_n$ such that $m + 1$ or more shares are sufficient to recover s, but m or fewer shares give away no statistical information about s. The operation of sharing s is expressed by $(s_1, \ldots, s_n) \leftarrow \mathsf{Share}^n_{m+1}(s)$, and the algorithm to recover s from shares $\bar{s}_1, \ldots, \bar{s}_{m+1}$ is written as $s \leftarrow \mathsf{Recover}(\bar{s}_1, \ldots, \bar{s}_{m+1})$. We use, for instance, the well-known implementation based on polynomial interpolation [26].

3.2 Direct secure deletion schemes

We now introduce a class of secure deletion schemes with a particularly simple implementation of the deletion operation. We call them *direct* because their deletion operation merely erases parts of the master key that corresponds directly to the deleted attributes.

More precisely, a *direct secure deletion scheme* always generates a master key M_t in the form of a tuple with exactly one component for every attribute in $\mathcal{A}$. The deletion operation for a set of attributes $A_t \subseteq \mathcal{A}$ erases those components of M_t that correspond to A_t. In other words, every master key M_t is an associative array indexed by $a \in \mathcal{A}$, where $M_t[a]$ denotes the component corresponding to a. The deletion operation for A_t computes M_{t+1} as

$$M_{t+1}[a] \leftarrow \begin{cases} \perp & \text{if } a \in A_t \\ M_t[a] & \text{otherwise.} \end{cases}$$

Stronger security for direct schemes. Recall from Definition 3 that in the last step of its experiment, the adversary is given the current value of the master key. Given that $\mathcal{D}$ is supplied by A, the set of all attributes to be deleted eventually, in the first step, it is clear that the master key given to the adversary at the end is $M_0|^{\mathcal{A} \setminus \mathcal{D}}$. As a consequence, A can receive it in the first rather than last step, hence providing the adversary with more information earlier in the experiment. We use this stronger security definition when showing the security of direct policy-based deletion schemes.

3.3 Basic scheme

A secure deletion scheme with a very basic policy can be implemented directly from a secret-key encryption scheme $\mathcal{S}$ and a secret-sharing scheme.

Let $G = (V, E)$ be a policy graph with $n \geq 1$ source nodes, connected to a single interior node that is also the only protection class. In other words, as shown in Figure 2, the nodes $V = \mathcal{A} \cup \mathcal{P}$ are given by a set of attributes $\mathcal{A} = \{a_1, \ldots, a_n\}$ and a set $\mathcal{P} = \{p\}$ composed of a single protection class, and the edges are $E = \{e_1, \ldots, e_n\}$, where $e_i = (a_i, p)$ for $i \in [n]$. The interior node p has a threshold parameter m.

We construct $\mathcal{E}$ = (Init, $\mathsf{Protect}$, Access, Delete) as follows:

- $\mathsf{Init}(\kappa, G)$
  - For each attribute $a_i \in \mathcal{A}$ select a random key $K_i \leftarrow \mathcal{S}.\mathsf{Keygen}(\kappa)$ and set $M_0[a_i] \leftarrow K_i$;

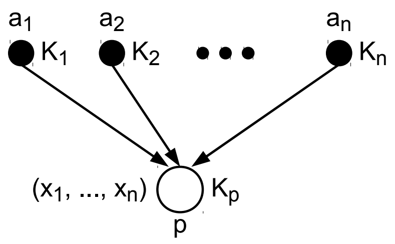

Figure 2. Basic secure deletion scheme, implemented from encryption and secret sharing.

  - Select a random key $K_p \leftarrow \mathcal{S}.\mathsf{Keygen}(\kappa)$ and construct a $(n - m + 1)$-out-of-n secret sharing for the key, i.e., $(s_1, \ldots, s_n) \leftarrow \mathsf{Share}^n_{n-m+1}(K_p)$; then, compute $x_i \leftarrow \mathcal{S}.\mathsf{Encrypt}(K_i, s_i)$ for $i \in [n]$ and set the initial auxiliary state to be $S_0[p] \leftarrow (x_1, \ldots, x_n)$;
  - Output (M_0, S_0).

- $\mathsf{Delete}(M_t, S_t, A_t)$

  As this is a direct secure deletion scheme, proceed as defined in Section 3.2.

- $\mathsf{Protect}(M_t, S_t, p, f)$

  If $\mathsf{deleted}(G, D, p) = \text{TRUE}$, set $c \leftarrow \perp$. Otherwise, let $(x_1, \ldots, x_n) \leftarrow S_t[p]$; for each $i \in [n]$ such that $a_i \notin D$ compute the share $s_i \leftarrow \mathcal{S}.\mathsf{Decrypt}(M_t[a_i], x_i)$ and reconstruct the key K_p from the shares; finally, compute $c \leftarrow \mathcal{S}.\mathsf{Encrypt}(K_p, f)$. Output c as a protected version of the file f.

- $\mathsf{Access}(M_t, S_t, p, c)$

  If $\mathsf{deleted}(G, D, p) = \text{FALSE}$, reconstruct K_p as above and output $f \leftarrow \mathcal{S}.\mathsf{Decrypt}(K_p, c)$. Otherwise, when the protection class is inaccessible, return $f \leftarrow \perp$.

The auxiliary state S_t contains data only for the interior node p, i.e., the shares of the key K_p encrypted under the keys representing the n attributes. Note that the $\mathsf{Protect}$ and Access methods obtain at least $n - m + 1$ shares for reconstructing K_p whenever $\mathsf{deleted}(G, D, p) = \text{FALSE}$; this is shown in the next theorem.

Theorem 1. *The direct secure deletion scheme $\mathcal{E}$ with a policy graph G defined above is complete and secure.*

Proof. Given the structure of G_1 with p being an interior node with threshold m, it follows that the predicate $\mathsf{deleted}(G_1, D, p)$ is TRUE if and only if $|D| \geq m$.

It is easy to check that the scheme is complete because as long as $\mathsf{deleted}(G_1, D, p) = \text{FALSE}$, fewer than m attribute keys have been deleted; in other words, *more than* $n - m$ of the keys $K_1, \ldots, K_n$ are present in the master key. Using the auxiliary data $S_t[p]$, this allows to recover more than $n - m$ of the secret shares $s_1, \ldots, s_n$. Then the key K_p can be obtained by running algorithm $\mathsf{Recover}$ of the secret-sharing scheme. Hence, $\mathsf{Protect}$ and Access can operate on all files protected under p.

The proof that the scheme is secure proceeds in a sequence of games [27]:

Game 0. This is the original game and its experiment is defined according to the security definition. Note that in the first step of the experiment, A is required to specify the set of attributes $\mathcal{D}$ which satisfies $\mathsf{deleted}(G_1, \mathcal{D}, p) = \text{TRUE}$. As mentioned above, this happens if and only if $|\mathcal{D}| \geq m$.

Game 1. We proceed as in the previous game except that for each $i \in [n]$ such that $a_i \in \mathcal{D}$, the Init algorithm sets $x_i \leftarrow \mathcal{S}.\mathsf{Encrypt}(K_i, r_i)$, for a randomly chosen $r_i \in \{0,1\}^{|s_i|}$, and stores these in the auxiliary state value $S_0[p] = (x_1, \ldots, x_n)$. By the security of the encryption scheme $\mathcal{S}$ and the fact that all K_i with $a_i \in \mathcal{D}$ are removed from the master key before it is revealed to A, it holds that the adversary's advantage changes by at most negligible probability from the previous game.

Game 2. This game is initially the same as the previous game, in which Init chooses a random key $K_p \leftarrow \mathcal{S}.\mathsf{Keygen}(\kappa)$, secret-shares it to obtain the shares $s_1, \ldots, s_n$, and uses these for computing the auxiliary state. However, we also choose a different random key $K'_p \leftarrow \mathcal{S}.\mathsf{Keygen}(\kappa)$ and use this key rather than K_p to protect and access any files under p in responses to queries of A.

Recall that A will ask for at least m attributes to be deleted before it obtains the master key. Then, by the security of the secret-sharing scheme, the adversary cannot tell which key is secret-shared, hence behaves like in the previous game except for a negligible difference. Also, note that the key K'_p is completely independent from the master key and the auxiliary state.

Game 3. In this last game, when the adversary presents files f_0 and f_1 with the same length, the challenger computes $c^* \leftarrow \mathcal{S}.\mathsf{Encrypt}(K'_p, r)$ for a randomly chosen $r \in \{0,1\}^*$ of the same length as the files. By the security of the encryption scheme $\mathcal{S}$, the advantage of A changes at most by a negligible amount from the previous game. Moreover, as r is independent of f_1 and f_2, the adversary cannot do better than guess the bit b at random, hence $\Pr[\hat{b} = b] = \frac{1}{2}$.

As the adversary A can guess b exactly with probability $\frac{1}{2}$ in the last game and her advantage changes only negligibly between every two consecutive games, it follows that $\mathcal{E}$ is secure. $\square$

Note that the auxiliary state contains values neither for the attributes nor for the edges incident with them. We use this property when constructing more complex secure deletion schemes and maintain this invariant for all *direct* schemes.

3.4 Composition

Two secure deletion schemes can be composed into a more elaborate scheme whose policy graph results from combining the two basic policy graphs. We consider first the case when both schemes are *direct* and relax this requirement later on.

Suppose the two secure deletion schemes are arranged in a hierarchy as a *higher* and a *lower* scheme. The key step of the composition uses the higher secure deletion scheme to protect the master key of the lower scheme. The protection classes applied to particular components of the lower scheme's master key will determine the policy graph of the resulting scheme. Importantly, the lower master key needs no longer be stored in the erasable memory. In this way, secure deletion operations of the higher scheme extend to files protected with the lower scheme.

Let $\mathcal{E}_h$ and $\mathcal{E}_l$ be secure deletion schemes with policy graphs $G_h = (V_h, E_h)$ and $G_l = (V_l, E_l)$, protection classes $\mathcal{P}_h$ and $\mathcal{P}_l$, and potentially overlapping attribute sets $\mathcal{A}_h$ and $\mathcal{A}_l$, respectively. Apart from $\mathcal{A}_h$ and $\mathcal{A}_l$, the nodes V_h and V_l in the two graphs are mutually exclusive. Then, a secure deletion scheme $\mathcal{E}$ with policy graph G, attributes $\mathcal{A}$, and protection classes $\mathcal{P}$ is constructed as follows.

The composition specifies two sets $\mathcal{P}_J$ and $\mathcal{A}_J$, where $\mathcal{P}_J \subseteq \mathcal{P}_h$ and $\mathcal{A}_J \subseteq \mathcal{A}_l$, as well as a map $J : \mathcal{A}_J \to \mathcal{P}_J$. The map J determines to which protection class in $\mathcal{P}_J$ each attribute of $\mathcal{A}_J$ is joined. Also, we require that $\mathcal{A}_J \cap \mathcal{A}_h = \emptyset$, i.e., the attributes to be joined contain no attributes of G_h, so that the new policy graph will not contain cycles. The policy graph $G = (V, E)$ is defined as:

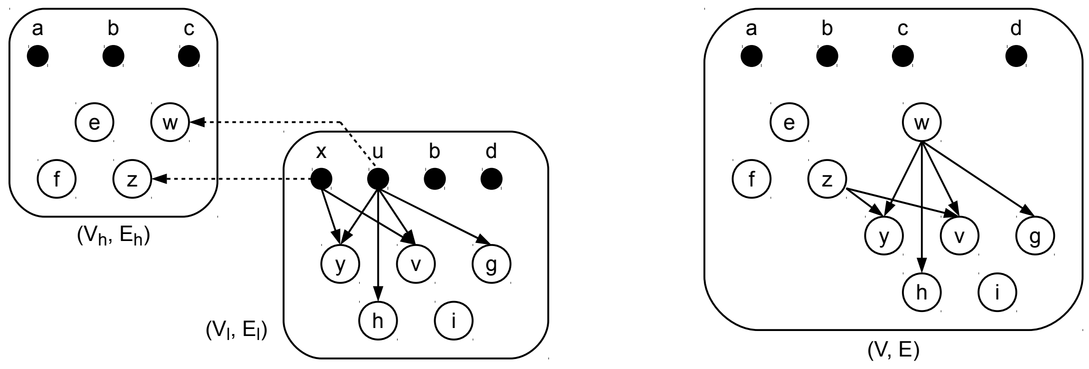

Figure 3. Two policy graphs (V_h, E_h) and (V_l, E_l) on the left are combined to (V, E), shown right. Note that $\mathcal{A}_J = \{x, u\}$ and $\mathcal{P}_J = \{w, z\}$, and the composition map specifies $J(x) = z$ and $J(u) = w$. Only the edges of case (3) in $\mathcal{E}.\mathsf{Init}$ are drawn here; the remaining edges are omitted and not affected by the composition.

- $V = V_h \cup V_l \setminus \mathcal{A}_J$, i.e., G contains all nodes from G_h and G_l except those attributes of G_l which are being joined to a protection class in G_h;
- $E = E_h \cup \{(u, v) \mid (u, v) \in E_l \wedge u \notin \mathcal{A}_J\} \cup \{(w, v) \mid (u, v) \in E_l \wedge u \in \mathcal{A}_J \wedge w = J(u)\}$; this denotes (1) all edges of G_h, (2) the edges of G_l not incident to nodes in $\mathcal{A}_J$, and (3) one edge (w, v) for every edge $(u, v) \in E_l$ with $u \in \mathcal{A}_J$, where $w = J(u)$ is the protection class to which u is joined according to the composition. Note that $|E| = |E_h| + |E_l|$ and there is a one-to-one mapping between E and $E_h \cup E_l$ determined by J and the original graph.

The attributes of $\mathcal{E}$ are all attributes of G_h and G_l except for those involved in the composition, i.e., $\mathcal{A} = \mathcal{A}_h \cup \mathcal{A}_l \setminus \mathcal{A}_J$, and the set of protection classes is the union of the present protection classes, $\mathcal{P} = \mathcal{P}_h \cup \mathcal{P}_l$. An illustration of how two policy graphs are composed is shown in Figure 3.

The algorithms (Init, $\mathsf{Protect}$, Access, Delete) of $\mathcal{E}$ are composed from those of $\mathcal{E}_h$ and $\mathcal{E}_l$:

- $\mathcal{E}.\mathsf{Init}(\kappa, G)$

  Compute $(M_{h,0}, S_{h,0}) \leftarrow \mathcal{E}_h.\mathsf{Init}(\kappa, G_h)$ and $(M_{l,0}, S_{l,0}) \leftarrow \mathcal{E}_l.\mathsf{Init}(\kappa, G_l)$, and for every $a \in \mathcal{A}$ set

  $$M_0[a] = \begin{cases} M_{h,0}[a] & \text{if } a \in \mathcal{A}_h \text{ and } a \notin \mathcal{A}_l, \\ M_{l,0}[a] & \text{if } a \notin \mathcal{A}_h \text{ and } a \in \mathcal{A}_l, \\ M_{h,0}[a] \parallel M_{l,0}[a] & \text{if } a \in \mathcal{A}_h \cap \mathcal{A}_l, \end{cases}$$

  where $\parallel$ denotes the concatenation of two entries or tuples into one tuple. Note that $M_0[a]$ may contain more than one entry; in general, $M_0[a]$ is a tuple with one entry for every edge incident to node a. For notational convenience, we write $M_0[a][v]$ for the entry in $M_0[a]$ that corresponds to the specific edge from a to v.

  The initial auxiliary state stores no data for the attributes in our constructions.[1] For every protection class $p \in V$, the auxiliary data $S_0[p]$ is simply $S_{h,0}[p]$ or $S_{l,0}[p]$ respective to whether $p \in V_h$ or $p \in V_l$. As there is one-to-one mapping between E and $E_h \cup E_l$, we consider every

[1]This may not necessarily be true for other constructions; should that be the case, auxiliary data for $\mathcal{A}_J$ specified by the composition map can be stored either in the auxiliary data of the nodes to which they are joined or in the new edges created in the composition, i.e., those in case (3) described above.

edge $e = (u, v) \in E_h \cup E_l$ and define the initial auxiliary state of its corresponding edge in E as follows:

- $S_0[(u, v)] = S_{h,0}[(u, v)]$, if $(u, v) \in E_h$;
- $S_0[(u, v)] = S_{l,0}[(u, v)]$, if $(u, v) \in E_l \wedge u \notin \mathcal{A}_J$;
- $S_0[(w, v)] = S_{l,0}[(u, v)] \parallel \mathcal{E}_h.\mathsf{Protect}(M_{h,0}, S_{h,0}, w, M_{l,0}[u][v])$, if $(u, v) \in E_l$ for $u \in \mathcal{A}_J$ and $w = J(u)$, where $w \in \mathcal{P}_J$ is the protection class to which $u \in \mathcal{A}_J$ is joined.

That is, all edges in E which are also present in E_h or E_l keep their initial auxiliary state unchanged. The edges from a node $w \in V_h$ to a node $v \in V_l$ store the auxiliary data of (u, v) and the master-key component from $\mathcal{E}_l$ related to (u, v) protected under w with $\mathcal{E}_h$; here, $u \in V_l$ is the attribute which was joined to w such that the edge $(u, v) \in E_l$ corresponds to $(w, v) \in E$. Note that in our construction, edges adjacent to the attributes have no auxiliary data; thus, in the last case, $S_0[(w, v)] = \mathcal{E}_h.\mathsf{Protect}(M_{h,0}, S_{h,0}, w, M_{l,0}[u][v])$.

Protecting the components of the master key from the lower scheme with the higher scheme represents the main mechanism of the composition. It is also the building block for the hierarchical implementation of secure deletion schemes with arbitrary policy graphs, as explained in Section 3.5.

- $\mathcal{E}.\mathsf{Delete}(M_t, S_t, A_t)$

  Execute $\mathcal{E}_h.\mathsf{Delete}(M_t|^{V_h,E_h}, S_t|^{V_h,E_h}, A_t \cap \mathcal{A}_h)$ and $\mathcal{E}_l.\mathsf{Delete}(M_t|^{V_l,E_l}, S_t|^{V_l,E_l}, A_t \cap \mathcal{A}_l)$, and update the master key and auxiliary state accordingly. Recall that when a scheme is direct, the deletion operation does not change the auxiliary state but only deletes part of the master key; hence, the deletion operations are trivial.

- $\mathcal{E}.\mathsf{Protect}(M_t, S_t, p, f)$

  - If $p \in \mathcal{P}_h$, then $c \leftarrow \mathcal{E}_h.\mathsf{Protect}(M_t|^{V_h,E_h}, S_t|^{V_h,E_h}, p, f)$;
  - Otherwise, when $p \in \mathcal{P}_l$, we reconstruct $M_{l,t}$ by computing for $a \in \mathcal{A}_l$:

  $$M_{l,t}[a] = \begin{cases} M_t[a] & \text{if } a \notin \mathcal{A}_J \\ \parallel_{(a,v) \in E_l} \mathcal{E}_h.\mathsf{Access}(M_t|^{V_h,E_h}, S_t|^{V_h,E_h}, w, S_t[(w, v)]) & \text{if } a \in \mathcal{A}_J, \end{cases}$$

    where $w = J(a)$ and $\parallel$ stands for the concatenation of multiple entries into a tuple.

    Then, compute $c \leftarrow \mathcal{E}_l.\mathsf{Protect}(M_{l,t}, S_t|^{V_l,E_l}, p, f)$. Note that any node $u \in V_l$ and any edge $(u, v) \in E_l$, where $u \in \mathcal{A}_J$, are not present in G; hence, they are not defined in S_t and its restriction $S_t|^{V_l,E_l}$. This agrees with the invariant defined in the last section that a *direct* secure deletion scheme stores no data in the auxiliary state for attributes and edges incident with them.[2]

  Return c as the protected version of f.

- $\mathcal{E}.\mathsf{Access}(M_t, S_t, p, c)$:

  This operation proceeds analogously to $\mathsf{Protect}$:

  - If $p \in \mathcal{P}_h$, then compute $f \leftarrow \mathcal{E}_h.\mathsf{Access}(M_t|^{V_h,E_h}, S_t|^{V_h,E_h}, p, c)$;

[2]In the general case, when there is auxiliary data associated with attributes or their adjacent edges, we adjust the reconstruction of $M_{l,t}$ and $S_{l,t}$ easily to match the modification needed in that case as described in the Init algorithm.

– If $p \in \mathcal{P}_l$, then reconstruct the master key $M_{l,t}$ of $\mathcal{E}_l$ as described in the $\mathcal{E}$.$\mathsf{Protect}$ algorithm and compute $f \leftarrow \mathcal{E}_l.\mathsf{Access}(M_{l,t}, S_t|^{V_l,E_l}, p, c)$.

Finally, return f.

Theorem 2. *The above policy-based secure deletion scheme $\mathcal{E}$ is* direct*, complete, and secure if $\mathcal{E}_h$ and $\mathcal{E}_l$ are* direct *policy-based secure deletion schemes,*

Proof. It is easy to show that $\mathcal{E}$ is complete from the fact that $\mathcal{E}_h$ and $\mathcal{E}_l$ are complete and from the way they are composed.

The proof of security proceeds in a sequence of games. In each game, we modify slightly further $\mathcal{E}$ when an algorithm B plays the role of a challenger in the experiment $\mathsf{Secdel}_{\mathsf{A},\mathcal{E}}(\kappa)$ against an adversary A attacking $\mathcal{E}$ by running the experiments $\mathsf{Secdel}_{\mathsf{B},\mathcal{E}_h}(\kappa)$ and/or $\mathsf{Secdel}_{\mathsf{B},\mathcal{E}_l}(\kappa)$; note that B plays the role of a challenger against A for $\mathcal{E}$ and the role of an adversary for $\mathcal{E}_h$ and/or $\mathcal{E}_l$.

Game 0. The experiment $\mathsf{Secdel}_{\mathsf{A},\mathcal{E}}(\kappa)$ proceeds according to its definition for the $\mathcal{E}$ defined above.

Game 1. The simulator B performs the experiment $\mathsf{Secdel}_{\mathsf{B},\mathcal{E}_h}(\kappa)$ in the role of the adversary, while simulating $\mathsf{Secdel}_{\mathsf{A},\mathcal{E}}(\kappa)$ for A. The scheme $\mathcal{E}$ is modified to use oracle access to protection and deletion operations for $\mathcal{E}_h$, and every time A moves from one step to the next in her experiment $\mathsf{Secdel}_{\mathsf{A},\mathcal{E}}(\kappa)$, so does B in $\mathsf{Secdel}_{\mathsf{B},\mathcal{E}_h}(\kappa)$.

In particular, for a set $\mathcal{D}$ given by A in the first step of her experiment, B outputs $\mathcal{D}_h \leftarrow \mathcal{D}|^{V_h,E_h}$ in the first step of his experiment. The $\mathcal{E}$.Init algorithm computes the auxiliary state as defined with the exception that B runs only $(M_{l,0}, S_{l,0}) \leftarrow \mathcal{E}_l.\mathsf{Init}(\kappa, G_l)$, whereas $S_{h,0}$ is obtained from the challenger in his experiment; at any time, the respective induced calls to $\mathcal{E}_h.\mathsf{Delete}$, $\mathcal{E}_h.\mathsf{Protect}$, and $\mathcal{E}_h.\mathsf{Access}$ are replaced by oracle calls provided to B by the challenger of $\mathsf{Secdel}_{\mathsf{B},\mathcal{E}_h}(\kappa)$.

This game is indistinguishable to A from the previous one as the only difference is the way B handles the recursive calls to $\mathcal{E}_h$ which are perfectly simulated from the experiment B runs against his challenger.

Game 2. This game is identical to the previous one except for computing the auxiliary state for edges in case (3) in the $\mathcal{E}$.Init algorithm (see Figure 3). Previously, the auxiliary value for the edge in G corresponding to $(u, v) \in E_h \cup E_l$, for $u \in \mathcal{A}_J$, was computed as

$$S_0[(w, v)] = \mathcal{E}_h.\mathsf{Protect}(M_{h,0}, S_{h,0}, w, M_{l,0}[u][v]),$$

where $(u, v) \in E_l$ for $u \in \mathcal{A}_J$ and $w = J(u)$.

For the same $u \in \mathcal{A}_J$ and $w = J(u)$, if $\mathsf{deleted}(G, \mathcal{D}, w) = \textsc{True}$, this is modified to:

$$S_0[(w, v)] = \mathcal{E}_h.\mathsf{Protect}(M_{h,0}, S_{h,0}, w, r),$$

for a randomly chosen $r \leftarrow \{0, 1\}^*$ of the same length as the corresponding key component.

This game is indistinguishable to A from the previous one except for a negligible probability due to the security of $\mathcal{E}_h$; in particular, for protection classes which are deleted before the master key is revealed, A cannot distinguish between a chosen value or a random value being protected under such a class.

Game 3. The simulator B performs the experiments $\mathsf{Secdel}_{\mathsf{B},\mathcal{E}_h}(\kappa)$ and $\mathsf{Secdel}_{\mathsf{B},\mathcal{E}_l}(\kappa)$ while simulating $\mathsf{Secdel}_{\mathsf{A},\mathcal{E}}(\kappa)$ for A. In the first step of the main experiment, A returns a set $\mathcal{D}$, from which B derives his set $\mathcal{D}_h \leftarrow \mathcal{D}|^{V_h,E_h}$ for the $\mathsf{Secdel}_{\mathsf{B},\mathcal{E}_h}(\kappa)$ experiment as before; for the experiment $\mathsf{Secdel}_{\mathsf{B},\mathcal{E}_l}(\kappa)$, $\mathcal{D}_l$ is computed as the union of all attributes $a \in \mathcal{A}_l$ such that:

- $a \notin \mathcal{A}_J \wedge a \in \mathcal{D}$; or
- $a \in \mathcal{A}_J \wedge \mathsf{deleted}(G, \mathcal{D}, J(a)) = \text{TRUE}$.

That is, all attributes of $\mathcal{A}_l$ which are present in G and belong to $\mathcal{D}$ *or* which are attributes joined in the composition and whose protection classes from G_h become inaccessible when all attributes from $\mathcal{D}$ are deleted. Once B returns $\mathcal{D}_h$ and $\mathcal{D}_l$ in the first step of the respective experiments against his challengers, he is able to compute M_0, S_0 for $\mathcal{E}$ as defined in the previous game. As before, every time A moves from one step to the next in her experiment so does B for his.

This game is indistinguishable to A from the previous one as the only difference is the way B handles the respective induced calls to $\mathcal{E}_l$ which are perfectly simulated from experiment that B runs against his challengers.

Note that in the last game B simulates $\mathsf{Secdel}_{\mathsf{A},\mathcal{E}}(\kappa)$ perfectly without the knowledge of $M_{h,0}$ and $M_{l,0}$ using only oracle access provided by $\mathsf{Secdel}_{\mathsf{B},\mathcal{E}_h}(\kappa)$ and $\mathsf{Secdel}_{\mathsf{B},\mathcal{E}_l}(\kappa)$. Hence, any adversary A winning $\mathsf{Secdel}_{\mathsf{A},\mathcal{E}}(\kappa)$ with non-negligible advantage implies an adversary B which can break the security of $\mathcal{E}_h$ or $\mathcal{E}_l$. Therefore, if $\mathcal{E}_h$ and $\mathcal{E}_l$ are secure, so is the composed $\mathcal{E}$. □

Next, we observe that we can relax the requirement that $\mathcal{E}_h$ is a direct scheme if $\mathcal{E}_h$ and $\mathcal{E}_l$ share no attributes; note this implies $V_h \cap V_l = \emptyset$ as the composition requires that the nodes of both graphs are disjoint except for the attributes. Hence, one may compose an arbitrary policy-based secure deletion scheme $\mathcal{E}_h$ with a *direct* scheme $\mathcal{E}_l$.

Theorem 3. *Let $\mathcal{E}_h$ and $\mathcal{E}_l$ be policy-based secure deletion schemes, where $\mathcal{E}_l$ is* direct. *If $V_h \cap V_l = \emptyset$, the two schemes can be composed as described above and the resulting scheme $\mathcal{E}$ is complete and secure.*

Proof. The construction above and proof do not utilize the fact that $\mathcal{E}_h$ is direct except for the possibility that $\mathcal{E}_h$ and $\mathcal{E}_l$ share attributes. Thus, if $\mathcal{A}_h \cap \mathcal{A}_l = \emptyset$, the result follows analogously. □

3.5 Direct policy-graph construction

One may use the basic secure deletion scheme of Section 3.3 and the composition operation to implement *direct* secure deletion schemes for arbitrary policy graphs. Recall that every DAG $G = (V, E)$ has a topological order, computable in time $O(|V| + |E|)$, which arranges the nodes of G in a sequence that respects the direction of all edges.

We traverse G in the topological order and gradually build up a secure deletion scheme for G. Initially we take the first encountered interior node v_0 and implement a secure deletion scheme for v_0 according to the basic scheme of Section 3.3. Subsequently, whenever we encounter the next interior node v, we take the subgraph G_v induced by v and its incoming edges, implement a basic secure deletion scheme for G_v according to Section 3.3, and compose it with the secure deletion scheme realized so far.

The resulting secure deletion scheme for G implicitly contains a key for every node and a key for certain edges. More precisely, for a source node (attribute) a, the key K_a is explicitly stored in the master key and has one entry for every outgoing edge; for any interior node (protection class) p with n incoming edges and threshold m, the key K_p is stored encrypted as follows. For every incoming edge (v, p), except for those incident to an attribute, the auxiliary state contains an encryption of a key $K_{v,p}$ under the key K_v, which corresponds to node v. In turn, the auxiliary state associated to p contains a vector $(x_{p,1}, \ldots, x_{p,n})$, where some $x_{p,j}$ is associated to v and represents an encryption of $s_{p,j}$ under $K_{v,p}$. The value $s_{p,j}$ is a share of K_p in a $(m+1)$-out-of-n secret sharing scheme.

Thus, every node in G is associated with one encryption key and every edge in G not incident to a source is also associated with one encryption key. The resulting structure is an iterative key-encrypting key-assignment scheme (IKEKAS) according to Crampton et al. [6].

The access cost is proportional to the size of G. More precisely, accessing a node p needs a maximum of two secret-key cryptographic operations for every edge that must be traversed (i.e., set to TRUE), in order to derive p (i.e., set p to TRUE). The master key contains one component for every attribute and the total size of the auxiliary state is in $O(|V| + |E|)$.

Observe that one may eliminate the encryption keys associated to the edges and encrypt the share of a child node directly with the key of the parent node. This reduces the number of cryptographic operations without impacting the security of the construction. However, the resulting scheme cannot obtained from the modular composition operation in Section 3.4, and a detailed security proof is left for future work.

Furthermore, the master key component for an attribute a, the key K_a, is a tuple of keys with one entry for every outgoing edge. This is done so that the resulting secure deletion scheme is the same, regardless of which topological order is used during composition, as the edges adjacent to attributes are used in a special way in the composition. However, once constructed, one can replace the tuple K_a with a single key K'_a, and for every edge (a, p) store the key $K_{a,p}$ protected with K'_a in the auxiliary state of the edge (a, p), as done for all other edges; this reduces the master-key size.

3.6 Tree construction

In early work on secure deletion, Di Crescenzo et al. [8] introduce a tree construction that protects data in an arbitrary number of emulated memory locations. The scheme is realized from persistent storage exposed to an adversary and allows to overwrite individual memory locations. Only a small erasable memory of constant size is needed for maintaining a master key.

Using our terminology, their scheme permits n protection classes $p_1, \ldots, p_n$ and each one can be specified for deletion independently of the others. The deletion policy graph consists of n attributes $a_1, \ldots, a_n$, the n protection classes $p_1, \ldots, p_n$, and n edges (a_i, p_i) for $i \in [n]$.

A balanced tree with n leaves, labeled by $p_1, \ldots, p_n$, is constructed as follows. First, a key K_v of a secret-key encryption scheme is generated, ranging over all nodes v in the tree. Next, for every node v, the keys of all children of v are encrypted with K_v and the resulting ciphertext is added to the auxiliary state associated to v. The key of the root node represents the master key; is not stored in the auxiliary state.

For protecting or for accessing a file under protection class p_i, all keys along the path from the root to p_i are decrypted, starting from the root, and K_{p_i} is used to encrypt or decrypt the file, respectively.

Deletion for attribute a_i makes all data protected under p_i inaccessible. This operation is implemented by generating fresh keys for all nodes on the path from the root to p_i in the tree. More precisely, the keys of all nodes and their siblings along this path are first decrypted, then a fresh key is generated for every node on the path except for p_i, and finally all fresh keys are encrypted under the fresh key of the respective parent node and stored in the auxiliary state. The fresh root key is written to the master key and the previous root key is deleted in the erasable memory.

Clearly, this construction represents a secure deletion scheme, which can be proved secure assuming a secret-key cryptosystem along the lines of the existing proof [8]. The access cost of the scheme as well as its deletion cost are $O(\log n)$ secret-key operations. The erasable memory contains only one key of the secret-key encryption scheme as the master key and the auxiliary state is of size $O(n)$.

3.7 Time-tree construction

Our model and the tree construction above make no assumption about the order in which attributes are deleted. However, one may sometimes exploit such restrictions. Consider a sequence of attributes $a_1, \dots, a_n$ modeling expiration time such that any a_i can only be deleted if $a_1, \dots, a_{i-1}$ have been deleted beforehand. This property gives way to build a more efficient scheme.

More precisely, consider a deletion policy graph consisting of attributes $a_1, \dots, a_n$, protection classes $p_1, \dots, p_n$, and edges (a_i, p_i) for $i \in [n]$, which represent an ordered sequence of time units from $[n]$. We want to support deletion only as ordered by time, and we want to support an efficient operation that deletes multiple subsequent units at once. To this end, we construct a balanced tree over the leaves $p_1, \dots, p_n$, design a master key of size $O(\log n)$, but use no auxiliary storage. The leaves of the tree represent the time units in a left-to-right order. The key of a node is computed from its parent's key using a pseudo-random function (PRF), where the parent's key serves as the PRF key and the input denotes whether the node is a left or right child. The root node's key is chosen uniformly at random. The master key consists of the key of the leaf corresponding to the earliest non-deleted time unit and the keys of all right siblings on the path from the root to that leaf. This way, only the keys for time units following the last deleted one can be derived. Storing only right-siblings like this is a well-known technique [19] to achieve forward-secure cryptosystems without changing the public key of an encryption scheme.

The deletion for the attribute a_t updates the master key to contain the key for the leaf p_{t+1} and the keys of all right siblings on the path from the root to that leaf; keys for all time periods greater than t can be derived from the master key in time $O(\log n)$ from the stored right-siblings. Deleting an arbitrary attribute a_t and all attributes preceding it takes $O(\log n)$ cryptographic operations, though the amortized cost is only $O(1)$ if the attributes are deleted one by one. Hence, this tree construction is superior to the previous one both for deleting time units in order and for deleting in arbitrary increments.

3.8 Combined construction

Recall that for composing two secure deletion schemes with Theorem 2, only the lower scheme must be direct. Therefore, we can combine the tree construction of Section 3.6 with the direct scheme of Section 3.5 to obtain a secure deletion scheme with some attributes organized in a tree, as described above, whereas the rest are organized using a direct scheme. This is useful for modeling a case where many attributes are values from a large set or interval, such as user identities or dates.

One can further combine two or more such tree constructions trivially through logical expressions if the attributes in the trees are all distinct. In this way, we can compose many separate tree schemes with a direct scheme to obtain an elaborate and practical secure deletion scheme. The prototype implementation described later follows this approach.

4 Properties

4.1 Efficiency of the schemes

In Table 1, we present a comparison of the direct graph and tree schemes described, respectively, in Section 3.5 and Section 3.6; as well as two obvious "trivial" constructions, discussed next. As mentioned earlier, the protection cost is the same for all schemes considered in this paper.

The first trivial scheme uses a separate key for each protection class. This results in a large master secret key but achieves fast deletion operations. The Ephemerizer [21], for instance, encrypts all files

Scheme	*Deletion cost*	*Access cost*	*Master-key size*
Trivial direct	$O(1)$	$O(1)$	$O(\|\mathcal{P}\|)$
Trivial tree	$O(\|\mathcal{P}\|)$	$O(1)$	$O(1)$
Direct graph	$O(1)$	$O(d \cdot \ell)$	$O(\|\mathcal{A}\|)$
Tree	$O(\log \|\mathcal{P}\|)$	$O(\log \|\mathcal{P}\|)$	$O(1)$
Time-tree*	$O(1)$	$O(\log \|\mathcal{P}\|)$	$O(\log \|\mathcal{P}\|)$

Table 1. Efficiency comparison. $|\mathcal{A}|$ and $|\mathcal{P}|$ denote the number of attributes and protection classes in the policy graph, d is the maximum in-degree of a node, and ℓ is the longest path in the graph.
* The time-tree supports only deletion going forward in time; its amortized deletion cost is $O(1)$ and $O(\log |\mathcal{P}|)$ in the worst case.

with a particular expiration time with the same time-specific key, and Vanish [13] encrypts every user-data object with an independent key. Likewise, the Data Node Encrypted Filesystem (DNEFS) [23] uses this approach for protecting every data node of the flash filesystem independently.

The second trivial scheme implements a tree of depth one: it uses a single encryption key as a master key and one encryption key for each protection class; the master key is used to encrypt the keys of the protection classes and the ciphertexts are stored in the auxiliary state. Deletion requires the re-encryption of all remaining protection-class keys with a new master key. This scheme appears, e.g., in the extension of DNEFS to an encrypted filesystem [23]. Note that removing this level of indirection, and using the master key to protect the files directly, would result in much worse deletion performance, as the number of files is typically much greater than the number of protection classes.

The operational cost of our direct policy-graph secure deletion scheme is determined by the parameters of its graph, namely, the maximum in-degree d of a node and the longest path ℓ. The main advantages of this scheme are its fast deletion operation coupled with its high expressibility. Unlike the other schemes, which consider only protection classes mapped one-to-one to attributes, the policy-graph scheme allows flexible policies formulated through logical expressions over attributes. In practice, d and ℓ will often be small numbers, though in $O(|\mathcal{A}|)$ and $O(|\mathcal{P}|)$, respectively.

4.2 Relation to secret-key encryption

Every secure deletion scheme is also a secret-key cryptosystem with security against chosen-plaintext attacks. We only sketch this relation here; adding the formal details is straightforward.

Recall that a secret-key cryptosystem $\mathcal{S}$ consists of three algorithms for key generation, encryption, and decryption, respectively. The following steps emulate $\mathcal{S}$ from a secure deletion scheme $\mathcal{E}$:

1. Let G_1 be the minimal policy graph with one attribute a, one protection class p, and one edge from a to p. For key generation in $\mathcal{S}$, run the initialization algorithm of $\mathcal{E}$ with G_1 and use its output as the secret key.
2. For encryption of a plaintext m with $\mathcal{S}$, invoke the protection algorithm of $\mathcal{E}$ on p and m, obtain a ciphertext c, and output (p, c).
3. For decrypting a ciphertext (p, c) of $\mathcal{S}$, invoke the access algorithm of $\mathcal{E}$, and output the response.

We claim that $\mathcal{S}$ is a secret-key cryptosystem with indistinguishable ciphertexts under chosen-plaintext attacks (IND-CPA security [18]). To see this, suppose $\mathcal{S}$ is not secure. Then we construct a simulator SIM that contradicts the security of $\mathcal{E}$ in experiment Secdel, by interacting with an adversary $\mathsf{A}_{\mathcal{S}}$ that breaks the security of $\mathcal{S}$.

The simulator executes the operations of $\mathcal{S}$ according the described emulation of $\mathcal{S}$ from $\mathcal{E}$; note that SIM does not call Delete. When $\mathsf{A}_{\mathcal{S}}$ outputs two plaintexts m_0 and m_1 such that one is to be encrypted as challenge, the simulator outputs p, m_0, and m_1. Then Secdel responds with a ciphertext c^* that contains a representation of m_b, where $b \in \{0, 1\}$. According to the emulation of $\mathcal{S}$, the simulator gives (p, c^*) to $\mathsf{A}_{\mathcal{S}}$. When $\mathsf{A}_{\mathcal{S}}$ outputs a bit $\hat{b}_{\mathcal{S}}$ as its guess for the challenge plaintext, then SIM queries its oracle for the deletion operation with attribute set $\{a\}$, ignores the master key that it receives, and outputs $\hat{b}_{\mathcal{S}}$. Note that SIM emulates the IND-CPA security experiment perfectly. By the assumption that $\mathsf{A}_{\mathcal{S}}$ breaks the indistinguishability of ciphertexts of $\mathcal{S}$, it follows that $\Pr[\mathsf{Secdel}_{\mathsf{SIM},\mathcal{E}}(\kappa) = 1] - \frac{1}{2}$ is not negligible. Hence, $\mathcal{E}$ is not a secure policy-based deletion scheme.

5 Prototype implementation

Here we describe a filesystem prototype of a policy-based secure deletion scheme according to Sections 2 and 3. The *secure-deletion filesystem (delfs)* is implemented as an extension to EncFS [10], a virtual cryptographic filesystem in Linux based on FUSE [12]. As a virtual filesystem, *delfs* does not handle space allocation itself, just like EncFS, but acts as a transparent protection layer and projects the stored directories and files with the same structure onto a lower-layer (physical) filesystem.

5.1 Overview

Data stored in *delfs* seamlessly benefits from secure deletion. Together with every instance of a *delfs*-mounted directory tree, the user specifies a deletion policy in the format described later. Every file maintained by *delfs* is associated with a protection class and protected accordingly. A file can be accessed as long as its protection class is accessible according to the deletion policy.

As shown in Figure 4, three directories are involved in operating a *delfs* filesystem. First, *secure_dir* contains the master key of *delfs* and must be backed by erasable memory. It should be accessible only to the process running *delfs* and protected against exposure to any other entity during operation. In practice, it may be a POSIX-compatible local filesystem that supports physical secure deletion and that is never backed up. The *delfs* prototype uses a small ext2-formatted partition on a magnetic disk that supports secure deletion through in-place updates and free-space filling [24]. Second, bulk storage is provided by *raw_dir*, which represents the non-erasable memory such as a NAS server or cloud storage; only protected versions of files and auxiliary data reside here. Finally, the user stores and accesses protected files through *mount_dir*.

When a *delfs* directory is mounted for the first time, an initialization file has to be provided as well, which specifies the initial deletion policy and its attributes. All files written to *mount_dir* benefit from secure deletion. Files can be securely deleted by running a dedicated *delfsctl* utility and specifying attributes to delete. This securely deletes files according to the deletion policy such that an adversary can no longer recover them later. The adversary might obtain the contents of *secure_dir* afterwards and see all data that has ever been written to *raw_dir*, but can no longer infer anything about the data in the deleted files.

5.2 Attributes and policy specification

In order not to burden the user with specifying a deletion policy graph that may contain hundreds or thousands of possible attributes, *delfs* implements only restricted deletion policies compared to the model of Section 2. We argue below that these are sufficient for practical purposes.

All attributes in *delfs* are partitioned into *attribute types* such that every attribute occurs in one type. Types represent the categories relevant for secure deletion, such as owners, projects, organizations,

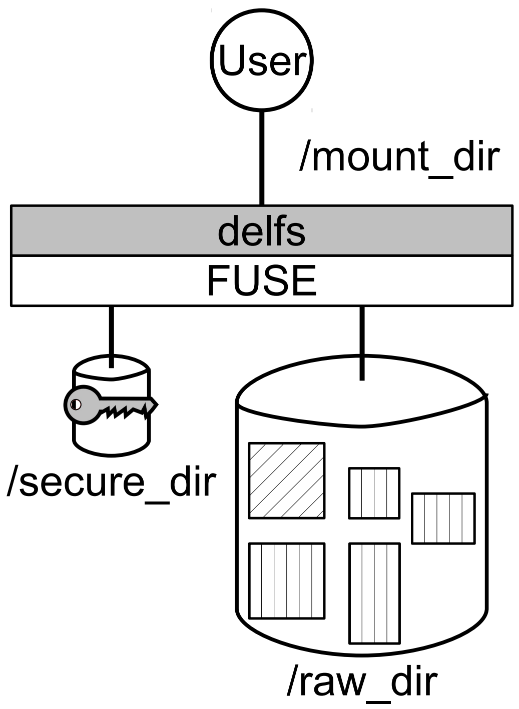

Figure 4. *delfs* architecture.

or expiration dates. For each attribute type, many different attribute values can be specified through enumeration of strings or by giving a range for numeric attributes. A range is expanded into an enumeration. A sample *delfs* attribute specification in the initialization file might look like this (written in `libConfig++` syntax):

```
types = (
    {
        name = "user";
        attributes = ["Alice", "Bob", "Charlie"];
        implementation = "simple";
    },
    {
        name = "project";
        attributes = ["X", "Y", "Z"];
        implementation = "simple";
    },
    {
        name = "expiration";
        attributes = ["2000", "2099"];
        specification = "range",
        implementation = "tree";
    }
);
```

Here, attribute type *expiration* is specified as a range, and this simply maps to an enumeration of the 100 different attributes. The meaning of *implementation* is explained in the next section. The attribute types in *delfs* model the familiar convention that objects have typed attributes and that every attribute can have only one value (for example, attributes of a POSIX file in the *stat* structure or attributes in a tuple of a relational database).

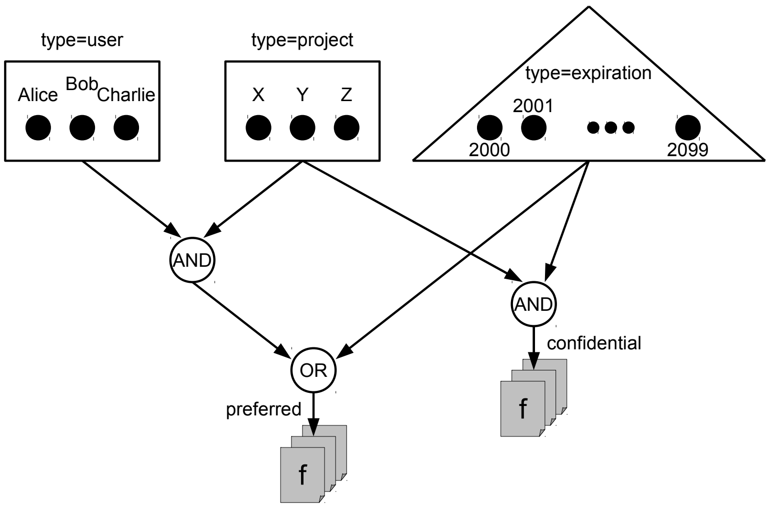

Figure 5. An illustration of the two *delfs* policies *preferred* and *confidential* as described in the text. The attribute type *expiration* uses the *tree* implementation, indicated by the triangle.

The deletion policy is given as a collection of *policies*, where each policy has a name and is represented by a logical expression over the attribute types, using AND and OR operators. By instantiating every attribute type with all of its possible values, each policy maps to many different multiple protection classes. Hence, the exact protection class is defined by the policy and the attribute values associated to a file.

Every policy in the configuration corresponds to a protection class according to our model. Below is a sample *delfs* deletion policy from the initialization file, as shown in Figure 5:

```
policies = (
    {
        name = "preferred";
        expr = "((user AND project) OR expiration)";
    },
    {
        name = "confidential";
        expr = "(expiration AND project)";
    }
);
```

When a file stored with policy *preferred* has attributes *user* $=$ *Bob*, *project* $= X$, and *expiration* $=$ 2014, this corresponds to protection class p_5 in the graph of Figure 1. Compared to the deletion policy graph in Section 2, *delfs* policies support only AND and OR gates. We discuss how attribute values are associated to files below.

5.3 Implementation

delfs implements the constructions of Section 3, with the exception of threshold gates and the time-tree scheme. The master key structure M is stored in *secure_dir*, whereas the auxiliary state S and all protected data reside in permanent memory under *raw_dir*. Every file stored in *delfs* is associated to one policy. If more flexibility is required, the policy graph should be extended.

A *delfs* attribute type with *simple* implementation corresponds to a direct secure deletion scheme with one protection class for every *delfs*-attribute value according to Section 3.3. For *delfs* attribute types with *tree* implementation, which are typically those with a large number of attribute values, the construction of Section 3.6 is invoked. These are further combined according to the *delfs* policies through the policy-graph construction in Section 3.5. The resulting scheme implements the combined model of Section 3.8.

When *delfs* is invoked for the first time on a particular mount point, the initial policy must be given. The current policy, some defaults, and auxiliary state are then stored in a system-wide state file in the root of *raw_dir*. Information specific to a file, such as its attribute values, the applicable policy, and the encrypted file-encryption key, are stored together with the file itself in its *extended attributes*.

Once the filesystem is mounted, every new file created inside *mount_dir* is protected according to the policy. The file takes its initial attribute values and policy from a special *defaults file*, located inside the directory where the file will reside; note that FUSE makes it possible to obtain the pathname when creating a file. If the defaults file is not present, parent directories in the path to *mount_dir* are searched; if no file defaults file is found, the system-wide initial policy stored in *raw_dir* is applied. An example *delfs* defaults file may contain:

```
attributes = ["project=X", "expiration=2013"];
policy = "confidential";
```

The user can perform regular filesystem operations on the files under *delfs* without affecting their deletion policy, as long as these operations leave the extended attributes intact.

Deletion-specific operations are done through a *delfsctl* administrative tool. In particular, (1) it performs secure deletion operations according to one or more given attributes and may update *secure_dir* during this operation; (2) it can reclaim space in *raw_dir* for files that have been deleted according to the policy and therefore have become inaccessible; and (3) it can manipulate the *delfs* policy and modify the attributes of existing files. However, the policy graph can only be extended. As this is, strictly speaking, not captured in our cryptographic model, we note that by allowing only extensions, each subsequent policy contains also the previous ones; hence, the security analysis for the latest policy captures also the previous states.

When *delfs* starts, it reads the master key from a file in *secure_dir* and buffers it in the daemon during operation. For every secure deletion operation, *delfs* first updates the auxiliary state and writes it back to *raw_dir*; then it updates the master key and overwrites the file in *secure_dir* with the changed contents.

Note that the design so far has no provision for securely deleting a directory and all its subdirectories. This can be achieved as follows. A predefined attribute named *PARENT* exists and can be used to formulate policies. The attribute may occur directly in a deletion policy. It represents the presence, in the filesystem, of the parent directory of a file, such that deleting that directory through a filesystem command triggers a secure deletion operation for the attribute *PARENT* on the children of the directory. Its implementation uses the tree construction of Section 3.6 with the topology of the filesystem tree and metadata maintained in per-directory state files. All *delfs* operations that modify the directory tree may thus implicitly modify the auxiliary state.

5.4 Evaluation

The prototype uses AES-256 for secret-key encryption in the *delfs*-specific code and the "standard configuration" of EncFS with AES-192 encryption. For the policy described in Figure 5, *delfs* has a master key composed of 7 encryption keys. The auxiliary state stores one ciphertext per edge and two ciphertexts per interior node plus the auxiliary data for the expiration tree, i.e., 201 ciphertexts in total with a binary tree of depth 7; all ciphertexts are encryptions of 256-bit AES keys. Every read and write operation has an overhead of 8 AES operations for the *confidential* protection class and of 12 AES operations for the *preferred* protection class, whereby all paths leading to the protection class' node are evaluated in the straightforward manner. In principle, the key-derivation could be optimized only for the necessary nodes; however, *delfs* resorts to caching and evaluates the whole graph only once at mount time. Every delete operation merely removes entries from the master key for the direct policies, performs the re-encryptions (in the auxiliary storage) for each of the 7 levels of the policy tree, and updates the root key stored in the *delfs* master key. The master key file is overwritten with the master key from the cached representation to complete the deletion operation.

The actual time incurred for secure deletion mainly depends on the choice of storage medium for the master key. When using a small ext2-formatted partition, one should also fill the filesystem with arbitrary data, in order to overwrite any potentially freed blocks from the master-key file. Measuring the performance of reading and writing files protected with EncFS and *delfs* shows that *delfs* introduces only a negligible overhead, as the encryption and decryption of the payload dominates the read and write operations.

6 Conclusion

This paper introduces the novel cryptographic notion of a policy-based secure deletion scheme. It uses a policy expressed as a directed acyclic graph and provides operations for protecting data under protection classes, accessing data, and deleting data according to attributes. An implementation in the secret-key model is given that generalizes earlier work of Tang et al. [28], which was restricted in the power of its policies and was tied to particular properties of its underlying cryptographic mechanisms.

The approach has been validated through a prototype implementation of policy-based secure deletion in the form of a virtual filesystem layer. It introduces a pragmatic approach to expressing typical attributes from practical systems through through defining a suitable policy graph.

Future work will focus on strengthening the security model by removing the requirement for the adversary to declare all attributes to be deleted in advance. Another promising extension consists in modeling and building secure deletion schemes with dynamic policies that support addition as well as removal of attributes and protection classes.

Acknowledgments

We thank Robert Haas, Alexis Hafner, and Anıl Kurmuş for interesting suggestions and discussions about this topic.

This work has been supported in part by the European Union's Seventh Framework Programme (FP7/2007–2013), grant number ICT-257243 TCLOUDS.

References

[1] Mikhail J. Atallah, Marina Blanton, Nelly Fazio, and Keith B. Frikken. Dynamic and efficient key management for access hierarchies. *ACM Transactions on Information and System Security*, 12(3), 2009.

[2] Mikhail J. Atallah, Marina Blanton, and Keith B. Frikken. Incorporating temporal capabilities in existing key management schemes. In *Proc. 12th European Symposium On Research In Computer Security (ESORICS)*, volume 4734 of *Lecture Notes in Computer Science*, pages 515–530. Springer, 2007.

[3] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *Proc. 28th IEEE Symposium on Security & Privacy*, pages 321–334, 2007.

[4] Dan Boneh and Richard Lipton. A revocable backup system. In *Proc. 6th USENIX Security Symposium*, 1996.

[5] Jason Crampton. Practical and efficient cryptographic enforcement of interval-based access control policies. *ACM Transactions on Information and System Security*, 14(1), 2011.

[6] Jason Crampton, Keith M. Martin, and Peter R. Wild. On key assignment for hierarchical access control. In *Proc. 19th IEEE Computer Security Foundations Symposium (CSF)*, pages 98–111, 2006.

[7] Alfredo De Santis, Anna Lisa Ferrara, and Barbara Masucci. Efficient provably-secure hierarchical key assignment schemes. *Theoretical Computer Science*, 412:5684–5699, 2011.

[8] Giovanni Di Crescenzo, Niels Ferguson, Russell Impagliazzo, and Markus Jakobsson. How to forget a secret. In Christoph Meinel and Sophie Tison, editors, *Proc. 16th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1563 of *Lecture Notes in Computer Science*, pages 500–509. Springer, 1999.

[9] Electronic Frontier Foundation. Surveillance self-defense project. `https://ssd.eff.org/`, 2013.

[10] EncFS. EncFS encrypted filesystem. `http://www.arg0.net/encfs`.

[11] European Parliament and Council. Protection of individuals with regard to the processing of personal data and on the free movement of such data. Directive 95/46/EC, 1995.

[12] FUSE. Filesystem in userspace. `http://fuse.sourceforge.net/`.

[13] Roxana Geambasu, Tadayoshi Kohno, Amit A. Levy, and Henry M. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proc. 18th USENIX Security Symposium*, 2009.

[14] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proc. 13th ACM Conference on Computer and Communications Security (CCS)*, pages 89–98, 2006.

[15] Susan Hohenberger and Brent Waters. Attribute-based encryption with fast decryption. In *Public Key Cryptography*, pages 162–179, 2013.

[16] Nikolai Joukov, Harry Papaxenopoulos, and Erez Zadok. Secure deletion myths, issues, and solutions. In *Proc. 3rd International IEEE Security in Storage Workshop (SISW)*, 2005.

[17] Nikolai Joukov and Erez Zadok. Adding secure deletion to your favorite file system. In *Proc. 3rd International IEEE Security in Storage Workshop (SISW)*, 2005.

[18] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman & Hall/CRC, 2007.

[19] Hugo Krawczyk. Perfect forward secrecy. In *Encyclopedia of Cryptography and Security*. 2005.

[20] Soumyadeb Mitra and Marianne Winslett. Secure deletion from inverted indexes on compliance storage. In *Proc. Workshop on Storage Security and Survivability (StorageSS)*, pages 67–72, 2006.

[21] Radia Perlman. File system design with assured delete. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2007.

[22] Zachary N. J. Peterson, Randal Burns, Joe Herring, Adam Stubblefield, and Aviel D. Rubin. Secure deletion for a versioning file system. In *Proc. 4th USENIX Conference on File and Storage Technologies (FAST)*, pages 143–154, 2005.

[23] Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proc. 21st USENIX Security Symposium*, 2012.

[24] Joel Reardon, Srdjan Capkun, and David Basin. SoK: Secure data deletion. In *Proc. 34th IEEE Symposium on Security & Privacy*, 2013.

[25] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *Proc. EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 457–473. Springer, 2005.

[26] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[27] Victor Shoup. Sequences of games: A tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. `http://eprint.iacr.org/`.

[28] Yang Tang, Patrick P. C. Lee, John C. S. Lui, and Radia Perlman. FADE: Secure overlay cloud storage with file assured deletion. In *Proc. Securecomm*, 2010.

[29] Michael Wei, Laura M. Grupp, Frederick E. Spada, and Steven Swanson. Reliably erasing data from flash-based solid state drives. In *Proc. 9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.