Hindering Data Theft with Encrypted Data Trees

Jorge Blasco[a,*], Juan E. Tapiador[a], Pedro Peris-Lopez[a], Guillermo Suarez-Tangil[a]

[a] *COSEC - Computer Security Lab*
Department of Computer Science, Universidad Carlos III de Madrid
Avda. Universidad 30, 28911, Leganes (Madrid), Spain

Abstract

Data theft is a major threat for modern organizations with potentially large economic consequences. Although these attacks may well originate outside an organization's information systems, the attacker—or else an insider—must eventually make contact with the system where the information resides and extract it. In this work, we propose a scheme that hinders unauthorized data extraction by modifying the basic file system primitives used to access files. Intuitively, our proposal emulates the chains used to protect valuable items in certain clothing shopping centers, where shoplifting is prevented by forcing the thief to steal the whole rack of items. We achieve this by encrypting sensitive files using nonces (i.e., pseudorandom numbers used only once) as keys. Such nonces are available, also in encrypted form, in other objects of the file system. The system globally resembles a distributed Merkle hash tree, in such a way that getting access to a file requires previous access to a number of other files. This forces any potential attacker to extract not only the targeted sensitive information, but also all the files chained to it that are necessary to compute the associated key. Furthermore, our scheme incorporates a probabilistic rekeying mechanism to limit the damage that might be caused by patient extractors. We report experimental results measuring the time overhead introduced by our proposal and compare it with the effort an attacker would need to successfully extract information from the system. Our results show that the scheme increases substantially the effort required by an insider, while the introduced overhead is feasible for standard computing platforms.

Keywords: Data Leakage Prevention, Insiders, Information Theft, File System Protection

1. Introduction

Insider threats are an increasing concern for organizations. One major threat posed by dishonest employees is the theft of sensitive information, which is considered as one of the most economically damaging risks since it usually results

*Corresponding author
Email addresses: jbalis@inf.uc3m.es (Jorge Blasco), jestevez@inf.uc3m.es (Juan E. Tapiador), pperis@inf.uc3m.es (Pedro Peris-Lopez), guillermo.suarez.tangil@uc3m.es (Guillermo Suarez-Tangil)

in loss of competitiveness, economic fees imposed by governments, and loss of reputation [1, 2]. Economic losses produced by data theft related events have been extensively addressed in several reports and surveys. One of such studies, conducted over banking and financial institutions in the US, states that 91% of the organizations victims of data theft experienced financial losses exceeding half a million USD in 30% of the cases [3].

An insider can be defined as a user with legitimate access to the organization's information systems [4]. Unlike attackers from outside the organization, malicious insiders usually possess valuable information about the network infrastructure, including the way data is stored across it and, more significantly, its value [5]. Malicious insiders can interfere with the availability of the organization services or can put at risk the confidentiality of sensitive information such as personal data or intellectual property. In fact, as stated in [5], the sense of entitlement to information makes an insider much more prone to stealing data. Previous research has shown that malicious insiders consider the risk of being caught and the difficulty of their actions while planning the attack [6]. Even when data leakage prevention mechanisms are in place, insiders can often rely on covert channels [7] that are not monitored as vehicles to extract valuable information. Since these channels usually have a very low bandwidth [8], increasing the required amount of information to transfer can render infeasible the entire data extraction process. Thus, if the time required to extract information is high enough, the insider may reconsider his attack, or even get caught in the process.

In this paper, we address this problem by proposing an encrypted file system where keys resemble a Merkle tree structure. Instead of using a DRM (Digital Rights Management) approach, which requires only the extraction of the sensitive file and a very short (but secret) key, our system requires the insider to extract much larger amounts of information. More specifically, the key used to encrypt a file is a function of the other files' contents. Such files are available in the file system but may be also encrypted, in such a way that other files need to be accessed, and so on. Overall, this creates a dependency graph over the file system such that one node (file) is unreadable (undecryptable) unless other nodes have been read before. Thus, an attacker would need to extract not only the target file(s), but also all others needed to compute all keys involved in the decryption process. This will necessarily increase the time required for an attacker to successfully extract a piece of information and also the probability of detection. However, this mechanism alone does not suffice. An attacker who wants to extract a file may access all necessary files, compute the associated key, and then extract the encrypted file and the key. To thwart this strategy, we introduce a probabilistic mechanism that automatically re-encrypts a file while being read, which in turn will trigger a cascade of re-encryptions in dependent files. While this will force the attacker to start over again, a legitimate user attempting to read the file will have to do it too. As detailed later, adjusting this mechanism is essential to achieve an adequate trade-off between security and file system efficiency.

The rest of this paper is structured as follows. Section 2 provides and overview of previous work in systems and models to prevent data leakage. Section 3 gives a general overview of the proposed system, describes potential application scenarios, provides formal definitions and assumptions, and presents the adversary model. In Section 4, we describe a number of primitive file ac-

cess operations upon which the proposed system is built. A detailed complexity analysis of each operation is provided in Section 5, while Section 6 discusses the effort required by the attacker to successfully extract data and the trade-off between security and file system efficiency. Section 7 describes our prototype implementation, the experimental setting used to assess it, and the obtained results. Finally, Section 8 concludes the paper by summarizing our contributions and outlining future research directions.

2. Related Work

The difficulty of extracting sensitive information from an organization may vary depending on the technical controls implemented to avoid such attacks. In an organization without any technical measures to prevent it, data theft can be as simple as copying sensitive files into a portable storage device. If the usage of portable drives is restricted by any physical or technical means, information may still be extracted through the network using a number of standard channels. For example, an insider could upload locally available files to a cloud storage service, or attach them to an email, or just use a file transfer network protocol, to mention just a few. In order to avoid such leakages, organizations often implement prevention mechanisms that monitor devices and network protocols and restrict the way users may employ applications to extract data. We next describe some relevant proposals in this direction.

The MITRE Corporation developed a system, named ELICIT, to avoid data leakage and other insider threats [9]. ELICIT uses network-related events to detect employees who perform actions inside their privileges but outside their actual scope, therefore violating the need-to-know policy [10]. The system works by first translating user-generated network traffic into information-use events. These events describe operations at the document level, such as reading, writing, sending, printing, etc. The stream of information-use events is then combined with a social network based on contextual information. This provides a global picture upon which a number of suspicion indicators can be derived, mostly related to dubious transfers of information among employees and devices.

In the commercial arena, the term Data Leakage Protection (DLP) systems—also known as Data Loss Protection, or Data Leakage Prevention—has gained some momentum in the last few years [1, 11, 12]. At the architectural level, a DLP system is often seen as composed of various interconnected components (sensors, information discovery agents, content filtering agents, etc.) that are deployed in the organization infrastructure to avoid exfiltration of sensitive information. DLP systems first identify what information an organization holds, how sensitive it is, and where it is stored. Once properly configured, the system constantly monitors potential leakage vectors, such as removable devices or network connections, in order to detect the transmission of data that was previously identified as sensitive. In the vast majority of the cases, the detection engine is just a pattern-matching algorithm that analyzes data items placed on a set of predefined leakage vectors against a database built from sensitive information. Such a database may include simple keywords or more sophisticated structures, including data patterns or fingerprints [13, 14, 15]. In general, most DLP systems include a number of response actions that block transmissions identified as containing sensitive information. DLP systems are widely deployed in many organizations and prevent most data leakages, both accidental and malicious.

However, as in the case of related technologies such as Intrusion Detection Systems (IDS), the pattern-matching detection process suffers from a number of inherent limitations that facilitate evasion, allowing an insider to bypass them [16, 17].

Encrypted file systems (see, e.g., [18, 19]) constitute a related technology also intended to prevent data loss. In this case, however, the main purpose is not to avoid leakages emanating from malicious employees or compromised accounts, but from the (accidental or not) loss of devices such as laptops, smartphones, portable drives, etc. A variety of Digital Rights Management (DRM) systems rely on similar ideas to mitigate threats related to data theft [20, 21]. Many DRM solutions encrypt sensitive files using randomly generated passwords that are "securely" kept by the applications responsible of enforcing a correct usage [22]. The idea is simple: unauthorized extraction of (encrypted) files does not lead to a security problem provided that the associated cryptographic keys remain secret. Unfortunately, protecting the keys is often not that simple, as repeatedly demonstrated by a number of attacks on widely known commercial DRM systems (see, e.g., [23]). Furthermore, assuming that access to the encrypted file is available, one weakness of such systems is that the only information an insider needs to extract is the key, which is generally a very little amount of information.

A traditional approach to dealing with data leakage problems was formulated roughly 30 years ago in the so-called multilevel security (MLS) models. Such systems classify users and data objects into a number of security levels and provide clear access rules as to whether a data access (i.e., read or write) is allowed depending on the object level and the user clearance. For example, the well-known Bell-LaPadula (BLP) [24] model forbids read operations if the user clearance is lower than the object level, and write operations if the user clearance is higher than the object level. Overall, both rules provide a working environment where information flow is restricted. Anderson [25] discusses a number of practical complications associated with such systems, both in terms of implementation and operational issues. Nevertheless, MLS systems alone are not enough to prevent data leakage if the extraction is carried out by an insider with sufficient privileges to get access to the data.

The scheme presented in this paper bears some resemblances with previous work on encrypted file systems and DRM approaches. For example, Chang et al. proposed in [26] a layered watermarking system in which the key to access a given level is found in a watermark embedded at a previous level. Our system is more explicit and does not rely on any embedding: in order to decrypt a file, a potential attacker—and also a legitimate user—must previously access and decrypt a variable number of other files. Thus, in our approach keys need not to be protected. Instead, they are freely available in the file system, but conveniently distributed so as to force any read operation to incur additional readings. Coupled with an automatic rekeying mechanism, this effectively creates a number of dependency chains among files that hinders data extraction both by patient adversaries (i.e., those that choose to extract data over a large amount of time) and also by those having only a limited amount of bandwidth. Our scheme does not consider authentication or integrity issues related to objects in the file system. Both traditional and novel protocols (see, e.g., [27, 28]) can be adapted for this purpose.

3. Data Chaining Trees and Adversarial Model

In this section, we describe in detail our proposal. We first provide a general overview of our system along with some illustrative application scenarios. Next, we introduce a number of assumptions and definitions, including the adversarial model.

3.1. Overview

The main goal of the scheme proposed in this paper is not to prevent unauthorized data extraction by enforcing restrictions on how data is accessed, but to difficult it to an extent that the effort required by an attacker to do so is impractical. Intuitively, the core idea consists of logically linking together a number of objects (files) in the file system, in a way that access to one of them requires previous access to others. We achieve this by encrypting all files with keys that follow a structure similar to a Merkle tree [29]. Specifically, each object is encrypted using a nonce as a key. Nonces are random numbers that are used only once, and are widely used in cryptographic protocols to provide "freshness" to encrypted messages—i.e., to guarantee that two messages are never identical even if their contents are the same. The key used to encrypt an object is, in turn, encrypted and stored within a different object. The key used to encrypt the first key follows the same rule: it is randomly generated, encrypted, and stored somewhere else. Thus, access to the original file must be done in reverse order, recovering keys that are used to obtain other keys until the final object is decrypted. Rather than a linear chain, the process can be easily generalized to a tree-like structure, in such a way that each key is distributed among more than one object. We refer to such structures as Data Chaining Trees (DCT).

If the scheme described above is implemented using an appropriate mode of encryption, accessing each nonce (key) will require decrypting the entire data object where it is stored. An attacker interested in extracting just one object would need to extract all linked files and compute the associated keys offline. This will force the attacker to extract potentially much more information, which may increase the likelihood of detecting the attack. Alternatively, the attacker may opt for extracting just the encrypted file, compute the key by accessing all required files, and then extract it too. As described later in Section 4, this strategy is partially neutralized by an automatic rekeying mechanism that periodically generates new nonces, replace them in the appropriate objects and update the encrypted version of the object. If appropriately configured, this mechanism frustrates data extraction by forcing the attacker to start over again with the data extraction. However, this also imposes an overhead on the system performance, which makes the scheme rather unsuitable for system-wide deployment. We next describe a number of scenarios where the scheme could be successfully deployed.

3.2. Application Scenarios

DCTs are not designed to operate on a system-wide deployment, i.e., applied to the entire file system. Instead, their use should be limited to those parts of the data storage requiring higher protection against data leakage. For example, many companies organize their work effort around the notion of project, with several employees participating in a project and, generally, one employee working in more than one project simultaneously. The level of sensitivity often

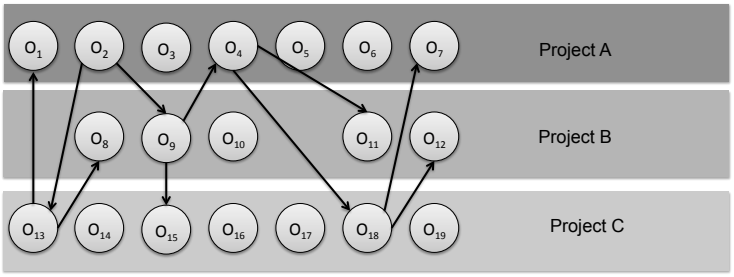

Figure 1: Project management scenario example

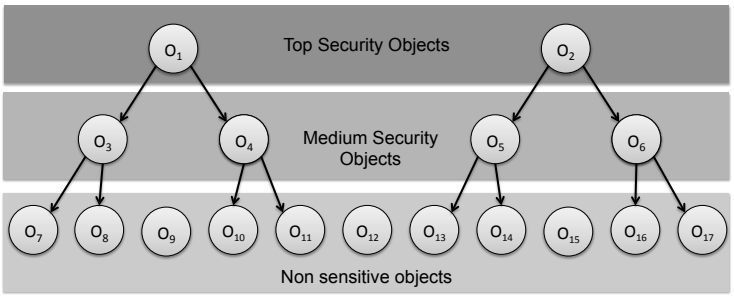

Figure 2: Multi Level Security Scenario Example

varies across projects, and therefore all data associated with a project (e.g., code, working documents, etc.) may require a level of protection different from files belonging to other projects. In some cases, some projects could just require no protection at all. In such scenarios, DCTs could be applied to each project separately, with a level of security proportional to the project sensitivity. Moreover, each data object from project should be linked to data objects belonging to a different project (see Fig. 1). Thus, an attacker who exfiltrates an entire project does not possess all required information to compute the decryption keys. Note that, in this case, not all files belonging to a project need to be encrypted under a DCT. Furthermore, while the DCT associated with some files may be small (i.e., few additional objects need to be accessed in advance to compute the key), more sensitive objects may be endorsed with larger DCTs. As described later in detail, this is a tuneable parameter that can be adjusted for each file depending on the protection requirements and the tolerable overhead in accessing it.

The notion of "level of protection" implicit in a DCT can be leveraged to organize a data repository into various hierarchical levels according to the impact associated with the loss of each file. Data with little to none impact at all can be left unprotected, while files belonging to higher levels are encrypted using a DCT with a level of protection proportional to the level. Such a hierarchy can be exploited to link together objects in different levels as graphically depicted in Fig. 2, with root nodes being highly sensitive objects and leaves requiring no protection at all.

In addition to the two scenarios described above, DCTs can be also used in a variety of environments where access to a file requires, as a form of implicit authentication, proof of possession of other data. For example, in a BYOD

(Bring Your Own Device) working environment users may be allowed to keep a local copy of project files in their laptops or tablet computers. However, such files may be linked through a DCT to a number of other files residing in the organization servers, which cannot be accessed from outside. This would effectively render such files unusable unless the employee is in the organization premises.

3.3. Definitions

In this section, we introduce a formal description of Data Chaining Trees and other supporting definitions.

Definition 1. A directed acyclic graph (DAG) is a non-cyclic graph $G = (V, ED)$, where $V = \{n_1, \ldots, n_k\}$ is the set of nodes and $ED \subseteq V \times V$ the set of directed edges between nodes. A directed edge is represented by a tuple $ED_i = \{n_{head}, n_{tail}\}$, where n_{head} is the origin node of the edge (head) and n_{tail} is the destination node (tail).

Definition 2. A directed tree (DT) is a DAG that would become a tree if the edge direction is ignored. For the purposes of this paper, all trees will be rooted. A $(w, l)-$DT is a DT where each node (except leaves) has exactly w children and the length of the path between the root and any leaf is upper bounded by l.

Definition 3. Let M_i be some data (e.g., an entire file, a single data block, etc.) that requires protection against data leakage, and let N_i be a randomly generated number (nonce) associated to M_i. A Data Chaining Tree $DCT = \{O_1, \ldots, O_n\}$ for M_i is a regular tree where each node O_j represents the encrypted version of a data element M_j (along with its corresponding nonce N_j) and directed edges specify the dependency relation between different data elements. For the purpose of this work, the amount of dependent files in each O_j is up to one, i.e., a node can only be the end of one edge in the whole graph. Fig. 3 shows a possible DCT configuration.

Definition 4. Let O_i and O_j be two DCT objects. O_j is said to be *required* by O_i if previous access to data in O_j is needed to access data in O_i. O_i is said to be *dependent* from O_j. The set of all required objects of O_i is named R_i and the set of all dependent objects is named D_i.

Definition 5. Let E_i be the encrypted version of data $M_i \parallel N_i$ belonging to node O_i, where $\parallel$ represents the concatenation of two pieces of data. Therefore, $O_i = \{R_i, D_i, E_i\}$.

Definition 6. If $|R_i| > 0$ the data in O_i will be encrypted using the nonces from its required objects. That is, $E_i = \texttt{Ciph}(k, M_i \parallel N_i)$, where k is a key derived from the concatenation of nonces belonging to required objects R_i and `Ciph` is an encryption operation in Propagating Cipher-Block Chaining (PCBC) mode. If $|R_i| = 0$ (i.e., it has no required objects), data will not be encrypted and $E_i = M_i \parallel N_i$.

Definition 7. The security level s of a DCT object O_i is the number of objects that must be accessed (i.e., decrypted) in order to compute the key needed to

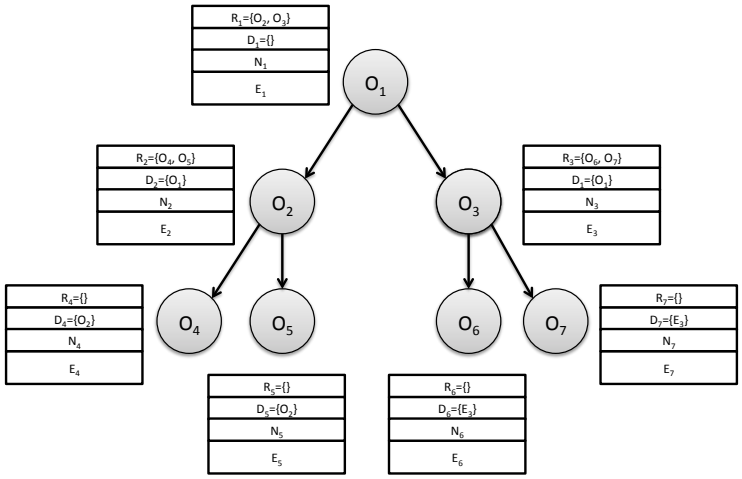

Figure 3: General Overview of Chaining Data System

decrypt the root object. In the case of regular trees, such a security level is given by

$$s = \frac{w^l - 1}{w - 1} - 1, \tag{1}$$

where w represents the number of children of each node and l the distance, measured in number of nodes, from O_i to a leaf of the tree (including O_i).

3.4. Adversarial Model

DCTs are effective to thwart attacks from adversaries that pursue the objectives described next and who possess the following capabilities:

- The adversary's goal is to extract a particular piece of sensitive information, not the maximum amount of data regardless of its value.
- High bandwidth channels that may allow the extraction of massive amounts of data outside the organization are controlled by some security mechanism, such as for example a DLP solution. However, an insider is left with the capability of extracting data using a number of covert channels existing in the infrastructure. Such channels are difficult to detect but have a low bandwidth.
- The adversary, using the appropriate applications, has access to the unencrypted version of all files allowed by his privileges.
- The adversary can read and write objects in the file system using the appropriate applications, but he cannot extract decrypted information from these applications. That is, the adversary cannot access the system memory where decrypted information resides. Obviously, a digital camera or other devices could be used to copy such information and extract it from the organization. However, we assume that this would only allow extraction of a limited amount of information.

Primitive	Description
$\texttt{PRF}(x)$	Pseudo-Random Function
$\texttt{Encrypt}(D, K)$	Encryption of D using key K
$\texttt{Decrypt}(E, K)$	Decryption of E using key K
$\texttt{KDF}(O_i)$	Key derivation function for object O_i
$\texttt{ReadNonce}(O_i)$	Read nonce from object O_i
$\texttt{ReadData}(O_i)$	Read data from object O_i
$\texttt{WriteData}(O_i, M_i)$	Writes data block M_i into object O_i
$\texttt{UpdateEncryption}(O_i, O_j^{'}, O_j)$	Updates encryption of object O_i
$\texttt{DeleteObject}(O_i)$	Removes object O_i
$\texttt{InitDCT}(O_i)$	Initializes a DCT rooted at O_i

Table 1: DCT primitives.

- The adversary has access to all raw data that is directly stored in the file system, including both sensitive objects encrypted with a DCT and files that do not require protection. Consequently, for each file the adversary knows the dependency relationships established by the DCT of each file and, therefore, can identify the files and order required to decrypt it.

4. DCT Primitives

One main advantage of DCTs is that they facilitate the protection of certain sensitive file system objects while being transparent to the final user. As nonces encrypted within other objects in the file system are used as keys, there is no need for the user to remember any secret key, nor even to be aware of the underlying encryption of some parts of the file system. This makes all accesses transparent to the end user, as the operating system will be in charge of managing DCTs and handle encryption and decryption operations. Thus, if a user requests read access to a file, the system must decrypt the data object and deliver it to the application used to access that file. Similarly, when a file is modified, it may be necessary to update the encryption of its dependent files, since the derived key will change as a consequence of the modifications.

Working with DCTs is carried out through the ten primitives shown in Table 1. In the remaining of this section we describe each one of them.

4.1. Encryption and Decryption

Data (M_i) contained in DCT objects (O_i) that require a minimum level of protection is stored encrypted (E_i). Our scheme does not assume a specific algorithm for this, but it is recommended that this operation is implemented by a fast encryption algorithm, such as for example a block cipher in an appropriate mode of operation. In particular, we encourage the use of Propagating Cipher-Block Chaining (PCBC) mode, as this mode of operation requires the previous block's plaintext and ciphertext to perform decryption of a given block. Thus, in order to decrypt the last block of the ciphertext, the rest of the ciphertext needs to be decrypted. Although this mode of operation is not commonly used, it has an interesting property that our scheme requires to work: it forces the insider to extract the entire encrypted object to be able to decrypt the nonces inside

E. Thus, we assume the existence of two cryptographically secure algorithms, $\texttt{Encrypt}(D, K)$ and $\texttt{Decrypt}(E, K)$, such that $\texttt{Decrypt}(\texttt{Encrypt}(D, K), K) = D$.

4.2. Key Derivation Function

The key used to encrypt all non-leaf objects in a DCT is derived from the nonces of their required objects (R). The Key Derivation Function, termed $\texttt{KDF}(O_i)$, combines all the required nonces N_i of R_i for O_i and produces a constant sized output. In order to generate sufficiently secure keys, $\texttt{KDF}(O_i)$ should hold some properties.

In this paper, we consider functions that take as input a string composed of the concatenation of all the nonces extracted from the required objects $R_i = O_j, \ldots, O_k$ for the object O_i. To do this, we use a Pseudo-Random Function $\texttt{PRF}()$ to derive the key, as specified in Algorithm 1. Note that the algorithm makes use of the primitive $\texttt{ReadNonce}()$, which will be described later.

Each time the data inside an object has to be encrypted or decrypted, this key has to be computed. Additionally, in order to avoid security risks, derived keys should be erased from memory after each usage. Although preserving keys in memory may reduce system overheads, it would also make easier for an attacker to access sensitive files.

Algorithm 1: $\texttt{KDF}(O_i)$

```
Data: An object O_i = {R_i, D_i, E_i}
Result: The key K_i used to encrypt/decrypt the object
K = "";
if |R_i| > 0 then
    foreach O_j in O_i.R_i do
        K = K || ReadNonce(O_j);
    end
    return PRF(K);
else
    return "";
end
```

4.3. Read Nonce

Nonces must be decrypted to derive the keys of their dependent object. If a nonce N_i is located in a non-leaf node of a DCT O_i, a series of $\texttt{ReadNonce}()$ operations must be recursively executed in order to obtain the nonces required to decrypt E_i and access the nonce N_i. This algorithm is described in Algorithm 2.

4.4. Read Data

A read operation returns the decrypted data M_i of an object O_i. If an object has no required objects (i.e., it is a leaf in the DCT), its data is not encrypted and therefore can be directly read. Nevertheless, if a file has a non-empty set of required objects, the nonces of those objects will have to be first read to derive the encryption key. Thus, the read operation recursively accesses all objects in

Algorithm 2: `ReadNonce`(O_i)

Data: An object $O_i = \{R_i, D_i, E_i\}$
Result: The unencrypted nonce N_i contained in O_i
if $|R_i| > 0$ **then**
  $K = \texttt{KDF}(O_i)$;
  $M_i \parallel N_i = \texttt{Decrypt}(E_i, K)$;
  return N_i
else
  $M_i \parallel N_i = E_i$;
  return N_i;
end

the DCT until reaching the leaves. Note that such accesses are implicit in the `Key`(O_i) call. The operation is described in Algorithm 3.

Algorithm 3: `ReadData`(O_i)

Data: An object $O_i = \{R_i, D_i, E_i\}$
Result: The unencrypted data M_i contained in O_i
if $|R_i| > 0$ **then**
  $K = \texttt{KDF}(O_i)$;
  $M_i \parallel N_i = \texttt{Decrypt}(E_i, K)$;
  return M_i
else
  $M_i \parallel N_i = E_i$;
  return M_i;
end

4.5. Write Data

Algorithm 4 describes the operations required to write a new piece of data into a data object. In this case, the new piece of data has to be encrypted with the key currently used for that object, which is obtained from all the nonces in its required objects.

4.6. Update Encryption

This procedure is called whenever a change in the DCT forces a change in the key used to encrypt an object. This event is triggered when the nonce of a required object is modified or when there is a change in the set of required objects of a node. Algorithm 5 describes this process.

4.7. DCT Initialization

This primitive initializes a DCT from a tree with plaintext data objects. The procedure operates recursively as described in Algorithm 6, first deriving the key from all required objects and then encrypting each DCT node. For a (w, d)-DCT (i.e., a regular tree with width w and depth d), the algorithm visits all the $\sum_{n=0}^{d-1} w^n$ objects, generates the appropriate keys (except for the leaves) and encrypts the contents.

Algorithm 4: `WriteData`(O_i, M_i)

Data: An object $O_i = \{R_i, D_i, E_i\}$, the data M_i' to write in the object
Result: The object with the new data $O_i' = \{R_i, D_i, E_i'\}$
if $|R_i| > 0$ **then**
  K = `KDF`(O_i);
  $M_i \parallel N_i =$ `Decrypt`(E_i, K);
  $E_i' =$ `Encrypt`$(M_i', K) \parallel N_i$;
  $O_i' = \{O_i.R_i, O_i.D_i, E_i'\}$;
  return O_i';
else
  $O_i' = \{O_i.R_i, O_i.D_i, M_i' \parallel N_i\}$;
  return O_i';
end

4.8. Delete Data

If a node belonging to a DCT is deleted from the file system, the DCT must be updated by using a new object that takes over the position of the one just deleted. One possibility is to use a free object in the file system, i.e., one that does not belong to an existing DCT. This operation is almost equivalent to a `WriteData`() call, but using the new object as replacement for the deleted one. In fact, as the nonce is not refreshed, there is no need to update the encryption of the parent node. The process is described in Algorithm 7.

4.9. Automatic Node Update

A node encryption update is only triggered when a nonce of a required object is modified. If a nonce is never refreshed, each node's encrypted data would remain constant over time, thus facilitating data extraction by an insider. This risk can be reduced by automatically triggering some encryption updates, for example by replacing a nonce in the DCT structure. Note that this will make useless some data objects already extracted by the insider, as they can no longer be used to derive the decryption key of a certain object. We elaborate on this later in Section 6 when discussing conditions to impede data extraction.

There are several possible changes to perform during an automatic update to thwart an ongoing extraction of sensitive information. A simple but effective solution consists of triggering an automatic node update with probability p every time a node is read. Such updates work by picking one child node and replacing all nodes in the branch, including the selected child (see Fig. 4). This is equivalent to a DCT initialization starting in the node that triggered the automatic node update with an additional update encryption operation on the parent node if it is not the root ($l < l_{max}$).

Assume an insider who pursues extracting the root node of a regular (w, d)-DCT. The attacker needs to extract all the $\frac{w^d-1}{w-1}$ nodes. However, if the automatic node update process is triggered and one branch of the DCT has to be replaced, say one hanging from a node at height $\hat{l} < d$, all the additional $\frac{w^{\hat{l}}-1}{w-1}$ nodes belonging to the new branch must be extracted too. For instance, in the example shown in Fig. 4, once the data in the node O_5 is read, the whole branch starting in O_5 is replaced by other branch. If the attacker uses a bottom-up

Algorithm 5: `UpdateEncryption`(O_i, N'_j, O_j)

Data: An object whose encryption needs to be updated $O_i = \{R_i, D_i, E_i\}$, the modified nonce N'_j and the old required object $O_j = \{R_j, D_j, E_j\}$
Result: The updated object $O'_i = \{R_i, D_i, E'_i\}$
if $|R_i| > 0$ **then**
  $K_E =$ "";
  $K_D =$ "";
  foreach O_k *in* $O_i.R_i$ **do**
    if $O_k = O_j$ **then**
      $K_E = K_E \parallel N'_j$;
    else
      $K_E = K_E \parallel$ `ReadNonce`(O_k);
    end
    $K_D = K_D \parallel$ `ReadNonce`(O_k);
  end
  $M_i \parallel N_i =$ `Decrypt`$(E_i,$ `PRF`$(K_D))$;
  $E'_i =$ `Encrypt`$(M_i \parallel N_i,$ `PRF`$(K_E))$;
  $O'_i = \{O_i.R_i, O_i.D_i, E'_i\}$;
  return O'_i;
else
  return O_i;
end

Algorithm 6: `InitDCT`(O_i)

Data: The root node (not initialized) of the DCT $O_i = \{R_i, D_i, M_i \parallel N_i\}$
Result: The initialized node O'_i
if $|R_i| > 0$ **then**
  $E_i =$ `Encrypt`$(M_i \parallel N_i,$ `KDF`$(O_i))$;
  $O'_i = \{R_i, D_i, E_i\}$;
  foreach O_j *in* $O_i.R_i$ **do**
    `InitDCT`(O_j);
  end
  return O'_i;
else
  return O_i;
end

extraction approach, the entire branch would be invalidated, which will make necessary to re-read more nodes than by following a top-down approach.

Automatic encryption updates introduce a non-negligible overhead in the system, which could negatively affect its usability. Fortunately, this can be conveniently adjusted by defining different values for the update probability p. Additionally, a significant cost reduction can be achieved by using precomputed branches, either from randomly picked files or by exchanging branches between two different DCTs.

Algorithm 7: `DeleteObject`(O_i)

Data: A node to be deleted $O_i = \{R_i, D_i, E_i\}$
Result: The node replacing the deleted node in the DCT, O_j
Get free data M_j;
Get new nonce N_j;
if $|R_i| > 0$ **then**
  $E_j = \texttt{Encrypt}(M_j \parallel N_j, \texttt{KDF}(O_i))$;
  $O_j = \{R_i, D_i, E_j\}$;
  return O_j;
else
  $O_j = \{R_i, D_i, M_j \parallel N_j\}$;
  return O_j;
end

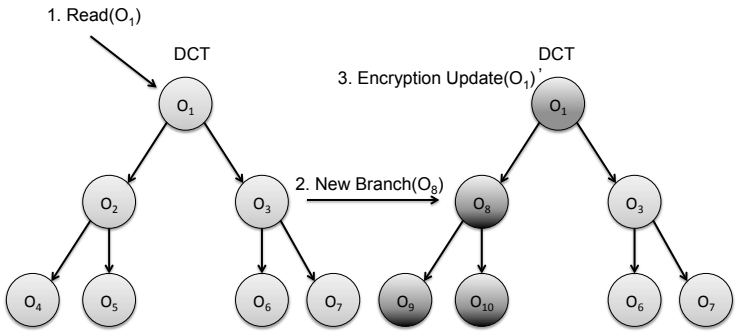

Figure 4: Automatic Node Update Example

4.10. Concurrent Operations

DCTs allow concurrent operations over the same file as in a standard file system for desktop computers, with some restrictions. In addition to the `InitDCT`, which is required to initialize DCT trees, other DCT primitives can be executed concurrently with no restriction provided that the node is not affected by any other operation being executed simultaneously. If two DCT primitives affect the same node, some considerations must be taken into account.

The key derivation function (`KDF`) governs all other DCT primitives, as it has to be executed to generate the key used to decrypt and encrypt objects. While this function is being executed for a node O_i, the nonces of all required objects R_i cannot be modified. For example, this may happen if some of those objects are being deleted (`DeleteObject`) at the same time that an automatic node update in an upper level of the DCT is triggered (`AutoUpdate`). When a node is deleted, it is replaced for another one, modifying one of the nonces used to generate the key. In order to avoid that, two strategies can be implemented. First, node removal for required nodes can be disabled while a key derivation is being carried out in the parent node. As an alternative, the file can be logically deleted (i.e., from the user's point of view), but not physically until the key derivation operation finishes. If an automatic update is triggered during the key derivation of a node, the operation must halt until the encryption of the

triggering node has been updated. In other words, automatic updates should have priority over any other primitive, as their goal is to hinder potential data extraction taking place. The key derivation function will be executed again starting in the node that triggered the encryption update.

Fortunately, other DCT primitives do not create conflicts if executed concurrently. If two `ReadData` operations are executed at the same time, in the worst case scenario two reads over the same raw data will be required. The same applies if a `WriteData` operation is performed on a node while a `ReadData` operation is being performed in a required node. Note that the write operation only changes the contents of the plaintext data, but does not alter the nonce of the required object. In fact, a nonce cache could be implemented to avoid duplicate readings of raw data when concurrently reading/writing nodes. On the other hand, during a `KDF` execution, nonces being read could be stored in this cache until the operation completes. Thus, if another `KDF` operation requires access to a nonce that is found in the cache, it will save a disk access operation and one decryption. However, we do not encourage the use of such caches since it introduces a potential weakness by facilitating access to decrypted nonces.

5. Complexity Analysis

In this section, we analyze the overhead introduced by using DCT primitives to access data objects. To do this, we present and discuss the computational complexity of the primitives introduced above. Empirical results will be presented later when discussing our experimental evaluation.

5.1. Encryption, Decryption, and Key Derivation

For simplicity in our analysis, we assume that both encryption and decryption operations have the same complexity. This is generally true for most existing stream and block ciphers. Furthermore, in order to keep the notation simple we denote by $C_{\texttt{Ciph}}$ the cost of an encryption or decryption operation. Note that such a cost varies (often linearly) depending on the size of the encrypted/decrypted data. We do not explicitly model this. Instead, our interest is in the number of cryptographic operations needed, even though the cost of such operations certainly depends on the input size. In this analysis we consider that the cost of executing $\texttt{PRF}(n)$ is negligible, as the size of nonces is assumed to be orders of magnitude smaller than the size of the data to be encrypted.

The key derivation operation requires access to the decrypted versions of all the children of the object that is accessed, which in turn causes one key derivation for all of them but the leaves of the DCT. Thus, the cost for an object at height l in a regular (w, d)-DCT is

$$\begin{aligned} C_{\texttt{KDF}}(w,l) &= \sum_{i=1}^{l-1} w^i \cdot C_{\texttt{Ciph}} \\ &= -C_{\texttt{Ciph}} + \sum_{i=0}^{l-1} w^i \cdot C_{\texttt{Ciph}} \\ &= -C_{\texttt{Ciph}} + C_{\texttt{Ciph}} \cdot \sum_{i=0}^{l-1} w^i \\ &= C_{\texttt{Ciph}} \left(\frac{w^l - 1}{w - 1} - 1 \right). \end{aligned} \tag{2}$$

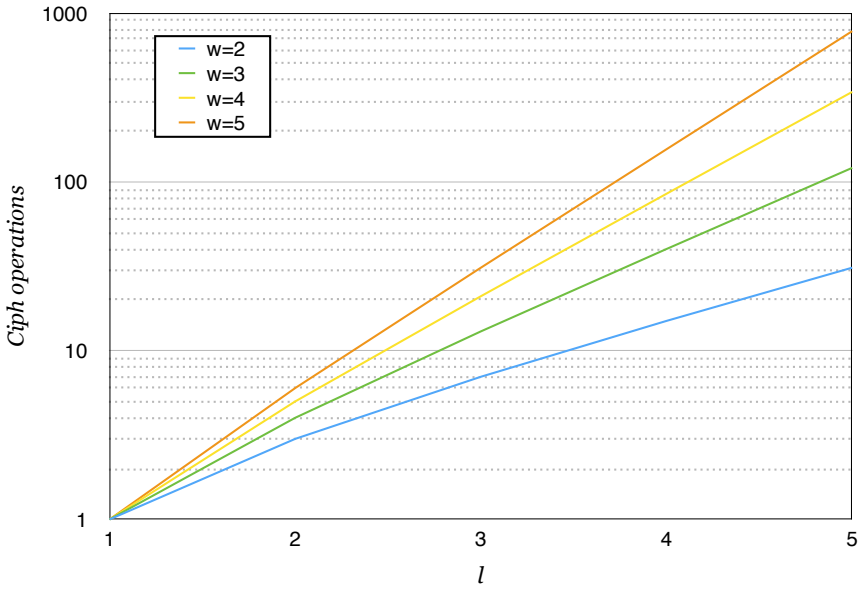

Figure 5: Cost of a `ReadData`() operation as a function of w and l

5.2. Read Data

The cost of reading the data contents M_i of an object O_i located at height l in a regular (w, l)-DCT is simply the cost of obtaining its key plus the cost of decrypting the object:

$$C_{\texttt{Read}}(w,l) = C_{\texttt{KDF}}(w,l) + C_{\texttt{Ciph}} = C_{\texttt{Ciph}}\frac{w^l - 1}{w - 1}. \tag{3}$$

As shown in Fig. 5, the computational complexity of a read operation is exponential in the DCT's height l, so a careful choice of the parameters (w, l) must be done. For example, for $w = 2$ and $l = 3$, a `ReadData`() operation requires 7 decryptions. This number increases to 15 for $l = 4$. If $(3, 3)$-DCTs or $(4, 3)$-DCTs were to be used, the number of required operations would be 13 and 21, respectively.

Reading a nonce takes exactly the same procedure as reading the object's data, but keeping the nonce instead of the data. This implies $C_{\texttt{Read}}(w,l) = C_{\texttt{ReadNonce}}(w,l)$. In the remaining of this paper, we will represent by $C_{\texttt{Read}}(w,l)$ the cost of reading a nonce.

5.3. Write Data

Writing a new piece of data M_i' first requires a read operation to extract the nonce N_i, which remains the same. This operation can also be used to extract the key needed to encrypt the new plaintext data. An additional encryption is required to encrypt back the new data with the old nonce appended. In the case of leaves (i.e, $l = 0$), writing data just requires to replace M_i with the new M_i',

which has no cost in terms of encryptions. Therefore, the cost of a `WriteData`() operation is

$$C_{\texttt{Write}}(w,l) = \begin{cases} 0 & l = 0 \\ C_{\texttt{Read}}(w,l) + C_{\texttt{Ciph}} & l > 0 \end{cases} \tag{4}$$

5.4. Update Encryption

An encryption update is triggered whenever a nonce in a required object is modified. This is, modifying the nonce of an object at level l triggers an encryption update at level $l+1$. Updating the encryption of a node requires a `ReadData`() operation to obtain the decrypted version of that data object, plus an encryption with the newly generated key. However, as shown in Algorithm 5, the nonces obtained during the decryption process can be used for the encryption process, not requiring another access to obtain the encryption nonces as the new nonce is already provided. Therefore, an encryption update has the same cost as a read operation in the node to be updated, $C_{\texttt{Read}}(w,l)$, plus the cost of encrypting again the contents with the new key, $C_{\texttt{Ciph}}$. Note that this is exactly the cost of a write operation when $l > 0$:

$$C_{\texttt{UpdateEnc}}(w,l) = C_{\texttt{Write}}(w,l) = C_{\texttt{Read}}(w,l) + C_{\texttt{Ciph}}. \tag{5}$$

5.5. DCT Initialization

When initializing a DCT, all data objects have their contents unencrypted. Initializing them requires the same effort as writing a data object. However, instead of performing decryptions to obtain the keys needed to decrypt each nonce before re-encryption, the system just encrypts each file. Therefore, the cost in a tree with $l > 0$ is given by

$$C_{\texttt{InitDCT}}(w,l) = C_{\texttt{Write}}(w,l) = C_{\texttt{Read}}(w,l) + C_{\texttt{Ciph}}. \tag{6}$$

5.6. Delete Data

Deletion involves the same operations as a `WriteData`() call, so

$$C_{\texttt{Delete}}(w,l) = C_{\texttt{Write}}(w,l). \tag{7}$$

5.7. Automatic Node Update

When an automatic node update is triggered in a node at height l, two processes must be executed: a new branch initialization at height $l-1$ and an encryption update for the parent node. On the one hand, the branch initialization has cost $C_{\texttt{InitDCT}}(w,l-1) = C_{\texttt{Read}}(w,l-1) + C_{\texttt{Ciph}}$. On the other hand, the encryption update on the parent node of the new branch has a cost $C_{\texttt{UpdateEnc}}(w,l) = C_{\texttt{Read}}(w,l) + C_{\texttt{Ciph}}$. Consequently

$$C_{\texttt{AutoUpdate}}(w,l) = C_{\texttt{Read}}(w,l-1) + C_{\texttt{Read}}(w,l) + 2 \cdot C_{\texttt{Ciph}}. \tag{8}$$

Recall, however, that if the new branch is precomputed (for example, when the system is idle), only the encryption update cost would be required. In that case, $C_{\texttt{AutoUpdate}}(w,l) = C_{\texttt{Read}}(w,l) + C_{\texttt{Ciph}}$.

(w,l)-DCT primitive	No. Cryptographic Operations
KDF	$C_{\texttt{KDF}}(w,l) = C_{\texttt{Ciph}}\left(\frac{w^l-1}{w-1}-1\right)$
ReadData	$C_{\texttt{Read}}(w,l) = C_{\texttt{Ciph}}\frac{w^l-1}{w-1}$
WriteData	$C_{\texttt{Write}}(w,l) = C_{\texttt{Ciph}}\left(\frac{w^l-1}{w-1}+1\right)$
UpdateEncryption	$C_{\texttt{UpdateEnc}}(w,l) = C_{\texttt{Ciph}}\left(\frac{w^l-1}{w-1}+1\right)$
DeleteObject	$C_{\texttt{Delete}}(w,l) = C_{\texttt{Ciph}}\left(\frac{w^l-1}{w-1}+1\right)$
InitDCT	$C_{\texttt{InitDCT}}(w,l) = C_{\texttt{Ciph}}\left(\frac{w^l-1}{w-1}+1\right)$
AutoUpdate	$C_{\texttt{AutoUpdate}}(w,l) = C_{\texttt{Ciph}}\left(\frac{w^l+w^{l-1}-2}{w-1}+2\right)$

Table 2: Complexity of DCT primitives.

5.8. Summary

Table 2 summarizes the complexity of all the DCT primitives discussed above. In general, all operations carried out over a DCT object require a $O(w^l)$ number of encryption/decryption operations. As we will discuss in the next section, this puts some limits on the values for w and l that would make efficient the access to files while significantly hindering the task of an insider interested in extracting data.

6. Hindering Data Extraction

The time required by an attacker to successfully extract an object depends on the number of decryptions to be done until obtaining the associated key. As discussed above, automatic node updates mitigate the risk of a patient insider who first extract the encrypted file and then computes, and also extracts, the key required to access it. Thus, random rekeyings will force the attacker to start over again the key derivation process, which in practice will translate into more files that need to be exfiltrated. We next analyze how different values of the probability p of automatic node update relate to the additional effort required by the adversary, but also on the system efficiency as perceived by a legitimate user.

As described in Section 3.4, the goal of the attacker is to extract some objects from the file system. The insider's only capability consists of reading raw files from disk and exfiltrating them by some means. Therefore, all information he is able to read is encrypted (decrypted data is only stored in memory), except for the case of DCT leaves. The DCT structure forces the insider to extract the decrypted versions of the child objects in order to compute the key used to decrypt the file he wants to extract. Thus, for instance, to extract a file that is at height $l = 1$, the insider will have to first extract all the nonces stored in the leaves connected to that object, then extract the encrypted object, and finally decrypt it. However, if the insider wants to read a file that is at height $l = 2$, he

has two possible strategies. On the one hand, he can first obtain the encrypted version of the children ($l = 1$) and then try to obtain the nonces that allow him to decrypt those nodes. If the automatic node encryption is triggered during this process, he will only have to read again that node and his parent. Note, however, that when reading that node again (and his parent), an additional node update can be triggered too.

On the other hand, the insider can use a bottom-up approach, firstly obtaining the leaves of the DCT ($l = 0$). In this scenario, if a node update is triggered while reading one particular node, the insider will have to read again the whole branch starting at the node that triggered the automatic node update. Thus, to obtain a successful read the attacker must extract all the nodes required to decrypt the node he wants to access before the triggered encryption update ends. In Section 4.9 we measured the amount of nodes an attacker has to extract in order to successfully decrypt a node. Taking into account the cost of extracting one node $C_{\texttt{OneExt}}$ (available bandwidth), the total cost required by the insider will be

$$C_{\texttt{Extract}}(w, l) = C_{\texttt{OneExt}} \frac{w^l - 1}{w - 1}. \tag{9}$$

As the size of nonces will generally be orders of magnitude smaller than the size of data objects, we consider that the cost of extracting the nonces in the leaves ($l = 0$) of the DCT is negligible compared to the cost of extracting an encrypted data node. Moreover, as described in Section 5.7, the cost required to calculate the new branch and update the parent node is

$$C_{\texttt{AutoUpdate}}(w, l) = 2 \cdot C_{\texttt{Ciph}} \left(\frac{w^l + w^{l-1} - 2}{w - 1} + 2 \right), \tag{10}$$

so from the defender's perspective, the automatic node update will succeed if $C_{\texttt{Extract}}(w, l) > C_{\texttt{AutoUpdate}}(w, l)$. Expanding the costs above we get

$$\begin{aligned} C_{\texttt{Extract}}(w, l) &> C_{\texttt{AutoUpdate}}(w, l) \\ C_{\texttt{OneExt}} \tfrac{w^l - 1}{w - 1} &> 2 \cdot C_{\texttt{Ciph}} \left(\frac{w^l + w^{l-1} - 2}{w - 1} + 2 \right) \\ C_{\texttt{Ciph}} &< C_{\texttt{OneExt}} \frac{w^l - 1}{(w^l + w^{l-1} + 2w - 4)} \\ C_{\texttt{Ciph}} + C_{\texttt{PRF}} &< \rho_{w,l} C_{\texttt{OneExt}}. \end{aligned} \tag{11}$$

The ratio

$$\rho_{w,l} = \frac{w^l - 1}{(w^l + w^{l-1} + 2w - 4)} \tag{12}$$

determines the conditions under which the reading of a node is useful for the attacker if a node update is triggered while that node is being read, *provided that no other node update is triggered in subsequent node extractions.* Therefore, if the time taken to encrypt/decrypt is less than $\rho_{w,l}$ times the time required to extract an object, the adversary will fail in its attempt and will be forced to extract again that node. Table 3 shows the required ratio for different values of (w, l). For instance, in a $(2, 3)$-DCT an insider will not be able to successfully extract an object from the DCT if the time required to encrypt the branch is less than 1.71 ($1/\rho_{w,l}$) times the time required to extract an object. Note how

$\rho_{w,l}$	$l = 2$	$l = 3$	$l = 4$	$l = 5$
$w = 2$	0.5	0.583	0.625	0.646
$w = 3$	0.571	0.684	0.727	0.742
$w = 4$	0.625	0.75	0.787	0.797
$w = 5$	0.666	0.794	0.825	0.831

Table 3: Ratio between the time required by cryptographic operations and time of extraction required for the automatic encryption update to succeed.

such a ratio increases as DCTs becomes larger. In fact, $\rho_{w,l}$ tends to 1.0 when increasing w and l.

In practical terms, the cost $C_{\texttt{OneExt}}$ will be generally upper bounded by the available network bandwidth. Such information, together with the costs discussed in Section 7.2, will guide the security administrator in choosing appropriate parameters for each DCT.

7. Empirical Evaluation

We next report and discuss experimental results obtained with a prototype implementation of the DCT scheme presented above. One major goal of our evaluation is to assess both the effort required by the attacker to succeed in extracting sensitive data and the practical overhead imposed over legitimate data access operations.

7.1. Implementation and Experimental Setup

Experiments have been carried out with a Java implementation of the DCT scheme described in this paper. Our implementation acts as a middleware that intercepts data access operations and translates them into the appropriate DCT primitives. Thus, the DCT middleware runs on top of a current file system implementation. Buffer sizes and other parameters are delegated to the file system layer. This greatly reduces the complexity of our implementation while making feasible to use such filtered accesses for specific applications. For example, it is straightforward to configure certain applications (e.g., office suites, software development environments, etc.) so that file access are only permitted through the DCT middleware. Additionally, this minimizes interactions with other applications already installed in the system. Our current DCT prototype uses the Bouncy Castle[1] implementation of AES-128 in PCBC mode. Although this is not the most efficient implementation, it provides us with a good upper bound to measure the overhead introduced during read and write operations.

The experiments were executed on a desktop computer with an Intel Core i5 processor at 2.7 GHz with 8 GB of RAM and a 5.400 rpm hard drive. The goal of these experiments is not to obtain a comparison with state-of-the-art implementations of current encrypted file systems, but to compare the efficiency of a DCT based file system against a file system with no encryption in the same machine. Therefore, our implementation has not been optimized for performance. As mentioned in Section 3.2, the DCT middleware is not intended for general

[1] Available at `http://bouncycastle.org`

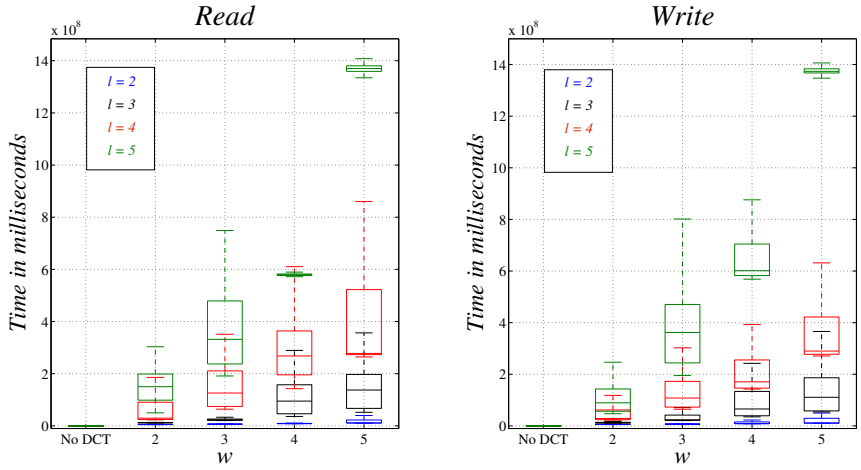

Figure 6: Time overhead per Mb of data required by `ReadData`() and `WriteData`() operations for different DCT configurations. The boxplots have been obtained with 100 executions.

purpose data access, but only for specific portions of the file system that require protection.

In our experiments, a DCT is built to protect files with different security levels. We consider the security level of a file to be the pair (w, l). In each experiment we consider scenarios where a file with security level (w, l) is read or written by a legitimate user or stolen by a malicious insider. For this, we create a large number of files with varying size and random contents for each experiment. File sizes vary from 1 Mb to 5 Mb, which are representative of many data objects (e.g., source code files, spreadsheets, documents, etc.) present in many working environments.

7.2. System Overhead

In our first experiment, we compare the time required to access and write files of 1 Mb using different DCT configurations, including as baseline case the time required without using a DCT. We have considered all combinations of security levels with $w \in \{2, 3, 4, 5\}$ and $l \in \{2, 3, 4, 5\}$. Each read and write operation has been executed 100 times. Fig. 6 shows the distribution of access times obtained for each DCT configuration with respect to the baseline case of not using any DCT.

The experimental results confirm that read and write operations take roughly the same time. Recall that, according to the formal analysis, the difference amounts to one additional encryption only, which takes negligible time. In the simplest DCT configuration ($w = 2$ and $l = 2$), read and write operations require 5 milliseconds (83 more times than a regular file system read and 48 more times than a regular file system write). These values grow exponentially with l. In fact, reading 1 Mb of data when $w = 5$ and $l = 5$ requires up to 1.37 seconds more than a regular read operation.

Applying high protection (e.g., $w = 5$ and $l = 5$) to each file in the system introduces a non-negligible system overhead that would result in a poor system

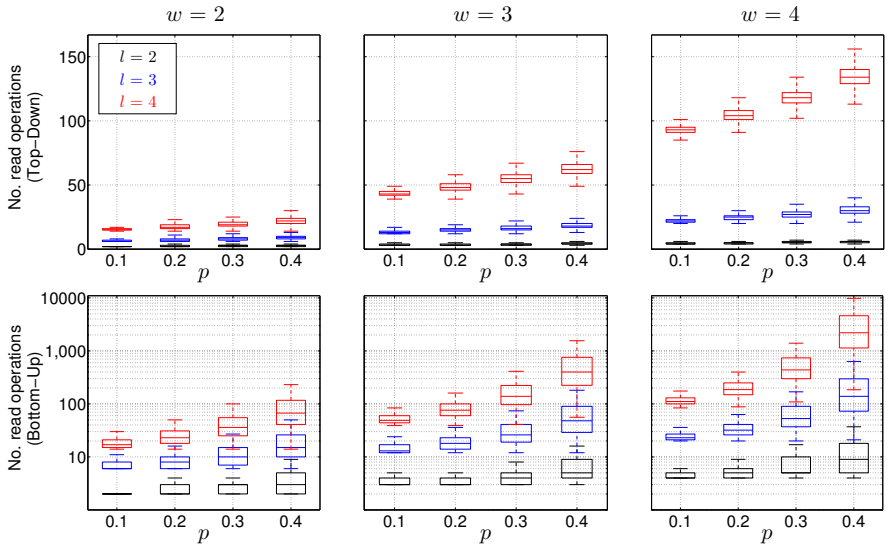

Figure 7: Number read operations required by an attacker (with both approaches) to extract a node with different configurations of $w = 2, 3, 4$, $l = 2, 3, 4$ and $p = 0.1, 0.2, 0.3, 0.4$.

performance for the final user. However, we believe these are reasonable for a reduced number of documents, particularly if they are not accessed constantly.

7.3. Adversary Overhead

In order to measure the effort required by an attacker to successfully obtain a piece of data, several simulations have been conducted. In these, we measure the amount of data objects the insider needs to extract from a DCT in order to successfully decrypt a data object with different values of w and l. Although legitimate operations always perform reads on a bottom-up approach, the attacker can extract files using either a bottom-up or a top-down approach. We have performed simulations using both approaches for the attacker.

In these experiments, we assume that the system is able to generate new branches before the insider extracts an entire branch as previously discussed. Each time an automatic update is triggered, the node being read and its parent must be read again. If the probability of triggering an encryption update is higher than 0.5, it may be infeasible for the attacker to extract the information, as the number of reads will tend to infinity. However, this setting also makes the system infeasible, as a legitimate read will also trigger an infinite number of encryption updates. Tests have been performed with all combinations of $w \in \{2, 3, 4\}$ and $l \in \{2, 3, 4, 5\}$ with probabilities ranging from $p = 0.1$ up to $p = 0.4$. Ten thousand executions were performed for each configuration, resulting in 48 different experiments. Figure 7 shows the boxplots obtained for each configuration of w, l, and p. As expected, the number of read operations increases if the attacker follows the bottom-up approach, which is the same used by legitimate reads. Note also that l has a much more significant effect on performance than w, as it appears in the exponent in the associated cost values.

In addition to the previous experiments, we have measured the average amount of time that an insider would require to successfully extract a piece

of sensitive information when the system is being used (Table 4). We consider a scenario in which DCTs are not being used to protect information and compared it with others where DCTs with different configurations of w, l and p are used. Furthermore, we consider different extraction speeds available for the insider, which will always act using a top-down approach (as it requires less read operations as shown in Fig. 7). First, we consider an scenario where an insider is able to extract information directly through a link with an average mobile Internet connection (i.e., 1 Mbps). Secondly, we consider an scenario in which workstations and devices belonging to the organization are controlled by a DLP solution [30]. Under this scenario, the insider must extract information using a covert channel, as other available channels are being monitored. We have restricted our experiments to two different covert channels that could be used in scenarios where DCTs may be useful. The first one uses the memory bus locking mechanism to create a covert channel inside a cloud environment [31]. In this covert channel, the authors achieved an average transmission rate of 107 bps with a very low error rate (0.75%). The second scenario considers a covert channel for Android platforms described in [32]. In particular, by using file locks of shared files, authors are able to covertly exchange information between two applications at 685 bps.

Table 4 summarizes the actual time that an attacker would require to extract files of 1 Mb and 5 Mb for the three extraction vectors described above. For example, an attacker attempting to extract a file protected with a $(4, 4)$-DCT and $p = 0.4$ would require, on average, 134 times more than the time required without DCTs. Thus, rather than the 2.12 hours needed to extract a 5 Mb file, a $(4, 4)$-DCT would force the attacker to spend almost 12 days to succeed. These times, together with the overheads shown in Fig. 7, could be used to determine a good balance between the level of protection for each group of files and the tolerable overhead.

8. Conclusions

In this paper, we have proposed a scheme to hinder illegal data extraction of sensitive information by malicious insiders. Our approach is based on linking together a number of data objects by cryptographic means, in a way that access to one of them requires previous access to others. Overall, this creates an encrypted file system where each file decryption key is actually distributed over other files and freely available. Thus, an insider trying to exfiltrate a particular piece of information will need to extract not only the target files, but also other files. The system is particularly useful against insiders that rely on uncontrolled leakage vectors, such as for example covert channels, to silently extract information.

Our proposal has been empirically evaluated with a prototype implementation that acts as a middleware sitting on top of the file system layer. We have analyzed the overheads introduced in legitimate read and write operations due to encryption operations. Overall, if encryption is faster than the extraction bandwidth available for the insider, the system can be used to protect a restricted number of sensitive files with a reasonable cost, while greatly hindering information extraction attacks. In practical terms, DCTs may introduce a non-negligible overhead in the system operation if too many sensitive objects must

		Extraction Speed		
	Block Size	**1 Mbps**	**685 bps[32]**	**107 bps[31]**
no DCT	1 Mb	1 sec	25.51 min	2.72 hours
	5 Mb	5 sec	2.12 hours	13.61 hours
$(2, 3, 0.1)$	1 Mb	6.55 sec	2.78 hours	17.87 hours
	5 Mb	32.78 sec	13.93 hours	3.71 days
$(2, 3, 0.4)$	1 Mb	8.7 sec	3.7 hours	23.71 hours
	5 Mb	43.55 sec	18.52 hours	4.93 days
$(2, 4, 0.1)$	1 Mb	15.41 sec	6.55 hours	1.78 days
	5 Mb	1.28 min	1.36 days	8.74 days
$(2, 4, 0.4)$	1 Mb	21.62 sec	9.19 hours	2.45 days
	5 Mb	1.8 min	1.91 days	12.26 days
$(3, 3, 0.1)$	1 Mb	13.19 sec	5.61 hours	1.49 days
	5 Mb	1.1 min	1.17 days	7.48 days
$(3, 3, 0.4)$	1 Mb	17.93 sec	7.62 hours	2.03 days
	5 Mb	1.49 min	1.58 days	10.17 days
$(3, 4, 0.1)$	1 Mb	43.17 sec	18.36 hours	4.89 days
	5 Mb	3.6 min	3.82 days	24.48 days
$(3, 4, 0.4)$	1 Mb	1.03 min	1.09 days	7.00 days
	5 Mb	5.14 min	5.46 days	35 days
$(4, 3, 0.1)$	1 Mb	22.06 sec	9.38 hours	2.5 days
	5 Mb	1.83 min	1.95 days	12.51 days
$(4, 3, 0.4)$	1 Mb	30.17 sec	12.83 hours	3.42 days
	5 Mb	2.51 min	2.67 days	17.11 days
$(4, 4, 0.1)$	1 Mb	1.55 min	1.65 days	10.57 days
	5 Mb	7.76 min	8.25 days	52.85 days
$(4, 4, 0.4)$	1 Mb	2.23 min	2.37 days	15.21 days
	5 Mb	11.18 min	11.88 days	76.06 days

Table 4: Average extraction time required by an insider with different (w, l, p) DCT configurations and extraction speeds using a top-down approach.

be protected or if the DCT parameters (w, l, p) are too high. Such a security-performance trade-off is inherent to DCTs. However, our current implementation admits a number of optimizations that would increase its performance in real-world deployments. For example, rather than sitting on top of the file system layer, it would be convenient to have an independent implementation that could access data block natively. This would translate into a considerable increase in performance.

Acknowledgements

We are very grateful to the anonymous reviewers for constructive feedback and insightful suggestions that helped to improve the quality of our original manuscript. This work was supported by the MINECO grant TIN2013-46469-R (SPINY: Security and Privacy in the Internet of You).

References

[1] Simon Liu and Rick Kuhn. Data loss prevention. *IT Professional*, 12(2):10–13, 2010.

[2] Ramona R Rantala. Cybercrime Against Businesses. Technical report, U.S. Department of Justice, July 2004.

[3] Marisa Reddy R., Michelle Keeney, Eileen Kowalsky, Dawn Cappelli, and Andrew Moore. Insider Threat Study: Illicit Cyber Activity in the Banking and Finance Sector. Technical Report 7, Carnegie Mellon Software Engineering Institute, January 2005.

[4] Shari Lawrence Pfleeger, Joel B. Predd, Jeffrey Hunker, and Carla Bulford. Insiders Behaving Badly: Addressing Bad Actors and Their Actions. *IEEE Transactions on Information Forensics and Security*, 5(1):169–179, 2010.

[5] Andrew P Moore, Dawn M Cappelli, Thomas C Caron, Eric Shaw, and Randall F Trzeciak. Insider Theft of Intellectual Property for Business Advantage: A Preliminary Model. In *1st International Workshop on Managing Insider Security Threats*, pages 1–22, West Lafayett, 2009.

[6] Dawn M Cappelli, Andrew P Moore, and Randall F Trzeciak. *The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud).* Addison-Wesley Professional, 2012.

[7] Yali Liu, Cherita Corbett, Ken Chiang, Rennie Archibald, Biswanath Mukherjee, and Dipak Ghosal. Sidd: A framework for detecting sensitive data exfiltration by an insider attack. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–10. IEEE, 2009.

[8] Annarita Giani, Vincent H Berk, and George V Cybenko. Data exfiltration and covert channels. In *Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense V. Edited by Carapezza, Edward M.. Proceedings of the SPIE*, volume 6201, page 620103, 2006.

[9] Deanna D Caputo, Greg Stephens, Brad Stephenson, Megan Cormier, and Minna Kim. An empirical approach to identify information misuse by insiders. In *Recent Advances in Intrusion Detection*, pages 402–403. Springer, 2008.

[10] Boanerges Aleman-Meza, Phillip Burns, Matthew Eavenson, Devanand Palaniswami, and Amit Sheth. An ontological approach to the document access problem of insider threat. In *Intelligence and Security Informatics*, pages 486–491. Springer, 2005.

[11] George Lawton. New technology prevents data leakage. *Computer*, 41(9):14–17, 2008.

[12] S. Charbonneau. The role of user-driven security in data loss prevention. *Computer Fraud & Security*, 2011(11):5 – 8, 2011.

[13] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[14] S. Schleimer, D.S. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 85–94. ACM, 2003.

[15] J.P. Kumar and P Govindarajulu. Duplicate and Near Duplicate Documents Detection: A Review. *European Journal of Scientific Research*, 32(4):514–527, 2009.

[16] Alejandro Hernández. Trend Micro Data Loss Prevention 5.2 Data Leakage Through Certain HTTP/HTTPS Channels. Technical report, 2010.

[17] Jorge Blasco, Julio Cesar Hernandez-Castro, Juan E. Tapiador, and Arturo Ribagorda. Bypassing Information Leakage Protection with Trusted Applications. *Computers & Security*, 31(4):557–568, June 2012.

[18] Matt Blaze. A cryptographic file system for unix. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 9–16. ACM, 1993.

[19] S. Lee, H.-R. Lee, S. Lee, and J. Kim. Drmfs: A file system layer for transparent access semantics of drm-protected contents. *Journal of Systems and Software*, 85(5):1058 – 1066, 2012.

[20] J.-S. Li, C.-J. Hsieh, and C.-Fu H. A novel {DRM} framework for peer-to-peer music content delivery. *Journal of Systems and Software*, 83(10):1689 – 1700, 2010.

[21] A. Lazouski, F. Martinelli, and P. Mori. Usage control in computer security: A survey. *Computer Science Review*, 4(2):81 – 99, 2010.

[22] Yang Yu and Tzi-cker Chiueh. Enterprise Digital Rights Management: Solutions against Information Theft by Insiders. Technical report, Department of Computer Science, Stony Brook University, 2004.

[23] Alapan Arnab and Andrew Hutchison. Digital rights management – an overview of current challenges and solutions. In *Proceedings of Information Security South Africa (ISSA) Conference 2004*. Citeseer, 2004.

[24] DE Bell and LJ LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report ESD-TR-73-278, Mitre Corporation, 1973.

[25] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2001.

[26] F. C. Chang, H. C. Huang, and H. M. Hang. Layered access control schemes on watermarked scalable media. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 49(3):443–455, 2007.

[27] Y. H. Huang, K. H. Fan, and W. S. Hsieh. Message authentication scheme for vehicular ad-hoc wireless networks without rsu. *Journal of Information Hiding and Multimedia Signal Processing*, 6(1):113–122, 2015.

[28] H. Zhu. Structured and efficient password-based group key agreement protocol. *Journal of Information Hiding and Multimedia Signal Processing*, 5(4):649–665, 2014.

[29] Ralph C Merkle. A certified digital signature. In *Advances in Cryptology-CRYPTO89 Proceedings*, pages 218–238. Springer, 1990.

[30] G Lawton. New Technology Prevents Data Leakage. *Computer*, 41:14–17, 2008.

[31] Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. *Networking, IEEE/ACM Transactions on*, PP(99):1–1, 2014.

[32] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.