On Consistency of Encrypted Files

Alina Oprea[1] and Michael K. Reiter[2]

[1] Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
alina@cs.cmu.edu
[2] Electrical & Computer Engineering Department, Computer Science Department, and CyLab, Carnegie Mellon University, Pittsburgh, PA, USA
reiter@cmu.edu

Abstract. In this paper we address the problem of consistency for cryptographic file systems. A cryptographic file system protects the users' data from the file server, which is possibly untrusted and might exhibit Byzantine behavior, by encrypting the data before sending it to the server. The consistency of the encrypted file objects that implement a cryptographic file system relies on the consistency of the two components used to implement them: the file storage protocol and the key distribution protocol.

We first define two generic classes of consistency conditions that extend and generalize existing consistency conditions. We then formally define consistency for encrypted file objects in a generic way: for any consistency conditions for the key and file objects belonging to one of the two classes of consistency conditions considered, we define a corresponding consistency condition for encrypted file objects. We finally provide, in our main result, necessary and sufficient conditions for the consistency of the key distribution and file storage protocols under which the encrypted storage is consistent. Our framework allows the composition of existing key distribution and file storage protocols to build consistent encrypted file objects and simplifies complex proofs for showing the consistency of encrypted storage.

1 Introduction

Consistency for a file system that supports data sharing specifies the semantics of multiple users accessing files simultaneously. Intuitively, the ideal model of consistency would respect the real-time ordering of file operations, i.e., a read would return the last written version of that file. This intuition is captured in the model of consistency known as linearizability [16], though in practice, such ideal consistency models can have high performance penalties. It is well known that there is a tradeoff between performance and consistency. As such, numerous consistency conditions weaker than linearizability, and that can be implemented more efficiently in various contexts, have been explored. Sequential consistency [19], causal consistency [4], PRAM consistency [22] and more recently, fork consistency [24], are several examples.

In this paper we address the problem of consistency for encrypted file objects used to implement a cryptographic file system. A cryptographic file system protects the users' data from the file server, which is possibly untrusted and might exhibit Byzantine behavior, by encrypting the data before sending it to the server. When a file can be shared, the decryption key must be made available to authorized readers, and similarly authorized writers of the file must be able to retrieve the encryption key or else create one of their own. In this sense, a key is an object that, like a file, is read and/or written in the course of implementing the abstraction of an encrypted file.

Thus, an *encrypted file object* is implemented through two main components: the *key object* that stores the encryption key, and the *file object* that stores (encrypted) file contents. We emphasize that the key and file objects may be implemented via completely different protocols and infrastructures. Our concern is the impact of the consistency of each on the encrypted file object that they are used to implement. The consistency of the file object is obviously essential to the consistency of the encrypted data retrieved. At the same time, the encryption key is used to protect the confidentiality of the data and to control access to the file. So, if consistency of the key object is violated, this could interfere with authorized users decrypting the data retrieved from the file object, or it might result in a stale key being used indefinitely, enabling revoked users to continue accessing the data. We thus argue that the consistency of both the key and file objects affects the consistency of the encrypted file object. Knowing the consistency of a key distribution and a file access protocol, our goal is to find necessary and sufficient conditions that ensure the consistency of the encrypted file that the key object and the file object are utilized to implement.

The problem that we consider is related to the *locality* problem. A consistency condition is *local* if a history of operations on multiple objects satisfies the consistency condition if the restriction of the history to each object does so. However, locality is a very restrictive condition and, to our knowledge, only very powerful consistency conditions, such as linearizability, satisfy it. In contrast, the combined history of key and file operations can satisfy weaker conditions and still yield a consistent encrypted file. We give a generic definition of consistency $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$ for an encrypted file object, starting from any consistency conditions C_1 and C_2 for the key and file objects that belong to one of the two classes of generic conditions we define. Intuitively, our consistency definition requires that the key and file operations seen by each client can be arranged such that they preserve C_1-consistency for the key object and C_2-consistency for the file object, and, in addition, the latest key versions are used to encrypt file contents. The requirement that the most recent key versions are used for encrypting new file contents is important for security, as usually the encryption key for a file is changed when some users are revoked access to the file. We allow the decryption of a file content read with a previous key version (not necessarily the most recent seen by the client), as this would not affect security. Thus, a system implementing our definition guarantees both *consistency* for file contents and *security* in the sense that revoked users are restricted access to the encrypted file object.

Rather than investigate consistency for a single implementation of an encrypted file, we consider a collection of implementations that are all *key-monotonic*. Intuitively, in a key-monotonic implementation, there exists a consistent ordering of file operations such that the written file contents are encrypted with monotonically increasing key versions. We formally define this property that depends on the consistency of the key and file objects. We prove in our main result (Theorem 1) that ensuring that an implementation is key-monotonic is a necessary and sufficient condition for obtaining consistency for the encrypted file object, given several restrictions on the consistency of the key and file objects. Our main result provides a framework to analyze the consistency of a given implementation of an encrypted file object: if the key object and file object satisfy consistency conditions C_1 and C_2, respectively, and the given implementation is key-monotonic with respect to C_1 and C_2, then the encrypted file object is $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$-consistent.

In this context, we summarize our contributions as follows:

- We define two generic classes of consistency conditions. The class of *orderable consistency conditions* includes and generalizes well-known conditions such as linearizability, causal consistency and PRAM consistency. The class of *forking consistency conditions* is particularly tailored to systems with untrusted shared storage and extends fork consistency [24] to other new, unexplored consistency conditions.
- We define consistency for encrypted files: for any consistency conditions C_1 and C_2 of the key and file objects that belong to these two classes, we define a corresponding consistency condition $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$ for encrypted files. To our knowledge, our paper is the first to rigorously formalize consistency conditions for encrypted files.
- Our main result provides necessary and sufficient conditions that enable an encrypted file to satisfy our definition of consistency. Given a key object that satisfies a consistency property C_1, and a file object with consistency C_2 from one of the classes we define, our main theorem states that it is enough to ensure the key-monotonicity property in order to obtain consistency for the encrypted file object. This result is subject to certain restrictions on the consistency conditions C_1 and C_2.

In addition, in the full version of this paper [26], we give an example implementation of a consistent encrypted file from a sequentially consistent key object and a fork consistent file object. The proof of consistency of the implementation follows immediately from our main theorem. This demonstrates that complex proofs for showing consistency of encrypted files are simplified using our framework.

The rest of the paper is organized as follows: we survey related work in Section 2, and give the basic definitions, notation and system model in Section 3. We define the two classes of consistency conditions in Section 4 and give the definition of consistency for encrypted files in Section 5. Our main result, a necessary and sufficient condition for constructing consistent encrypted files, is presented in Section 6.

2 Related Work

SUNDR [21] is the first file system that provides consistency guarantees (fork consistency [24]) in a model with a Byzantine storage server and benign clients. In SUNDR, the storage server keeps a signed *version structure* for each user of the file system. The version structures are modified at each read or write operation and are totally ordered as long as the server respects the protocol. A misbehaving server might conceal users' operations from each other and break the total order among version structures, with the effect that users get divided into groups that will never see the same system state again. SUNDR only provides data integrity, but not data confidentiality. In contrast, we are interested in providing consistency guarantees in encrypted storage systems in which keys may change, and so we must consider distribution of the encryption keys, as well.

For obtaining consistency conditions stronger than fork consistency (e.g., linearizability) in the face of Byzantine servers, one solution is to distribute the file server across n replicas, and use this replication to mask the behavior of faulty servers. Modern examples include BFT [9], SINTRA [8] and PASIS [1]. An example of a distributed encrypting file system that provides strong consistency guarantees for both file data and meta-data is FARSITE [2]. File meta-data in FARSITE (that also includes the encryption key for the file) is collectively managed by all users that have access to the file, using a Byzantine fault tolerant protocol. There exist distributed implementations of storage servers that guarantee weaker semantics than linearizability. Lakshmanan et al. [18] provide causal consistent implementations for a distributed storage system. While they discuss encrypted data, they do not treat the impact of encryption on the consistency of the system.

Several network encrypting file systems, such as SiRiUS [14] and Plutus [17], develop interesting ideas for access control and user revocation, but they both leave the key distribution problem to be handled by clients through out-of-band communication. Since the key distribution protocol is not specified, neither of the systems makes any claims about consistency. Other file systems address key management: e.g., SFS [23] separates key management from file system security and gives multiple schemes for key management; Cepheus [12] relies on a trusted server for key distribution; and SNAD [25] uses separate key and file objects to secure network attached storage. However, none of these systems addresses consistency. We refer the reader to the survey by Riedel et al. [27] for an extensive comparison of the security properties of various encrypting file systems.

Another area related to our work is that of consistency semantics. Different applications have different consistency and performance requirements. For this reason, many different consistency conditions for shared objects have been defined and implemented, ranging from strong conditions such as linearizability [16], sequential consistency [19], and timed consistency [28] to loose consistency guarantees such as causal consistency [4], PRAM [22], coherence [15,13], processor consistency [15,13,3], weak consistency [10], entry consistency [7], and release consistency [20]. A generic, continuous consistency model for wide-area replication that generalizes the notion of serializability [6] for transactions on replicated objects has been introduced by Yu and Vahdat [30]. We construct two generic classes of consistency conditions that include and extend some of the existing conditions for shared objects.

Different properties of generic consistency conditions for shared objects have been analyzed in previous work, such as *locality* [29] and *composability* [11]. Locality analyzes for which consistency conditions a history of operations is consistent, given that the restriction of the history to each individual object satisfies the same consistency property. Composability refers to the combination of two consistency conditions for a history into a stronger, more restrictive condition. In contrast, we are interested in the consistency of the combined history of key and file operations, given that the individual operations on keys and files satisfy possibly different consistency properties. We also define generic models of consistency for histories of operations on encrypted file objects that consist of operations on key and file objects.

Generic consistency conditions for shared objects have been restricted previously only to conditions that satisfy the *eventual propagation* property [11]. Intuitively, eventual propagation guarantees that all the write operations are eventually seen by all processes. This assumption is no longer true when the storage server is potentially faulty and we relax this requirement for the class of forking consistency conditions we define.

3 Preliminaries

3.1 Basic Definitions and System Model

Most of our definitions are taken from Herlihy and Wing [16]. We consider a system to be a set of processes $p_1, \dots, p_n$ that invoke operations on a collection of shared objects. Each operation o consists of an *invocation* $\mathsf{inv}(o)$ and a *response* $\mathsf{res}(o)$. We only consider read and write operations on single objects. A write of value v to object X is denoted $X.\mathsf{write}(v)$ and a read of value v from object X is denoted $v \leftarrow X.\mathsf{read}()$.

A *history* H is a sequence of invocations and responses of read and write operations on the shared objects. We consider only *well-formed* histories, in which every invocation of an operation in a history has a matching response. We say that an operation belongs to a history H if its invocation and response are in H. A *sequential history* is a history in which every invocation of an operation is immediately followed by the corresponding response. A *serialization* S of a history H is a sequential history containing all the operations of H and no others. An important concept for consistency is the notion of a *legal sequential history*, defined as a sequential history in which read operations return the values of the most recent write operations.

Notation. For a history H and a process p_i, we denote by $H|p_i$ the sequential history of operations in H done by p_i. For a history H and objects $X_1, \dots, X_n$, we denote by $H|(X_1, \dots, X_n)$ the restriction of H to operations on objects $X_1, \dots, X_n$. We denote by $H|w$ all the write operations in history H and by $H|p_i + w$ the operations in H done by process p_i and all the write operations done by all processes in history H.

3.2 Eventual Propagation

A history satisfies *eventual propagation* [11] if, intuitively, all the write operations done by the processes in the system are eventually seen by all processes. However, the order in which processes see the operations might be different. More formally, eventual propagation is defined below:

Definition 1 (Eventual Propagation and Serialization Set). *A history H satisfies* eventual propagation *if for every process p_i, there exists a legal serialization S_{p_i} of $H|p_i + w$. The set of legal serializations for all processes $S = \{S_{p_i}\}_i$ is called a* serialization set *[11] for history H.*

If a history H admits a legal serialization S, then a serialization set $\{S_{p_i}\}_i$ with $S_{p_i} = S|p_i + w$ can be constructed and it follows immediately that H satisfies eventual propagation.

3.3 Ordering Relations on Operations

There are several natural partial ordering relations that can be defined on the operations in a history H. Here we describe three of them: the *local* (or *process order*), the *causal order* and the *real-time order*.

Definition 2 (Ordering Relations). *Two operations o_1 and o_2 in a history H are ordered by local order (denoted $o_1 \xrightarrow{lo} o_2$) if there exists a process p_i that executes o_1 before o_2.*

The causal order extends the local order relation. We say that an operation o_1 directly precedes *o_2 in history H if either $o_1 \xrightarrow{lo} o_2$, or o_1 is a write operation, o_2 is a read operation and o_2 reads the result written by o_1. The causal order (denoted $\xrightarrow{*}$) is the transitive closure of the direct precedence relation.*

Two operations o_1 and o_2 in a history H are ordered by the real-time order (denoted $o_1 <_H o_2$) if $\mathsf{res}(o_1)$ precedes $\mathsf{inv}(o_2)$ in history H.

A serialization S of a history H induces a *total order* relation on the operations of H, denoted $\xrightarrow{S}$. Two operations o_1 and o_2 in H are ordered by $\xrightarrow{S}$ if o_1 precedes o_2 in the serialization S.

On the other hand, a serialization set $S = \{S_{p_i}\}_i$ of a history H induces a *partial order* relation on the operations of H, denoted $\xrightarrow{S}$. For two operations o_1 and o_2 in H, $o_1 \xrightarrow{S} o_2$ if and only if (i) o_1 and o_2 both appear in at least one serialization S_{p_i} and (ii) o_1 precedes o_2 in all the serializations S_{p_i} in which both o_1 and o_2 appear. If o_1 precedes o_2 in one serialization, but o_2 precedes o_1 in a different serialization, then the operations are concurrent with respect to $\xrightarrow{S}$.

4 Classes of Consistency Conditions

The goal of this paper is to analyze the consistency of encrypted file systems generically and give necessary and sufficient conditions for its realization. A *consistency condition* is a set of histories. We say that a history H is C-*consistent* if $H \in \mathsf{C}$ (this is also denoted by $\mathsf{C}(H)$). Given consistency conditions C and C', C is *stronger* than C' if $\mathsf{C} \subseteq \mathsf{C}'$.

As the space of consistency conditions is very large, we need to restrict ourselves to certain particular and meaningful classes for our analysis. One of the challenges we faced was to define interesting classes of consistency conditions that include some of the well known conditions defined in previous work (i.e., linearizability, causal consistency, PRAM consistency). Generic consistency conditions have been analyzed previously (e.g., [11]), but the class of consistency conditions considered was restricted to conditions with histories that satisfy eventual propagation. Given our system model with a potentially faulty shared storage, we cannot impose this restriction on all the consistency conditions we consider in this work.

We define two classes of consistency conditions, differentiated mainly by the eventual propagation property. The histories that belong to conditions from the first class satisfy eventual propagation and are *orderable*, a property we define below. The histories that belong to conditions from the second class do not necessarily satisfy eventual propagation, but the legal serializations of all processes can be arranged into a *forking tree*. This class includes fork consistency [24], and extends that definition to other new, unexplored consistency conditions. The two classes do not cover all the existing consistency conditions.

4.1 Orderable Conditions

Intuitively, a consistency condition C is orderable if it contains only histories for which there exists a serialization set that respects a certain partial order relation. Consider the example of *causal consistency* [4] defined as follows: a history H is causally consistent if and only if there exists a serialization set S of H that respects the causal order relation, i.e., $\xrightarrow{*} \subseteq \xrightarrow{S}$. We generalize the requirement that the serialization set respects the causal order to more general partial order relations. A subtle point in this definition is the specification of the partial order relation. First, it is clear that the partial order needs to be different for every condition C. But, analyzing carefully the definition of the causal order relation, we notice that it depends on the history H. We can thus view the causal order relation as a family of relations, one for each possible history H. Generalizing, in the definition of an orderable consistency condition C, we require the existence of a family of partial order relations, indexed by the set of all possible histories, denoted by $\{\xrightarrow{\mathsf{C},H}\}_H$. Additionally, we require that each relation $\xrightarrow{\mathsf{C},H}$ respects the local order of operations in H.

Definition 3 (Orderable Consistency Conditions). *A consistency condition C is orderable if there exists a family of partial order relations $\{\xrightarrow{\mathsf{C},H}\}_H$, indexed by the set of all possible histories, with $\xrightarrow{lo} \subseteq \xrightarrow{\mathsf{C},H}$ for all histories H such that:*

$$H \in \mathsf{C} \Leftrightarrow \text{ there exists a serialization set } S \text{ of } H \text{ with } \xrightarrow{\mathsf{C},H} \subseteq \xrightarrow{S} .$$

Given a history H from class C, a serialization set S of H that respects the order relation $\xrightarrow{\mathsf{C},H}$ is called a C-consistent serialization set *of H.*

We define class $\mathcal{C}_O$ to be the set of all orderable consistency conditions. A subclass of interest is formed by those consistency conditions in $\mathcal{C}_O$ that contain only histories for which there exists a legal serialization of their operations. We denote $\mathcal{C}_O^+$ this subclass of $\mathcal{C}_O$. For a consistency condition C from class $\mathcal{C}_O^+$, a serialization S of a history H that respects the order relation $\xrightarrow{\mathsf{C},H}$, i.e., $\xrightarrow{\mathsf{C},H} \subseteq \xrightarrow{S}$, is called a C-*consistent serialization* of H.

Linearizability [16] and sequential consistency [19] belong to $\mathcal{C}_O^+$ (with the corresponding ordering relations $<_H$ and $\xrightarrow{lo}$, respectively), and PRAM [22] and causal consistency [4] to $\mathcal{C}_O \setminus \mathcal{C}_O^+$ (with the corresponding ordering relations $\xrightarrow{lo}$ and $\xrightarrow{*}$, respectively).

4.2 Forking Conditions

To model encrypted file systems over untrusted storage, we need to consider consistency conditions that might not satisfy the eventual propagation property. In a model with potentially faulty storage, it might be the case that a process views only a subset of the writes of the other processes, besides the operations it performs. For this purpose, we need to extend the notion of serialization set.

Definition 4 (Extended and Forking Serialization Sets). *An extended serialization set of a history H is a set $S = \{S_{p_i}\}_i$ with S_{p_i} a legal serialization of a subset of* *operations from H, that includes (at least) all the operations done by process p_i. A forking serialization set of a history H is an extended serialization set $S = \{S_{p_i}\}_i$ such that for all $i, j, (i \neq j)$, any $o \in S_{p_i} \cap S_{p_j}$, and any $o' \in S_{p_i}$:*

$$o' \xrightarrow{S_{p_i}} o \Rightarrow (o' \in S_{p_j} \wedge o' \xrightarrow{S_{p_j}} o).$$

A forking serialization set is an extended serialization set with the property that its serializations can be arranged into a "forking tree". Intuitively, arranging the serializations in a tree means that any two serializations might have a common prefix of identical operations, but once they diverge, they do not contain any of the same operations. Thus, the operations that belong to a subset of serializations must be ordered the same in all those serializations. A forking consistency condition includes only histories for which a forking serialization set can be constructed. Moreover, each serialization S_{p_i} in the forking tree is a C-consistent serialization of the operations seen by p_i, for C a consistency condition from $\mathcal{C}_{\mathcal{O}}^{+}$.

Definition 5 (Forking Consistency Conditions). *A consistency condition* $\mathsf{FORK\text{-}C}$ *is* forking *if:*

1. C *is a consistency condition from* $\mathcal{C}_{\mathcal{O}}^{+}$*;*
2. $H \in \mathsf{FORK\text{-}C}$ *if and only if there exists a forking serialization set $S = \{S_{p_i}\}_i$ for history H with the property that each S_{p_i} is* C*-consistent.*

We define class $\mathcal{C}_{\mathcal{F}}$ to be the set of all forking consistency conditions $\mathsf{FORK\text{-}C}$. It is immediate that for consistency conditions C, C_1 and C_2 in $\mathcal{C}_{\mathcal{O}}^{+}$, (i) C is stronger than $\mathsf{FORK\text{-}C}$, and (ii) if C_1 is stronger than C_2, then $\mathsf{FORK\text{-}C}_1$ is stronger than $\mathsf{FORK\text{-}C}_2$.

$\mathcal{C}_{\mathcal{F}}$ extends the notion of fork consistency defined by Mazieres and Shasha [24].

5 Definition of Consistency for Encrypted Files

We can construct an encrypted file object using two components, the file object and the key object whose values are used to encrypt file contents. File and key objects might be implemented via different protocols and infrastructures. For the purpose of this paper, we consider each file to be associated with a distinct encryption key. We could easily extend this model to accommodate different granularity levels for the encryption keys (e.g., a key for a group of files).

Users perform operations on an encrypted file object that involve operations on both the file and the key objects. For example, a read of an encrypted file might require a read of the encryption key first, then a read of the file and finally a decryption of the file with the key read. We refer to the operations exported by the storage interface (i.e., operations on encrypted file objects) to its users as "high-level" operations and the operations on the file and key objects as "low-level" operations.

We model a cryptographic file system as a collection of encrypted files. Different cryptographic file systems export different interfaces of high-level operations to their users. We can define consistency for encrypted file objects offering a wide range of high-level operation interfaces, as long as the high-level operations consist of low-level write and read operations on key and file objects. We do assume that a process that creates an encryption key writes this to the relevant key object before writing any files encrypted with that key.

The encryption key for a file is changed most probably when some users are revoked access to the file, and thus, for security reasons, we require that *clients use the most recent key they have seen to write new file contents*. However, it is possible to use older versions of the encryption key to decrypt a file read. For example, in a *lazy revocation* model [12,17,5], the re-encryption of a file is not performed immediately when a user is revoked access to the file and the encryption key for that file is changed, but it is delayed until the next write to that file. Thus, in the lazy revocation model older versions of the key might be used to decrypt files, but new file contents are encrypted with the most recent key. In our model, we can accommodate both the lazy revocation method and the *active revocation* method in which a file is immediately re-encrypted with the most recent encryption key at the moment of revocation.

For completeness, here we give an example of a high-level operation interface for an encrypted file object ENCF, which is used in the example implementation given in the full version of this paper [26] :

1. Create a file, denoted as $\mathsf{ENCF.create\_file}(f)$. This operation generates a new encryption key k for the file, writes k to the key object and writes the file content f encrypted with key k to the file object.
2. Encrypt and write a file, denoted as $\mathsf{ENCF.write\_encfile}(f)$. This operation writes an encryption of file contents f to the file object, using the most recent encryption key that the client read.
3. Read and decrypt a file, denoted as $f \leftarrow \mathsf{ENCF.read\_encfile}()$. This operation reads an encrypted file from the file object and then decrypts it to f.
4. Write an encryption key, denoted as $\mathsf{ENCF.write\_key}(k)$. This operation changes the encryption key for the file to a new value k. Optionally, it re-encrypts the file contents with the newly generated encryption key if active revocation is used.

Consider a fixed implementation of high-level operations from low-level read and write operations. Each execution of a history H of high-level operations naturally induces a history H_l of low-level operations by replacing each completed high-level operation with the corresponding sequence of invocations and responses of the low-level operations. In the following, we define consistency $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$ for encrypted file objects, for any consistency properties C_1 and C_2 of the key distribution and file access protocols that belong to classes $\mathcal{C}_\mathcal{O}$ or $\mathcal{C}_\mathcal{F}$.

Definition 6. *(Consistency of Encrypted File Objects) Let H be a history of completed high-level operations on an encrypted file object* ENCF *and* C_1 *and* C_2 *two consistency properties from $\mathcal{C}_\mathcal{O}$. Let H_l be the corresponding history of low-level operations on key object* KEY *and file object* FILE *induced by an execution of high-level operations. We say that H is* $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$**-consistent** *if there exists a serialization set $S = \{S_{p_i}\}_i$ of H_l such that:*

1. *S is* enc*-legal, i.e.: For every file write operation $o = \mathsf{FILE.write}(c)$, there is an operation* $\mathsf{KEY.write}(k)$ *such that: c was generated through encryption with key*

k, $\mathsf{KEY.write}(k) \xrightarrow{S_{p_i}} o$ and there is no $\mathsf{KEY.write}(k')$ with $\mathsf{KEY.write}(k) \xrightarrow{S_{p_i}}$ $\mathsf{KEY.write}(k') \xrightarrow{S_{p_i}} o$ for all i;

2. *$S|\mathsf{KEY} = \{S_{p_i}|\mathsf{KEY}\}_i$ is a C_1-consistent serialization set of $H_l|\mathsf{KEY}$;*
3. *$S|\mathsf{FILE} = \{S_{p_i}|\mathsf{FILE}\}_i$ is a C_2-consistent serialization set of $H_l|\mathsf{FILE}$;*
4. *S respects the local ordering of each process.*

Intuitively, our definition requires that there is an arrangement (i.e., serialization set) of key and file operations such that the most recent key write operation before each file write operation seen by each client is the write of the key used to encrypt that file. In addition, the serialization set should respect the desired consistency of the key distribution and file access protocols.

If both C_1 and C_2 belong to $\mathcal{C}_{\mathcal{O}}^{+}$, then the definition should be changed to require the existence of a serialization S of H_l instead of a serialization set. Similarly, if C_2 belongs to $\mathcal{C}_{\mathcal{F}}$, we change the definition to require the existence of an extended serialization set $\{S_{p_i}\}_i$ of H_l. In the latter case, the serialization S_{p_i} for each process might not contain all the key write operations, but it has to include all the key operations that write key values used in subsequent file operations in the same serialization. Conditions (1), (2), (3) and (4) remain unchanged.

The definition can be immediately generalized to multiple encrypted file objects, as was done in the full version of this paper [26].

6 A Necessary and Sufficient Condition for the Consistency of Encrypted File Objects

After defining consistency for encrypted file objects, here we give necessary and sufficient conditions for the realization of the definition. We first outline the dependency among encryption keys and file objects, and then define a property of histories that ensures that file write operations are executed in increasing order of their encryption keys. Histories that satisfy this property are called *key-monotonic*. Our main result, Theorem 1, states that, provided that the key distribution and the file access protocols satisfy some consistency properties C_1 and C_2 with some restrictions, the key-monotonicity property of the history of low-level operations is necessary and sufficient to implement $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$ consistency for the encrypted file object.

6.1 Dependency among Values of Key and File Objects

Each write and read low-level operation is associated with a value. The value of a write operation is its input argument and that of a read operation its returned value. For o a file operation with value f done by process p_i, k the value of the key that encrypts f and $w = \mathsf{KEY.write}(k)$ the operation that writes the key value k, we denote the dependency among operations w and o by $\mathsf{R}(w, o)$ and say that file operation *o is associated with* key operation w.

The relation $\mathsf{R}(w, o)$ implies a causal order relation in the history of low-level operations between operations w and o. Since process p_i uses the key value k to encrypt the file content f, then either: (1) in process p_i there is a read operation $r = (k \leftarrow$

$\mathsf{KEY.read}())$ such that $w \xrightarrow{*} r \xrightarrow{lo} o$, which implies $w \xrightarrow{*} o$; or (2) w is done by process p_i, in which case $w \xrightarrow{lo} o$, which implies $w \xrightarrow{*} o$. In either case, the file operation o is causally dependent on the key operation w that writes the value of the key used in o.

6.2 Key-Monotonic Histories

A history of key and file operations is *key-monotonic* if, intuitively, it admits a consistent serialization for each process in which the file write operations use monotonically increasing versions of keys for encryption of their values. Intuitively, if a client uses a key version to perform a write operation on a file, then all the future write operations on the file object by all the clients will use this or later versions of the key.

We give an example in Figure 1 of a history that is not key-monotonic for sequential consistent keys and linearizable files. Here c_1 and c_2 are file values encrypted with key values k_1 and k_2, respectively. k_1 is ordered before k_2 with respect to the local order. $\mathsf{FILE.write}(c_1)$ is after $\mathsf{FILE.write}(c_2)$ with respect to the real-time ordering, and, thus, in any linearizable serialization of file operations, c_2 is written before c_1.

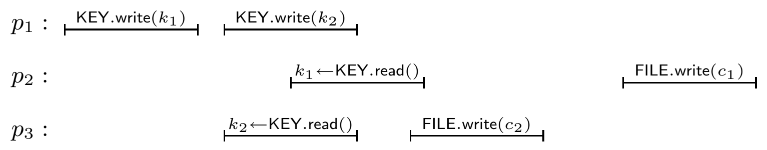

Fig. 1. A history that is not key-monotonic

To define key-monotonicity for a low-level history formally, we would like to find the minimal conditions for its realization, given that the key operations in the history satisfy consistency condition C_1 and the file operations satisfy consistency condition C_2. We assume that the consistency C_1 of the key operations is orderable. Two conditions have to hold in order for a history to be key-monotonic: (1) the key write operations cannot be ordered in opposite order of the file write operations that use them; (2) file write operations that use the same keys are not interleaved with file write operations using a different key.

Definition 7 (Key-Monotonic History). *Consider a history H with two objects, key* KEY *and file* FILE*, such that* $\mathsf{C}_1(H|\mathsf{KEY})$ *and* $\mathsf{C}_2(H|\mathsf{FILE})$*, where* C_1 *is an orderable consistency condition and* C_2 *belongs to either* $\mathcal{C}_\mathcal{O}$ *or* $\mathcal{C}_\mathcal{F}$*. H is a* key-monotonic history *with respect to* C_1 *and* C_2*, denoted* $\mathsf{KM}_{\mathsf{C}_1,\mathsf{C}_2}(H)$*, if there exists a* C_2*-consistent serialization (or serialization set or forking serialization set) S of* $H|\mathsf{FILE}$ *such that the following conditions holds:*

- (KM_1) *for any two file write operations* $f_1 \xrightarrow{S} f_2$ *with associated key write operations* k_1 *and* k_2 *(i.e.,* $\mathsf{R}(k_1, f_1)$*,* $\mathsf{R}(k_2, f_2)$*), it cannot happen that* $k_2 \xrightarrow{\mathsf{C}_1, H|\mathsf{KEY}} k_1$.
- (KM_2) *for any three file write operations* $f_1 \xrightarrow{S} f_2 \xrightarrow{S} f_3$*, and key write operation k with* $\mathsf{R}(k, f_1)$ *and* $\mathsf{R}(k, f_3)$*, it follows that* $\mathsf{R}(k, f_2)$.

The example we gave in Figure 1 violates the first condition. If we consider $f_2 = \mathsf{FILE.write}(c_2)$, $f_1 = \mathsf{FILE.write}(c_1)$, then f_2 is ordered before f_1 in any linearizable serialization and k_1 is ordered before k_2 with respect to the local order. But condition (KM_1) states that it is not possible to order key write k_1 before key write k_2.

The first condition (KM_1) is enough to guarantee key-monotonicity for a history H when the key write operations are uniquely ordered by the ordering relation $\xrightarrow{\mathsf{C}_1, H|\mathsf{KEY}}$. To handle concurrent key writes with respect to $\xrightarrow{\mathsf{C}_1, H|\mathsf{KEY}}$, we need to enforce the second condition (KM_2) for key-monotonicity. Condition (KM_2) rules out the case in which uses of the values written by two concurrent key writes are interleaved in file operations in a consistent serialization. Consider the example from Figure 2 that is not key-monotonic for sequential consistent key operations and linearizable file operations. In this example c_1 and c_1' are encrypted with key value k_1, and c_2 is encrypted with key value k_2. A linearizable serialization of the file operations is: $\mathsf{FILE.write}(c_1)$; $\mathsf{FILE.write}(c_2)$; $\mathsf{FILE.read}(c_2)$; $\mathsf{FILE.write}(c_1')$, and this is not key-monotonic. k_1 and k_2 are not ordered with respect to the local order, and as such the history does not violate condition (KM_1). However, condition (KM_2) is not satisfied by this history.

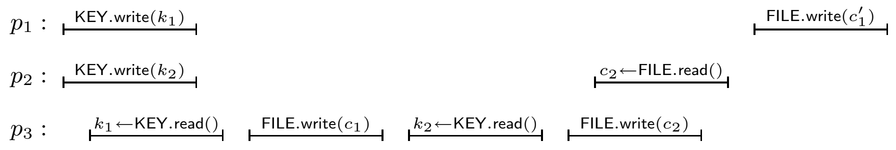

Fig. 2. A history that does not satisfy condition (KM_2)

In cryptographic file system implementations, keys are usually changed only by one process, who might be the owner of the file or a trusted entity. For single-writer objects, it can be proved that sequential consistency, causal consistency and PRAM consistency are equivalent. Since we require the consistency of key objects to be orderable and all orderable conditions are at least PRAM consistent (i.e., admit serialization sets that respect the local order), the weakest consistency condition in the class of orderable conditions for single writer objects is equivalent to sequential consistency. If the key distribution protocol is sequential consistent, the key-monotonicity conditions given in Definition 7 can be simplified. We present below the simplified condition. The proof of equivalence with the conditions from Definition 7 is given in the full version of this paper [26].

Proposition 1. *Let H be a history of operations on the single-writer key object* KEY *and file object* FILE *such that $H|\mathsf{KEY}$ is sequential consistent. H is key-monotonic if and only if the following condition is true:*

*(*SW-KM*) There exists a C_2-consistent serialization S (or serialization set or forking serialization set) of $H|\mathsf{FILE}$ such that for any two file write operations $f_1 \xrightarrow{S} f_2$ with associated key write operations k_1 and k_2 (i.e.,* $\mathsf{R}(k_1, f_1)$*,* $\mathsf{R}(k_2, f_2)$*), it follows that $k_1 \xrightarrow{lo} k_2$ or $k_1 = k_2$.*

6.3 Obtaining Consistency for Encrypted File Objects

We give here the main result of our paper, a necessary and sufficient condition for implementing consistent encrypted file objects, as defined in Section 5. Given a key distribution protocol with orderable consistency C_1 and a file access protocol that satisfies consistency C_2 from classes $\mathcal{C}_\mathcal{O}$ or $\mathcal{C}_\mathcal{F}$, the theorem states that key-monotonicity is a necessary and sufficient condition to obtain consistency $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$ for the encrypted file object. Some additional restrictions need to be satisfied. The proof of the theorem is in the full version of this paper [26].

Theorem 1. *Consider a fixed implementation of high-level operations from low-level operations. Let H be a history of operations on an encrypted file object* ENCF *and H_l the induced history of low-level operations on key object* KEY *and file object* FILE *by a given execution of high-level operations. Suppose that the following conditions are satisfied: (1)* $\mathsf{C}_1(H_l|\mathsf{KEY})$*; (2)* $\mathsf{C}_2(H_l|\mathsf{FILE})$*; (3)* C_1 *is orderable; (4) if* C_2 *belongs to* $\mathcal{C}_\mathcal{O}^+$*, then* C_1 *belongs to* $\mathcal{C}_\mathcal{O}^+$*. Then H is* $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$*-consistent if and only if H_l is a key-monotonic history, i.e.,* $\mathsf{KM}_{\mathsf{C}_1,\mathsf{C}_2}(H)$*.*

Discussion. Our theorem recommends two main conditions to file system developers in order to guarantee $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$-consistency of encrypted file objects. First, the consistency of the key distribution protocol needs to satisfy eventual propagation (as it belongs to class $\mathcal{C}_\mathcal{O}$) to apply our theorem. This suggests that using the untrusted storage server for the distribution of the keys, as implemented in several cryptographic file systems, e.g., SNAD [25] and SiRiUS [14], might not meet our consistency definitions. For eventual propagation, the encryption keys have to be distributed either directly by file owners or by using a trusted key server. It is an interesting open problem to analyze the enc-consistency of the history of high-level operations if both the key distribution and file-access protocols have consistency in class $\mathcal{C}_\mathcal{F}$. Secondly, the key-monotonicity property requires, intuitively, that file writes are ordered not to conflict with the consistency of the key operations. To implement this condition, one solution is to modify the file access protocol to take into account the version of the encryption key used in a file operation when ordering that file operation. We give an example of modifying the fork consistent protocol given by Mazieres and Shasha [24] in the full version of this paper [26].

Moreover, the framework offered by Theorem 1 simplifies complex proofs for showing consistency of encrypted files. In order to apply Definition 6 directly for such proofs, we need to construct a serialization of the history of low-level operations on both the file and key objects and prove that the file and key operations are correctly interleaved in this serialization and respect the appropriate consistency conditions. By Theorem 1, given a key distribution and file access protocol that is each known to be consistent, verifying the consistency of the encrypted file object is equivalent to verifying key monotonicity. To prove that a history of key and file operations is key monotonic, it is enough to construct a serialization of the file operations and prove that it does not violate the ordering of the key operations. The simple proof of consistency of the example encrypted file object presented in the full version of this paper [26] demonstrates the usability of our framework.

7 Conclusions

We have addressed the problem of consistency in encrypted file systems. An encrypted file system consists of two main components: a file access protocol and a key distribution protocol, which might be implemented via different protocols and infrastructures. We formally define generic consistency in encrypted file systems: for any consistency conditions C_1 and C_2 belonging to the classes we consider, we define a corresponding consistency condition for encrypted file systems, $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$. The main result of our paper states that if each of the two protocols has some consistency guarantees with some restrictions, then ensuring that the history of low-level operation is key-monotonic is necessary and sufficient to obtain consistency for an encrypted file object. The applicability of our definitions and main result to other classes of consistency conditions is a topic of future work.

Another contribution of this paper is to define two classes of consistency conditions that extend and generalize existing conditions: the first class includes classical consistency conditions such as linearizability and causal consistency, and the second one extends fork consistency. An interesting problem is to find efficient implementations of the new forking consistency conditions from the second class and their relation with existing consistency conditions.

References

1. M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," in *Proc. 20th ACM Symposium on Operating Systems (SOSP)*, pp. 59–74, ACM, 2005.
2. A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment," in *Proc. 5th Symposium on Operating System Design and Implementation (OSDI)*, Usenix, 2002.
3. M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger, "The power of processor consistency," Technical Report GIT-CC-92/34, Georgia Institute of Technology, 1992.
4. M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto, "Causal memory: Definitions, implementation and programming," *Distributed Computing*, vol. 1, no. 9, pp. 37–49, 1995.
5. M. Backes, C. Cachin, and A. Oprea, "Secure key-updating for lazy revocation," in *Proc. 11th European Symposium On Research In Computer Security (ESORICS)*, Springer-Verlag, 2006.
6. P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
7. B. Bershad, M. Zekauskas, and W. Sawdon, "The Midway distributed shared-memory system," in *Proc. IEEE COMPCON Conference*, pp. 528–537, IEEE, 1993.
8. C. Cachin and J. A. Poritz, "Secure intrusion-tolerant replication on the internet," in *Proc. International Conference on Dependable Systems and Networks (DSN)*, pp. 167–176, IEEE, 2002.
9. M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. 3rd Symposium on Operating System Design and Implementation (OSDI)*, pp. 173–186, Usenix, 1999.
10. M. Dubois, C. Scheurich, and F. Briggs, "Synchronization, coherence and event ordering in multiprocessors," *IEEE Computer*, vol. 21, no. 2, pp. 9–21, 1988.

11. R. Friedman, R. Vitenberg, and G. Chockler, "On the composability of consistency conditions," *Information Processing Letters*, vol. 86, pp. 169–176, 2002.
12. K. Fu, "Group sharing and random access in cryptographic storage file systems," Master's thesis, Massachusetts Institute of Technology, 1999.
13. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proc. 17th Annual International Symposium on Computer Architecture*, pp. 15–26, 1990.
14. E. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in *Proc. Network and Distributed Systems Security (NDSS) Symposium 2003*, pp. 131–145, ISOC, 2003.
15. J. Goodman, "Cache consistency and sequential consistency," Technical Report 61, SCI Committee, 1989.
16. M. Herlihy and J. Wing, "Linearizability: A corretness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.
17. M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.
18. S. Lakshmanan, M. Ahamad, and H. Venkateswaran:, "A secure and highly available distributed store for meeting diverse data storage needs," in *Proc. International Conference on Dependable Systems and Networks (DSN)*, pp. 251–260, IEEE, 2001.
19. L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. 28, no. 9, pp. 690–691, 1979.
20. D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford Dash multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, 1992.
21. J. Li, M. Krohn, D. Mazieres, and D. Shasha, "Secure untrusted data repository," in *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)*, pp. 121–136, Usenix, 2004.
22. R. Lipton and J. Sandberg, "Pram: A scalable shared memory," Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, 1988.
23. D. Mazieres, M. Kaminsky, M. Kaashoek, and E. Witchel, "Separating key management from file system security," in *Proc. 17th ACM Symposium on Operating Systems (SOSP)*, pp. 124–139, ACM, 1999.
24. D. Mazieres and D. Shasha, "Building secure file systems out of Byzantine storage," in *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 108–117, ACM, 2002.
25. E. Miller, D. Long, W. Freeman, and B. Reed, "Strong security for distributed file systems," in *Proc. First USENIX Conference on File and Storage Technologies (FAST)*, pp. 1–13, 2002.
26. A. Oprea and M. K. Reiter, "On consistency of encrypted files," Technical Report CMU-CS-06-113, Carnegie Mellon University, 2006. Available from `http://reports-archive.adm.cs.cmu.edu/anon/2006/CMU-CS-06-113.pdf`.
27. E. Riedel, M. Kallahalla, and R. Swaminathan, "A framework for evaluating storage system security," in *Proc. First USENIX Conference on File and Storage Technologies (FAST)*, pp. 15–30, 2002.
28. F. J. Torres-Rojas, M. Ahamad, and M. Raynal, "Timed consistency for shared distributed objects," in *Proc. 18th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 163–172, ACM, 1999.
29. R. Vitenberg and R. Friedman, "On the locality of consistency conditions," in *Proc. 17th International Symposium on Distributed Computing (DISC))*, pp. 92–105, 2003.
30. H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," *ACM Transactions on Computer Systems*, vol. 20, no. 3, pp. 239–282, 2002.