On Consistency of Encrypted Files

Alina Oprea[*] **Michael K. Reiter**[†]

March 2005
CMU-CS-06-113

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA; alina@cs.cmu.edu
[†]Electrical and Computer Engineering Department and Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA; reiter@cmu.edu

Keywords: cryptographic file systems, shared objects, consistency models, linearizability, fork consistency.

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 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 then 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. Lastly, we give an example implementation of a consistent encrypted file object, utilizing a fork consistent file access protocol and a sequentially consistent key distribution protocol. The proof of consistency of the implementation builds from our main theorem.

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 [17], 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 [20], causal consistency [3], PRAM consistency [23] and more recently, fork consistency [25], 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 there is an arrangement of key and file operations such that the most recent key write operation before each file operation seen by each client is the write of the key value used to encrypt the file contents. In addition, the arrangement of key and file operations should respect the desired consistency for the key and file operations.

Rather than investigate consistency for a single implementation of an encrypted file, we consider a collection of implementations that are all *key-monotonic*. Intuitively, if a client uses a key version to perform an operation on a file, then in a key-monotonic history all the future operations on the file object will use this or a later version of the key. We formally define this property that depends on the consistency of the key and file objects. We prove in our main result (Theorem 2) 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. We give an example implementation that uses a sequentially consistent key distribution protocol and a fork consistent file access protocol [25], and prove its consistency using our framework.

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 it extends fork consistency [25] 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.

- Lastly, 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. Finally, we describe a consistent encrypted file object implementation from a sequentially consistent key distribution protocol and a fork consistent file access protocol in Section 7.

2 Related Work

Initial research on secure file systems (CFS [7], TCFS [10]) considered file systems with limited sharing of information among their users. More recently, research in this area focused more on network file systems. In these systems, users can share files and access them concurrently.

SUNDR [22] is the first file system that provides consistency guarantees (fork consistency [25]) 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 stronger consistency conditions 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 efficient implementations based on state machine replication (see the survey of Schneider [29]) include BFT [9] and SINTRA [8]. There exist distributed implementations of storage servers that guarantee weaker semantics than linearizability. Lakshmanan et al. [19] 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 encrypted file systems, such as SiRiUS [15] and Plutus [18], 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 [24] separates key management from file system security and gives multiple schemes for key management; Cepheus [13] relies on a trusted server for key distribution; and SNAD [27] 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. [28] for an extensive comparison of the security properties of the existing 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 [17], sequential consistency [20], and timed consistency [30] to loose consistency guarantees such as causal consistency [3], PRAM [23], coherence [16, 14], processor consistency [16, 14, 2], weak consistency [11], entry consistency [6], and release consistency [21]. A generic, continuous consistency model for wide-area replication that generalizes the notion of serializability [5] for transactions on replicated objects has been introduced by Yu and Vahdat [32]. 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* [31] and *composability* [12]. Locality analyzes for which consis-

tency 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 a property called *eventual propagation* [12]. 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 [17]. We consider a system to be a set of processes $p_1, \ldots, p_n$ 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 operations in H done by process p_i (this is a sequential history). For a history H and objects $X, X_1, \ldots, X_n$, we denote by $H|X$ the restriction of H to operations on object X and by $H|(X_1, \ldots, X_n)$ the restriction of H to operations on objects $X_1, \ldots, X_n$. We denote $H|w$ all the write operations in history H and $H|p_i + w$ the operations 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* [12] 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* [12] 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), and are also meaningful for encrypted file objects over untrusted storage. Generic consistency conditions have been analyzed previously (e.g., [12]), 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 can not 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 tree (denoted *forking tree*). This class includes fork consistency [25], and extends that definition to other new, unexplored consistency conditions. The two classes do not cover all the existing (or possible) 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* [3] 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 $\mathcal{C}_{\mathcal{O}}$ to be the set of all orderable consistency conditions. A subclass of interest is formed by those consistency conditions in $\mathcal{C}_{\mathcal{O}}$ that contain only histories for which there exists a legal serialization of their operations. We denote $\mathcal{C}_{\mathcal{O}}^{+}$ this subclass of $\mathcal{C}_{\mathcal{O}}$. For a consistency condition C from class $\mathcal{C}_{\mathcal{O}}^{+}$, 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 [17] and sequential consistency [20] belong to $\mathcal{C}_{\mathcal{O}}^{+}$, and PRAM [23] and causal consistency [3] to $\mathcal{C}_{\mathcal{O}} \setminus \mathcal{C}_{\mathcal{O}}^{+}$. The partial ordering relations corresponding to each of these conditions, as well as other examples of consistency conditions and consistent histories, are given in Section 4.3.

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 operation. 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 FORK-C is *forking* if:

1. C is a consistency condition from $\mathcal{C}_{\mathcal{O}}^{+}$;
2. $H \in$ FORK-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 FORK-C. It is immediate that for consistency conditions C, C_1 and C_2 in $\mathcal{C}_{\mathcal{O}}^{+}$, (i) C is stronger than FORK-C, and (ii) if C_1 is stronger than C_2, then FORK-C_1 is stronger than FORK-C_2.

We can prove that fork consistency, as defined by Mazieres and Shasha [25], belongs to class $\mathcal{C}_{\mathcal{F}}$ and, in fact, is equivalent to FORK-Linearizability. We omit the proof here due to space limitations.

4.3 Examples

A table summarizing the type of serialization required for histories of each class of consistency conditions defined, as well as several examples of existing and new consistency conditions from each class and their partial order relations is given in Figure 1.

Examples of a linearizable and a causally consistent history are given in Figures 2 and 3, respectively. The history from Figure 2 admits the total ordering $X.\mathsf{write}(1); X.\mathsf{write}(2); X.\mathsf{write}(3); 3 \leftarrow X.\mathsf{read}()$, which respects the real-time ordering, and is thus linearizable. The history from

Class	Type of Serialization	Example of Condition	Partial Order
$\mathcal{C}_{\mathcal{O}}^{+}$	Serialization	Linearizability [17]	$<_H$
		Sequential Consistency [20]	$\xrightarrow{lo}$
$\mathcal{C}_{\mathcal{O}} \setminus \mathcal{C}_{\mathcal{O}}^{+}$	Serialization set	Causal Consistency [3]	$\xrightarrow{*}$
		PRAM Consistency [23]	$\xrightarrow{lo}$
$\mathcal{C}_{\mathcal{F}}$	Forking Serialization Set	FORK-Linearizability	$<_H$
		FORK-Sequential Consistency	$\xrightarrow{lo}$

Figure 1: Classes of Consistency Conditions

Figure 3 does not admit a legal sequential ordering of its operations, but it admits a serialization for each process that respects the causal order. The serialization for process p_1 is $X.\mathsf{write}(1)$; $X.\mathsf{write}(2)$; $2 \leftarrow X.\mathsf{read}()$ and that for process p_2 is $X.\mathsf{write}(2)$; $X.\mathsf{write}(1)$; $1 \leftarrow X.\mathsf{read}()$.

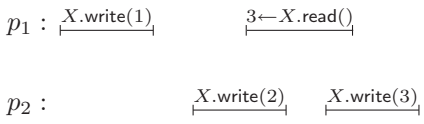

Figure 2: Linearizable history.

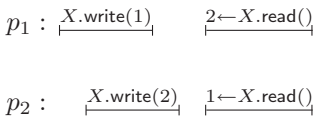

Figure 3: Causal consistent history.

We give an example of a history in Figure 4 that is not linearizable, but that accepts a forking tree shown in Figure 5 with each branch in the tree linearizable. Processes p_1 and p_2 do not see any operations performed by process p_3. Process p_3 sees only the writes on object X done by p_1 and p_2, respectively, but no other operations done by p_1 or p_2. Each path in the forking tree from the root to a leaf corresponds to a serialization for a process. Each branch in the tree respects the real-time ordering relation, and as such the history is FORK-Linearizable.

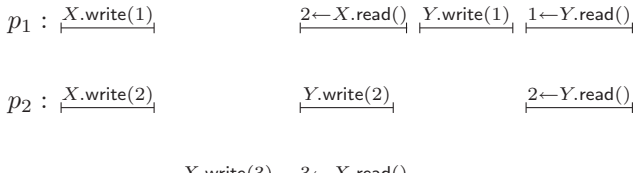

Figure 4: FORK-Linearizable history.

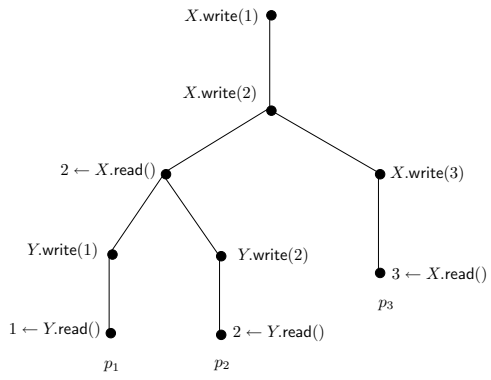

Figure 5: A forking tree for the history.

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 file objects. 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 [13, 18], 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 will be used in the example implementation given in Section 7:

1. Create a file, denoted as ENCF.create_file(f). This operation generates a new encryption key k for the file, writes the file content f encrypted with key k to the file object, and writes k to the key object.

2. Encrypt and write a file, denoted as 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 $f \leftarrow$ 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 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 operation seen by each client is the write of the key used to encrypt or decrypt 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 both C_1 and C_2 belong 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.

Generalization to multiple encrypted file objects. Our definition can be generalized to encrypted file systems that consist of multiple encrypted file objects $\mathsf{ENCF}_1, \ldots, \mathsf{ENCF}_n$. We assume that there is a different key object KEY_i for each file object FILE_i, for $i = 1, \ldots, n$.

Definition 7. (Consistency of Encrypted File Systems) Let H be a history of completed high-level operations on encrypted file objects $\mathsf{ENCF}_1, \ldots, \mathsf{ENCF}_n$ 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 objects $\mathsf{KEY}_1, \ldots, \mathsf{KEY}_n$ and file objects $\mathsf{FILE}_1, \ldots, \mathsf{FILE}_n$ 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 $o = \mathsf{FILE}_j.\mathsf{write}(c)$, there is an operation $\mathsf{KEY}_j.\mathsf{write}(k)$ such that: c is encrypted with key value k, $\mathsf{KEY}_j.\mathsf{write}(k) \xrightarrow{S_{p_i}} o$ and there is no $\mathsf{KEY}_j.\mathsf{write}(k')$ with $\mathsf{KEY}_j.\mathsf{write}(k) \xrightarrow{S_{p_i}} \mathsf{KEY}_j.\mathsf{write}(k') \xrightarrow{S_{p_i}} o$ for all i;
2. For all j, $S|\mathsf{KEY}_j$ is a C_1-consistent serialization set of $H_l|\mathsf{KEY}_j$;
3. $S|(\mathsf{FILE}_1, \ldots, \mathsf{FILE}_n)$ is a C_2-consistent serialization set of $H_l|(\mathsf{FILE}_1, \ldots, \mathsf{FILE}_n)$;
4. S respects the local ordering of each process.

The reason for condition 3 in the above definition is that a consistency property for a file system (as defined in the literature) refers to the consistency of operations on all file objects. In contrast, we are not interested in the consistency of all the operations on the key objects, as different key objects are used to encrypt values of different files, and are thus independent. We only require in condition (2) consistency for individual key objects.

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 operations are executed in increasing order of their encryption keys. Histories that satisfy this property are called *key-monotonic*. Our main result, Theorem 2, 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. There is a causal order relation in the history of low-level operations between a file operation and the key operation that writes the key used to encrypt that file. More precisely, if o is a file operation with value f done by process p_i, k is the value of the key that encrypts f and $w = \mathsf{KEY}.\mathsf{write}(k)$ is the operation that writes the key value k, then either: (1) in process p_i there is a read operation $r = (k \leftarrow \mathsf{KEY}.\mathsf{read}())$ 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. We denote this dependency between file operation o and key write operation w by $\mathsf{R}(w, o)$ and say that file operation o *is associated with* key operation w.

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 operations use monotonic (for a given partial order) keys for encryption and decryption of their values. Intuitively, if a client uses a key version to perform an operation on a file, then all the future operations on the file object by all the clients will use this or later versions of the key.

We give an example in Figure 6 of a history that is not key-monotonic for sequential consistent key operations and linearizable file operations. 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.

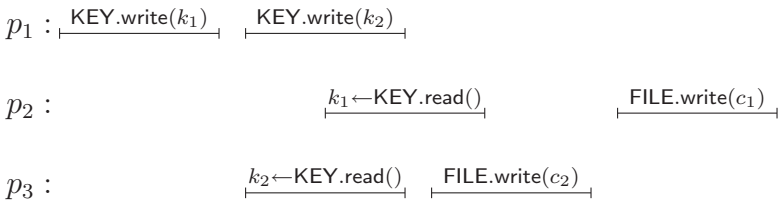

Figure 6: A history that is not key-monotonic.

We are interested in finding the minimal conditions that make a low-level history key-monotonic, 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 8 (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. 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 6 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 when the key write operations are uniquely ordered. To handle concurrent key writes, 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 7 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.

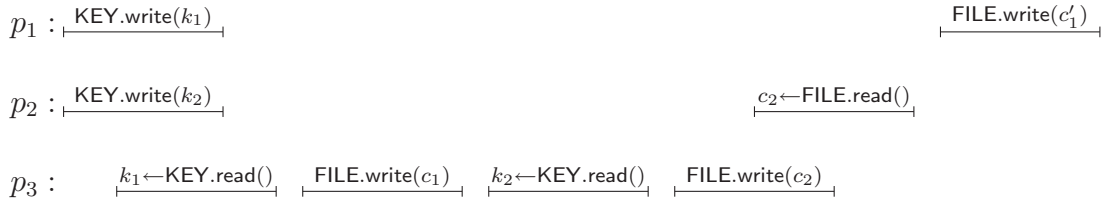

Figure 7: A history that does not satisfy condition (KM_2).

Sequentially Consistent Key Objects with a Single Writer 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 condition defined previously can be simplified. We present below the simplified condition and the proof of equivalence with the conditions from Definition 8.

Proposition 1. *Let H be a history of operations on the single-writer key object* KEY *and file object* FILE *such that $H|$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 $H|$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 $k_1 = k_2$.*

Proof. Suppose first that the conditions from Definition 8 are true. For a single-writer key object, all the key write operations are performed by a single process, and are thus ordered by local order. Then, for any two file write operations f_1 and f_2 with associated key write operations k_1 and k_2, it

is true that either $k_1 \xrightarrow{lo} k_2$ or $k_2 \xrightarrow{lo} k_1$ or $k_1 = k_2$. Condition (KM_1) from Definition 8 implies that $k_1 \xrightarrow{lo} k_2$ or $k_1 = k_2$, which proves condition (SW-KM).

In the reverse direction, suppose that condition (SW-KM) is true. This immediately implies condition (KM_1) from Definition 8. To prove condition (KM_2), consider three file write operations $f_1 \xrightarrow{S} f_2 \xrightarrow{S} f_3$ and key write operation k such that $\mathsf{R}(k, f_1)$ and $\mathsf{R}(k, f_3)$. Let k_2 be the key write operation associated with file operation f_2, i.e., $\mathsf{R}(k_2, f_2)$. By condition (SW-KM), it follows that $k \xrightarrow{lo} k_2 \xrightarrow{lo} k$, which implies $k = k_2$ and $\mathsf{R}(k, f_2)$. □

If all the key writes are performed by a single process, key write operations are totally ordered and they can be given increasing sequence numbers. Intuitively, condition (SW-KM) requires that the file write operations are ordered in increasing order of the encryption key sequence numbers.

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 a generic consistency property 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.

In order for the encrypted file object to be $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$-consistent, we need to construct an (extended) serialization set S that is enc-legal (see Definition 6). In the proof of the theorem, we need to separate the case when C_2 belongs to $\mathcal{C}_\mathcal{O}$ from the case when C_2 belongs to $\mathcal{C}_\mathcal{F}$. For C_2 in $\mathcal{C}_\mathcal{O}$ we need to construct an enc-legal serialization set of the history of low-level operations, whereas for C_2 in $\mathcal{C}_\mathcal{F}$ an enc-legal extended serialization set is required.

Furthermore, we need to distinguish the case of file access protocols with consistency in class $\mathcal{C}_\mathcal{O}^+$, when there exists a legal serialization of the file operations. From Definition 6, in order to prove enc-consistency of H_l, we need to construct an enc-legal serialization with all the key and write operations. This implies that there must be a serialization for the key operations, as well. Thus, if the consistency of the file access protocol is in class $\mathcal{C}_\mathcal{O}^+$, we require that the consistency of the key distribution protocol belongs to $\mathcal{C}_\mathcal{O}^+$, as well.

Theorem 2. *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., $\mathsf{KM}_{\mathsf{C}_1,\mathsf{C}_2}(H)$.

Proof. First we assume that H is $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$-consistent. From Definition 6, it follows that there exists an enc-legal serialization (or serialization set or extended serialization set) S of H_l such that $S|\mathsf{KEY}$ is C_1-consistent and $S|\mathsf{FILE}$ is C_2-consistent. Consider $S_F = S|\mathsf{FILE}$, which is a C_2-consistent serialization (or serialization set or extended serialization set) of $H_l|\mathsf{FILE}$. We prove that conditions (KM_1) and (KM_2) are satisfied for S_F.

1. Let f_1 and f_2 be two file write operations such that $f_1 \xrightarrow{S_F} f_2$, and let k_1 and k_2 be their associated key write operations. As S is enc-legal, it follows that $k_1 \xrightarrow{S} k_2$. $S|\mathsf{KEY}$ is C_1-consistent, and the fact that $k_1 \xrightarrow{S|\mathsf{KEY}} k_2$ implies that it is not possible to have $k_2 \xrightarrow{\mathsf{C}_1, H_l|\mathsf{KEY}} k_1$. This proves condition (KM_1).

2. Let f_1, f_2 and f_3 be three file write operations and k a key write operation such that $f_1 \xrightarrow{S} f_2 \xrightarrow{S} f_3$, $\mathsf{R}(k, f_1)$ and $\mathsf{R}(k, f_3)$. It follows that key write k is the closest key write operation before f_2 in S. The fact that S is enc-legal implies that the value of operation k is used to encrypt the file content written in f_2, and thus $\mathsf{R}(k, f_2)$. This proves condition (KM_2).

In the reverse direction, we distinguish three cases, depending on the class the consistency C_2 belongs to:

1. Both $\mathsf{C}_1, \mathsf{C}_2$ belong to $\mathcal{C}^+_{\mathcal{O}}$. We construct an enc-legal serialization of H_l that respects the four conditions from Definition 6 in four steps. We construct first a serialization S of H_l that contains all the file operations and that respects C_2-consistency. Then, we include the key writes into this serialization in an order consistent with C_1. Thirdly, we include the key read operations into S to preserve the legality of key operations and the local ordering with the file operations. Finally, we also need to prove that S is enc-legal.

First step. As H_l is a key-monotonic history, it follows from Definition 8 that there exists a C_2-consistent serialization S_F of $H_l|\mathsf{FILE}$ that respects conditions (KM_1) and (KM_2). We include into serialization S all the file operations ordered in the same order as in S_F.

Second step. From condition (KM_2) in the definition of $\mathsf{KM}_{\mathsf{C}_1,\mathsf{C}_2}$, the file write operations in serialization S_F that are dependent on different keys are not interleaved. We can thus insert a key write operation in serialization S before the first file write operation that is associated with that key write (if such a file operation exists).

From the first condition in $\mathsf{KM}_{\mathsf{C}_1,\mathsf{C}_2}$, we can prove that $S|\mathsf{KEY}$ (that contains only key writes used in the file operations from S) is C_1-consistent. Assume, by contradiction, that $S|\mathsf{KEY}$ is not C_1-consistent. Since C_1 is orderable, there exist two key write operations, k_1 and k_2, such that $k_1 \xrightarrow{S} k_2$ and $k_2 \xrightarrow{\mathsf{C}_1, H_l|\mathsf{KEY}} k_1$. From the way we included the key writes into S, there exists two file write operations f_1 and f_2 such that $k_1 \xrightarrow{S} f_1 \xrightarrow{S} k_2 \xrightarrow{S} f_2$. But $f_1 \xrightarrow{S} f_2$ and (KM_1) imply that it is not possible to have $k_2 \xrightarrow{\mathsf{C}_1, H_l|\mathsf{KEY}} k_1$, which is a contradiction.

We have omitted from S all the key writes that are not used in file operations from S, but we need to insert those key write operations in S. We include the key writes that are not used in any file operations in S to preserve the $\overset{\mathsf{C}_1, H_l|\mathsf{KEY}}{\longrightarrow}$ order of key operations. If an unused key write operation needs to be added between key writes k_1 and k_2, then it is included immediately before k_2, so that it does not break the enc-legality of serialization S. This is possible as the key writes that we insert are not related by any constraints to the file operations in S.

Third step. We need to insert the key read operations to preserve the legality of key operations and the local ordering with the file operations in S. Let $k_1, \ldots, k_s$ be all the key write operations in the order they appear in the serialization S constructed in the first two steps. We include a key read that returns the value written by key write operation k_l between k_l and k_{l+1} (if $k_l = k_s$ is the last key write operation, then we include the key read after k_s in S). For key read operations and file operations that are in the same interval with respect to key writes, we preserve local ordering of operations. Assume, by contradiction, that this arrangement violates local ordering between operations in different key intervals. Only two cases are possible:

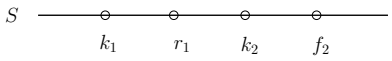

Figure 8: First case in which read r_1 can not be inserted into S

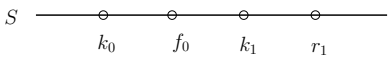

Figure 9: Second case in which read r_1 can not be inserted into S

- There exists a key read r_1 such that: the value returned by r_1 is written by key write operation k_1, r_1 is after file operation f_2 in local order, and f_2 belongs to a later key write interval than r_1, i.e., $k_1 \overset{S}{\longrightarrow} r_1 \overset{S}{\longrightarrow} k_2 \overset{S}{\longrightarrow} f_2$ (see Figure 8). If f_2 is a file read operation, then from the legality of file operations and the way we inserted the key write operations into S, it follows that there exists a file write operation f_2' such that $k_1 \overset{S}{\longrightarrow} r_1 \overset{S}{\longrightarrow} k_2 \overset{S}{\longrightarrow} f_2' \overset{S}{\longrightarrow} f_2$ and $\mathsf{R}(k_2, f_2')$.

  We can thus assume, w.l.o.g., that f_2 is a file write operation and $\mathsf{R}(k_2, f_2)$. From $\mathsf{R}(k_2, f_2)$ it follows that either $k_2 \overset{lo}{\longrightarrow} f_2$ or there exists a key read operation r_2 that returns the value written by k_2 and $r_2 \overset{lo}{\longrightarrow} f_2$.

  In the first case, $k_2 \overset{lo}{\longrightarrow} f_2$ and $f_2 \overset{lo}{\longrightarrow} r_1$ imply $k_2 \overset{lo}{\longrightarrow} r_1$. We inserted the key read operations into S to preserve the local order of key operations. Then, in serialization S, k_2 should be ordered before r_1. But $k_1 \overset{S}{\longrightarrow} r_1 \overset{S}{\longrightarrow} k_2$ and this represents a contradiction.

  In the second case, $r_2 \overset{lo}{\longrightarrow} f_2$ and $f_2 \overset{lo}{\longrightarrow} r_1$ imply $r_2 \overset{lo}{\longrightarrow} r_1$. For the same reason as above, r_2 should be ordered before r_1 in S. But $k_1 \overset{S}{\longrightarrow} r_1 \overset{S}{\longrightarrow} k_2 \overset{S}{\longrightarrow} r_2$ and this represents a contradiction.

- A file operation f_0 that belongs to an earlier key write interval than key read r_1 follows r_1 in the local order, i.e., $k_0 \overset{S}{\longrightarrow} f_0 \overset{S}{\longrightarrow} k_1 \overset{S}{\longrightarrow} r_1$ (see Figure 9). If f_0 is a file read operation, then from the legality of file operations and the way we inserted the key write operations into S, it

follows that there exists a file write operation f_0' such that $k_0 \xrightarrow{S} f_0' \xrightarrow{S} f_0 \xrightarrow{S} k_1 \xrightarrow{S} r_1$ and $\mathsf{R}(k_0, f_0')$.

We can thus assume, w.l.o.g., that f_0 is a file write operation and $\mathsf{R}(k_0, f_0)$. From $\mathsf{R}(k_0, f_0)$ it follows that either $k_0 \xrightarrow{lo} f_0$ or there exists a key read operation r_0 that returns the value written by k_0 and $r_0 \xrightarrow{lo} f_0$.

In the first case, $k_0 \xrightarrow{lo} f_0$, $r_1 \xrightarrow{lo} f_0$ and $\mathsf{R}(k_0, f_0)$ imply that $r_1 \xrightarrow{lo} k_0 \xrightarrow{lo} f_0$ (operation f_0 uses the latest key value read or written to encrypt the file value). Because S preserves the local order among key operations, r_1 should be ordered before k_0 in S. But this contradicts $k_0 \xrightarrow{S} k_1 \xrightarrow{S} r_1$.

In the second case, $r_0 \xrightarrow{lo} f_0$, $r_1 \xrightarrow{lo} f_0$ and $\mathsf{R}(k_0, f_0)$ imply that $r_1 \xrightarrow{lo} r_0 \xrightarrow{lo} f_0$. Because S preserves the local order among key operations, r_1 should be ordered before r_0 in S. On the other hand, the legality of S implies that $k_0 \xrightarrow{S} r_0 \xrightarrow{S} k_1 \xrightarrow{S} r_1$, but this contradicts the fact that r_1 should be ordered before r_0.

Fourth step. We need to prove that serialization S is enc-legal. From the way we included the key write operations into S in the second step, it follows that for any file write operation f_w, there exists a key write operation $k(f_w)$ such that $k(f_w) \xrightarrow{S} f_w$ and there does not exist another file operation k' with $k(f_w) \xrightarrow{S} k' \xrightarrow{S} f_w$. Moreover the key value written by $k(f_w)$ is used to encrypt the value written by f_w. This proves the enc-legality of S.

Summary. The serialization S respects the conditions from Definition 6 for $\mathsf{C}_2^{\mathsf{enc}}$-consistency:

1. S is enc-legal as proved in the fourth step;
2. $S|\mathsf{KEY}$ is C_1-consistent as proved in the second and third step;
3. $S|\mathsf{FILE} = S_F$ is C_2-consistent.
4. S respects local ordering between operations on the same object, as $S|\mathsf{KEY}$ and $S|\mathsf{FILE}$ respect local ordering. S respects local ordering between key writes and file operations from the construction of S. Additionally, the key reads are inserted to respect local ordering with file operations.

2. C_2 belongs to $\mathcal{C}_{\mathcal{O}}$, but not to $\mathcal{C}_{\mathcal{O}}^+$. The proof for this case proceeds similarly to the proof of the previous case with the difference that a serialization set $S = \{S_{p_i}\}_i$ needs to be constructed. We do not give here the full proof, but we only highlight the differences from the previous proof:

1. In the first step, from Definition 8, there exists a serialization set $S_F = \{S_{p_i}^F\}_i$ that respects conditions (KM_1) and (KM_2). File operations from each $S_{p_i}^F$ are included in the same order in S_{p_i}.
2. In the second step, we need to insert the key write operations in all serializations S_{p_i}. We can similarly prove that $S_{p_i}|\mathsf{KEY}$ is C_1-consistent for all i.

3. In the third step, we only need to insert in serialization S_{p_i} the key reads done by process p_i.

4. In the fourth step, we can prove that in each serialization S_{p_i} the closest key write before a file write is writing the key value used to encrypt the file value written by the file write operation.

3. C_2 belongs to $\mathcal{C}_{\mathcal{F}}$. The proof for the third case is similar to the proofs of the previous two cases, with the difference that an extended serialization set $S = \{S_{p_i}\}_i$ including the key and file operations needs to be constructed from a forking serialization set $S_F = \{S^F_{p_i}\}_i$ of the file operations. □

Discussion. Theorem 2 gives necessary and sufficient conditions for an encrypted file object to be $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$-consistent. We can easily generalize the theorem to give necessary and sufficient conditions for a collection of encrypted file objects or an encrypted file system to be $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$-consistent. The necessary and sufficient condition in this case for a history of high-level operations to be consistent is that the history of low-level operations restricted to each file object is key-monotonic with respect to its corresponding encryption key.

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:

1. 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., Farsite [1], SNAD [27] and SiRiUS [15], might not give an acceptable level of consistency. 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 $\mathcal{C}_{\mathcal{F}}$.

2. 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 [25] in the next section. An interesting problem is to design implementations of consistent encrypted file objects that do not require the file operations to keep track of the corresponding key operations.

Moreover, the framework offered by Theorem 2 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 2, 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 next section demonstrates the usability of our framework.

7 A Fork-Consistent Encrypted File Object

In this section, we apply our techniques to give an example of an encrypted file system that is fork consistent. It has been shown [25] that it is possible to construct a fork consistent file system even when the file server is potentially Byzantine. We use the SUNDR protocol [25] together with our main result, Theorem 2, to construct a fork consistent *encrypted* file system. For simplicity, we present the protocol for only one encrypted file object, but our protocols can be easily adapted to encrypted file systems.

System model. Users interact with the storage server to perform read and write operations on file objects. A file owner performs write operations on the key object associated with the file it owns, and users that have access permissions to the file can read the key object to obtain the cryptographic key for the file. There is, thus, a single writer to any key object, but multiple readers. Each key write operation can be assigned a unique sequence number, which is the total number of writes performed to that key object.

In our model, we store in a key object the key value and the key sequence number. For a file object, we also store the sequence number of the key used to encrypt the file value. We modify the write operation for both the FILE and KEY objects to take as an additional input the key sequence number. Similarly, the read operation for both the FILE and KEY objects returns the key sequence number (in addition to the object content).

Several cryptographic primitives are used in our example application:

1. For *confidentiality*, a symmetric encryption scheme $\mathcal{E}$ is used to encrypt files. $\mathcal{E}$ consists of three algorithms: a randomized key generation algorithm $\mathsf{GEN}(\cdot)$ that outputs an encryption key, a randomized encryption algorithm $\mathsf{Enc}_k(m)$ that outputs the encryption of a given message m with key k, and a deterministic decryption algorithm $\mathsf{Dec}_k(c)$ that decrypts a ciphertext c with key k. The correctness property requires that $\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = m$, for all keys k generated with the GEN algorithm and all messages m from the encryption domain.

2. Signature schemes are used for *integrity*. A signature scheme consists of three algorithms: a randomized key generation algorithm $\mathsf{GEN}(\cdot)$ that outputs a public key/secret key pair $(\mathsf{PK}, \mathsf{SK})$, a randomized or deterministic signing algorithm $\sigma \leftarrow \mathsf{Sign}_{\mathsf{SK}}(m)$ that outputs a signature of a given message m using the signing key SK, and a deterministic verification algorithm $\mathsf{Ver}_{\mathsf{PK}}(m, \sigma)$ that outputs a bit. A signature σ is *valid* on a message m if $\mathsf{Ver}_{\mathsf{PK}}(m, \sigma) = 1$. The correctness property requires that $\mathsf{Ver}_{\mathsf{PK}}(m, \mathsf{Sign}_{\mathsf{SK}}(m)) = 1$, for all key pairs $(\mathsf{PK}, \mathsf{SK})$ generated with the GEN algorithm and all messages m from the signature domain.

We assume that each user u of the file system has its own signing and verification keys, SK_u and PK_u, and there exists a public-key infrastructure that enables users to find the public keys for all other users of the file system.

In the description of our protocol, we distinguish three separate components: the implementation of high-level operations provided by the storage interface, the file access protocol and the key distribution protocol. We give the details about the implementation of the high-level operations and the file access protocol. For key distribution, any single-writer protocol that implements a sequential consistent shared object can be used (e.g., [4]), and we leave here the protocol unspecified. Finally, we prove the consistency of the protocol using our main result.

7.1 Implementation of High-Level Operations

The storage interface consists of four high-level operations similar to those presented in Section 5. Their implementation is detailed in Figure 10.

1. In create_file(f), an encryption key for the file is generated using the GEN algorithm of the encryption scheme. At the same time, the key sequence number stored locally by the file owner in variable seq is incremented (the variable seq needs to be initialized to 0). The user encrypts the file content f with the newly generated key and, finally, writes both the KEY and FILE objects.

2. In write_encfile(f), the encryption key k and its sequence number seq are read first. Then the user encrypts the file content f with the key value read and writes the encrypted file and the key sequence number to the file object.

3. In $f \leftarrow$ read_encfile(), the values of the key and file objects and their sequence numbers are read in variables (k, seq) and (c, seq'), respectively. The user checks that c is encrypted with the key that has the same sequence number to the key read, and retries if it did not read the correct key value. Finally, ciphertext c is decrypted with key k, resulting in file content f.

4. In write_key, the file object is read and decrypted using read_encfile. Then, the file is reencrypted with a newly generated key using create_file. In order to guarantee that the latest version of a file is encrypted with the latest encryption key and that the **repeat** loop in the implementation of the procedure read_encfile terminates, this procedure needs to be executed atomically (i.e., in isolation from all other clients' operations).

The create_file and write_key procedures can only be executed by file owners upon file creation and the change of the file encryption key, respectively. The write_encfile and read_encfile procedures are executed by writers and readers of the file when they perform a write or read operation, respectively.

```
1.  procedure create_file(f):
2.      k ← GEN()                          /* generate a new key */
3.      seq ← seq + 1                      /* increment the key sequence number stored locally */
4.      c ← Enc_k(f)                       /* encrypt the file with the new key */
5.      KEY.write(k, seq)                  /* invoke write operation on key object */
6.      FILE.write(c, seq)                 /* invoke write operation on file object */

7.  procedure write_encfile(f):
8.      (k, seq) ← KEY.read()              /* invoke read operation on key object*/
9.      c ← Enc_k(f)                       /* encrypt the file with the key read */
10.     FILE.write(c, seq)                 /* invoke write operation on file object */

11. procedure read_encfile():
12.     repeat
13.         (k, seq) ← KEY.read()          /* invoke read operation on file object */
14.         (c, seq') ← FILE.read()        /* invoke read operation on key object */
15.     until seq = seq'                   /* check that the key read matches the key used to encrypt the file */
16.     f ← Dec_k(c)                       /* decrypt the file with the key read */
17.     return f

18. procedure write_key():
19.     f ← read_encfile()                 /* read and decrypt file content */
20.     create_file(f)                     /* generate a new key and encrypt the file with it */
```

Figure 10: The encrypted file protocol for client u

7.2 The File Access Protocol

We first describe the original SUNDR protocol [25] that constructs a fork consistent file system. We then present our modifications to the protocol for guaranteeing consistency of an encrypted file object implemented with the high-level operations given in the previous subsection.

The SUNDR protocol. In the SUNDR protocol, the storage server can be split into two components: a block store (denoted $\mathsf{S_{BS}}$) on which clients can invoke read and write operations on blocks and a consistency server $\mathsf{S_{CONS}}$ that is responsible for ordering the read and write operations to the block store in order to maintain fork consistency.

Both the client and the consistency server in the SUNDR protocol need to keep state. $\mathsf{S_{CONS}}$ keeps a version structure for each client, signed by the client, and each client keeps a local copy of its own version structure. In more detail, the consistency server's state includes a version structure list or VSL consisting of one signed version structure per user, denoted $v[u], u = 1, \ldots, U$ (U is the total number of users). Each version structure $v[u]$ is an array of version numbers for each client in the system: $v[u][j]$ is the version number of user j, as seen by user u. Version numbers for a user are defined as the total number of read and write operations performed by that user. There is a natural ordering relation on version structures: $v \leq w$ if $v[i] \leq w[i]$ for all $i = 1, \ldots, U$.

```
1.  FILE.write(c, seq):
2.     last_op ← write                          /* store locally the type of the operation */
3.     last_seq ← seq                           /* store locally the key sequence number */
4.     last_file ← c                            /* store locally the encrypted file c */
5.     v_max ← check_cons()                     /* execute the version update protocol with S_CONS */
6.     S_BS.write(c)                            /* write the encrypted file to the block store */

7.  FILE.read():
8.     last_op ← read
9.     v_max ← check_cons()                     /* store locally the type of the operation */
10.    c ← S_BS.read()                          /* execute the version update protocol with S_CONS*/
11.    if not check_int(c, v_max) then abort    /* read the encrypted file from the block store */
12.    return (c, v_max.seq)                    /* check the integrity of the encrypted file */

13. check_cons():
14.    (v[1], ..., v[U]) ← S_CONS.vs_request()  /* receive version structures of all users from S_CONS */
15.    verify the signatures on all v[i]        /* abort if any of the signatures does not verify */
16.    if v[u] ≠ vs, then abort                 /* check its own version structure */
17.    x[u] ← vs[u] + 1; x[j] ← v[j][j], ∀j ≠ u  /* create a new version structure and initialize it */
18.    x.int ← vs.int
19.    if last_op = write then                  /* for a write operation: */
20.       update_int(last_file, x)              /* update the integrity information in x */
21.       x.seq ← last_seq                      /* update the key sequence number in x */
22.    if (v[1], ..., v[U]) are not totally ordered or   /* ensures that VSL is totally ordered and x is
          ∃i = 1, ..., U : x ≤ v[i]             greater than all the version structures from VSL */
23.    then abort
24.    vs ← x                                   /* store x locally */
25.    S_CONS.vs_update(Sign_{K_u}(x))          /* send version structure to S_CONS */
26.    return v_max = max(v[1], ..., v[U])      /* return the maximum version structure of all users */
```

Figure 11: The file access protocol for client u

Each version structure v also contains some integrity information denoted $v.int$. In the SUNDR protocol, the integrity information is the root of a Merkle hash tree [26] of all the files the user owns and has access to. The integrity information is updated by every client that writes a file. At every read operation, the integrity information in the most recent version structure is checked against the file read from $\mathsf{S_{BS}}$. We assume that there exists two functions for checking and updating the integrity information for a file: check_int(c, v) that given an encrypted file c and a version structure v checks the integrity of c using the integrity information in v, and update_int(c, v) that updates the integrity information $v.int$ using the new encrypted file c. We do not give here the details of implementing these two functions.

A user u has to keep locally the latest version structure vs that it signed. The code for user u is in Figure 11. At each read or write operation, u first performs the check_cons protocol (lines 5 and 9) with the consistency server, followed by the corresponding read or write operation to the block store. SUNDR uses a mechanism to ensure that the check_cons protocol is executed atomically,

```
1. S_BS.write(c) from u:
2.     FILE ← c                 /* store c to the file object FILE*/

3. S_BS.read() from u:
4.     return FILE to u         /* return content of file object FILE to u */
```

Figure 12: The code for the block store $\mathsf{S_{BS}}$

```
1. S_CONS.vs_request() from u:
2.     return (msg_vsl, v[1], ..., v[U]) to u      /* send the signed VSL to u */

3. S_CONS.vs_update(Sign_{K_u}(x)) from u:
4.     verify the signature on x                   /* abort if the signature does not verify */
5.     if ∃i = 1, ..., U : x ≤ v[i] then abort     /* check that x is greater than all version structures */
6.     else v[u] ← x                               /* update the version structure for u */
```

Figure 13: The code for the consistency server $\mathsf{S_{CONS}}$

but we skipped this for clarity of presentation. The code for the block store and the consistency server is in Figures 12 and 13, respectively.

In the check_cons protocol, the client first performs the vs_request RPC with the consistency server to receive the list of version structures of all users. The client first checks the signatures on the version structures and checks that its own version structure matches the one stored locally (lines 15-16). Then, the client creates a new version structure x that contains the latest version number for each user (lines 17-20). In the new version structure, the client's version number is incremented by 1, and the integrity information is updated only for write operations. Finally, the client checks that the version structures are totally ordered and the new version structure created is the maximum of all (lines 21-22). This last check guarantees that the version structures for successive operations seen by each client are strictly increasing (with respect to the ordering relation defined for version structures). If any of the checks performed by the client fails, then the client detected misbehavior of $\mathsf{S_{CONS}}$ and it aborts the protocol. The client sends the newly created signed version structure to the consistency server through the vs_update RPC. $\mathsf{S_{CONS}}$ checks that this version structure is the maximum of all existing version structures (line 5 in Figure 13) to protect against faulty clients.

For more details and a proof of fork-consistency of this protocol, see [25].

Modifications to the SUNDR protocol. We include in the version structure v of user u a key sequence number $v.seq$, which is the most recent version of the key used by user u in a file operation. We extend the total order relation on version structures so that $v \leq w$ if $v[i] \leq w[i]$ for all $i = 1, \ldots, U$ and $v.seq \leq w.seq$. For each write operation, we need to update the key sequence number in the newly created version structure (line 20 in Figure 11). The key sequence numbers in

the version structures guarantee that the file operations are serialized according to increasing key sequence numbers.

7.3 Consistency Analysis

The file access protocol guarantees a forking serialization set for the file operations. Intuitively, the operations in a serialization for a process (which form a branch in the forking tree) have totally ordered version structures (by line 21 in Figure 11 and line 5 in Figure 13). We extend the version structures to include the key sequence numbers. The total order of the version structures implies that the file operations in a serialization for a process have increasing key sequence numbers. This will allow us to prove that a history of low-level operations resulting from an execution of the protocol is key-monotonic.

Proposition 3. *Let H_l be a history of low-level read and write operations on the key object* KEY *and file object* FILE *that is obtained from an execution of the protocol in Figure 10. If $H_l|$KEY is sequential consistent and the file access protocol is implemented with the protocol in Figure 11, then the execution of the protocol is* enc*-consistent.*

Proof. Conditions (1), (2), (3) and (4) from Theorem 2 are satisfied. To apply the theorem, we only need to prove that H_l is key-monotonic with respect to sequential consistency and fork consistency. In particular, it is enough to prove condition (SW-KM).

Let S be any fork consistent forking serialization set of $H_l|$FILE and f_1 and f_2 two file write operations such that $f_1 \xrightarrow{S} f_2$. Let v^1_{max} and v^2_{max} be the two version structures returned by the check_cons protocol when f_1 and f_2 are executed. These are the version structures denoted by x created in lines 17-20 of the protocol in Figure 11.

The protocol guarantees that $v^1_{max} \leq v^2_{max}$, which implies that $v^1_{max}.seq \leq v^2_{max}.seq$. Line 20 in the protocol from Figure 11 guarantees that $v^1_{max}.seq$ contains the sequence number of the key with which the encrypted file content written in operation f_1 is encrypted. Similarly, $v^2_{max}.seq$ contains the sequence number of the key with which the value written in operation f_2 is encrypted.

Let k_1 and k_2 be the key write operations such that $\mathsf{R}(k_1, f_1)$ and $\mathsf{R}(k_2, f_2)$. $v^1_{max}.seq \leq v^2_{max}.seq$ implies that $k_1 \xrightarrow{lo} k_2$ or $k_1 = k_2$, which is exactly what condition (SW-KM) demands. From Theorem 2, it follows that the execution of the protocol is $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$-consistent, where C_1 is sequential consistency and C_2 is fork consistency. ∎

8 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. We also demonstrate how to implement a consistent encrypted file object from a sequential consistent key distribution protocol and the fork consistent file access protocol from the SUNDR file system [25, 22].

Another contribution of this paper is to define two classes of consistency conditions that are meaningful for encrypted file systems over untrusted storage: the first class includes classical consistency conditions such as linearizability, causal consistency, PRAM 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. The two classes do not cover all the existing (or possible) consistency conditions, and another challenge raised by this work is to define additional classes that are meaningful for consistency in encrypted file systems.

References

[1] 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.

[2] 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.

[3] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 1(9):37–49, 1995.

[4] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.

[5] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared-memory system. In *Proc. IEEE COMPCON Conference*, pages 528–537. IEEE, 1993.

[7] M. Blaze. A cryptographic file system for Unix. In *Proc. First ACM Conference on Computer and Communication Security (CCS)*, pages 9–16. ACM, 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)*, pages 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)*, pages 173–186. Usenix, 1999.

[10] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for Unix. In *Proc. USENIX Annual Technical Conference 2001, Freenix Track*, pages 199–212, 2001.

[11] M. Dubois, C. Scheurich, and F.A. Briggs. Synchronization, coherence and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, 1988.

[12] R. Friedman, R. Vitenberg, and G. Chockler. On the composability of consistency conditions. *Information Processing Letters*, 86:169–176, 2002.

[13] K. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, 1999.

[14] 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*, pages 15–26, 1990.

[15] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. Network and Distributed Systems Security (NDSS) Symposium 2003*, pages 131–145. ISOC, 2003.

[16] J. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, 1989.

[17] M. Herlihy and J. Wing. Linearizability: A corretness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[18] 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.

[19] 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)*, pages 251–260. IEEE, 2001.

[20] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[21] 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*, 25(3):63–79, 1992.

[22] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository. In *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 121–136. Usenix, 2004.

[23] R. Lipton and J. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, 1988.

[24] 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)*, pages 124–139. ACM, 1999.

[25] D. Mazieres and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 108–117. ACM, 2002.

[26] R. Merkle. A cerified digital signature. In *Proc. Crypto 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer-Verlag, 1989.

[27] 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)*, pages 1–13, 2002.

[28] 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)*, pages 15–30, 2002.

[29] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[30] 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)*, pages 163–172. ACM, 1999.

[31] R. Vitenberg and R. Friedman. On the locality of consistency conditions. In *Proc. 17th International Symposium on Distributed Computing (DISC))*, pages 92–105, 2003.

[32] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, 2002.