Efficient Cryptographic Techniques for Securing Storage Systems

Alina Mihaela Oprea

CMU-CS-07-119

April 2007

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Christian Cachin, IBM Zurich Research Laboratory
Gregory Ganger
Michael K. Reiter, Chair
Dawn Song

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Copyright © 2007 Alina Mihaela Oprea

Keywords: storage security, tweakable encryption, data integrity, Merkle trees, key management in cryptographic file systems, lazy revocation, consistency of encrypted data.

Abstract

The growth of outsourced storage in the form of storage service providers underlines the importance of developing efficient security mechanisms to protect the data stored in a networked storage system. For securing the data stored remotely, we consider an architecture in which clients have access to a small amount of *trusted storage*, which could either be local to each client or, alternatively, could be provided by a client's organization through a dedicated server. In this thesis, we propose new approaches for various mechanisms that are currently employed in implementations of secure networked storage systems. In designing the new algorithms for securing storage systems, we set three main goals. First, security should be added by clients transparently for the storage servers so that the storage interface does not change; second, the amount of trusted storage used by clients should be minimized; and, third, the performance overhead of the security algorithms should not be prohibitive.

The first contribution of this dissertation is the construction of novel space-efficient integrity algorithms for both block-level storage systems and cryptographic file systems. These constructions are based on the observation that block contents typically written to disks feature low entropy, and as such are efficiently distinguishable from uniformly random blocks. We provide a rigorous analysis of security of the new integrity algorithms and demonstrate that they maintain the same security properties as existing algorithms (e.g., Merkle tree). We implement the new algorithms for integrity checking of files in the EncFS cryptographic file system and measure their performance cost, as well as the amount of storage needed for integrity and the integrity bandwidth (i.e., the amount of information needed to update or check the integrity of a file block) used. We evaluate the block-level integrity algorithms using a disk trace we collected, and the integrity algorithms for file systems using NFS traces collected at Harvard university.

We also construct efficient key management schemes for cryptographic file systems in which the re-encryption of a file following a user revocation is delayed until the next write to that file, a model called *lazy revocation*. The encryption key evolves at each revocation and we devise an efficient algorithm to recover previous encryption keys with only logarithmic cost in the number of revocations supported. The novel key management scheme is based on a binary tree to derive the keys and improves existing techniques by several orders of magnitude, as shown by our experiments.

Our final contribution is to analyze theoretically the consistency of encrypted shared file objects used to implement cryptographic file systems. We provide sufficient conditions for the realization of a given level of consistency, when concurrent writes to both the file and encryption key objects are possible. We show that the consistency of both the key

distribution and the file access protocol affect the consistency of the encrypted file object that they implement. To demonstrate that our framework simplifies complex proofs for showing the consistency of an encrypted file, we provide a simple implementation of a fork consistent encrypted file and prove its consistency.

Acknowledgments

Always try to climb very high to be able to see far away.
Constantin Brancusi

I deeply thank my advisor, Mike Reiter, for giving me the first insight into the field of applied cryptography, opening new research perspectives to me, and closely guiding my work that led to this dissertation. His optimism and constant encouragement throughout the years always motivated me to go on.

I am very grateful to Christian Cachin for mentoring me during the summer I spent at IBM Research in Zurich and suggesting the project on key management schemes for lazy revocation that became Chapter 4 of this dissertation. I enjoyed and learned a lot from our engaging discussions and arguments on different research topics. I also thank Christian for carefully proofreading this dissertation and providing detailed comments and suggestions. I am greatly indebted to my other committee members, Greg Ganger and Dawn Song, for their suggestions that contributed to the improvement of this dissertation.

The years I have spent at Carnegie Mellon would not have been so enjoyable without the people from the security group. I would particularly like to thank Asad Samar, Charles Fry, Lea Kissner and Scott Garriss for numerous discussions on various research topics and beyond. I thank several people from the group for providing me statistics on the file contents in their home directories that were reported in Chapter 3 of this dissertation.

I also thank my dearest friends, Oana Papazoglu and Irina Dinu, for always being close when I needed them. My friends from Pittsburgh also contributed to my general well-being and happiness during all these years: Cristina Canepa, Adina Soaita, Cristina Arama and Emil Talpes.

Finally, I owe everything to my family. My grandparents raised me during early childhood. My parents have always believed in me and encouraged me to pursue my own path in life. Florin has been my constant support and source of energy during the last six years. And last but not least, Andrei's genuine smile helps me climb a bit higher everyday.

Contents

1 Introduction **1**

1.1 Securing Networked Storage Systems 2

1.2 Contributions 5

1.2.1 Efficient Integrity Algorithms for Encrypted Storage Systems . . 5

1.2.2 Key Management Schemes for Lazy Revocation 7

1.2.3 Consistency of Encrypted Files 9

1.3 Organization 11

2 Block-Level Integrity in Encrypted Storage Systems **13**

2.1 Preliminaries 14

2.1.1 Tweakable Enciphering Schemes 14

2.1.2 Everywhere Second Preimage Resistant Hash Functions 14

2.1.3 Pseudorandom Functions 15

2.1.4 Security of Message-Authentication Codes 16

2.1.5 Chernoff Bounds 16

2.2 System Model 17

2.3 Block Write Counters 18

2.4 Length-Preserving Stateful Encryption 19

2.5 Notions of Integrity for Encrypted Block-Level Storage 20

2.6 Constructions of Integrity Schemes for Encrypted Block-Level Storage . . 22

2.6.1 Hash scheme HASH-BINT 22

2.6.2 Schemes Based on a Randomness Test . . . 23
2.6.3 The Compression Scheme COMP-BINT . . . 28
2.7 The Entropy Statistical Test . . . 30
2.8 Evaluation . . . 33
2.9 Related Work . . . 35

3 Integrity in Cryptographic File Systems with Constant Trusted Storage 37
3.1 Preliminaries . . . 38
3.2 System Model . . . 41
3.3 Write Counters for File Blocks . . . 43
3.4 Integrity Constructions for Encrypted Storage . . . 46
3.4.1 The Merkle Tree Construction MT-FINT . . . 46
3.4.2 The Randomness Test Construction RAND-FINT . . . 48
3.4.3 The Compress-and-Hash Construction COMP-FINT . . . 51
3.5 Implementation . . . 53
3.6 Performance Evaluation . . . 54
3.6.1 The Impact of File Block Contents on Integrity Performance . . . 56
3.6.2 The Impact of File Access Patterns on Integrity Performance . . . 61
3.6.3 File Content Statistics . . . 62
3.6.4 Amount of Trusted Storage . . . 63
3.7 Discussion . . . 63
3.8 Related Work . . . 65

4 Lazy Revocation in Cryptographic File Systems 69
4.1 Preliminaries . . . 70
4.1.1 Pseudorandom Generators . . . 70
4.1.2 Trapdoor Permutations . . . 70
4.1.3 CPA-Security of Symmetric Encryption . . . 71
4.1.4 Signature Schemes and Identity-Based Signatures . . . 72

4.2 Formalizing Key-Updating Schemes 74
4.2.1 Definition of Key-Updating Schemes 74
4.2.2 Security of Key-Updating Schemes 75
4.3 Composition of Key-Updating Schemes 77
4.3.1 Additive Composition 77
4.3.2 Multiplicative Composition 80
4.4 Constructions of Key-Updating Schemes 83
4.4.1 Chaining Construction (CKU) 83
4.4.2 Trapdoor Permutation Construction (TDKU) 86
4.4.3 Tree Construction (TreeKU) 88
4.5 Performance of the Constructions 93
4.6 Experimental Evaluation of the Three Constructions 94
4.7 Cryptographic Primitives in the Lazy Revocation Model 97
4.7.1 Symmetric Encryption Schemes with Lazy Revocation (SE-LR) . 97
4.7.2 Message-Authentication Codes with Lazy Revocation (MAC-LR) 101
4.7.3 Signature Schemes with Lazy Revocation (SS-LR) 104
4.7.4 Applications to Cryptographic File Systems 108
4.8 Related Work 109

5 On Consistency of Encrypted Files 113
5.1 Preliminaries 114
5.1.1 Basic Definitions and System Model 114
5.1.2 Eventual Propagation 114
5.1.3 Ordering Relations on Operations 115
5.2 Classes of Consistency Conditions 115
5.2.1 Orderable Conditions 116
5.2.2 Forking Conditions 117
5.2.3 Examples 118

5.3 Definition of Consistency for Encrypted Files 120
5.4 A Necessary and Sufficient Condition for the Consistency of Encrypted File Objects 123
5.4.1 Dependency among Values of Key and File Objects 123
5.4.2 Key-Monotonic Histories 124
5.4.3 Simpler Conditions for Single-Writer Key Object 125
5.4.4 Obtaining Consistency for Encrypted File Objects 127
5.5 A Fork-Consistent Encrypted File Object 133
5.5.1 Implementation of High-Level Operations 134
5.5.2 The File Access Protocol 134
5.5.3 Consistency Analysis 138
5.6 Related Work 139

6 Conclusions 143

Bibliography 147

List of Figures

1.1 Network storage architecture. 2

2.1 Experiments defining the security of pseudorandom function family PRF. 15

2.2 The UpdateCtr and GetCtr algorithms. 18

2.3 Implementing a length-preserving stateful encryption scheme with write counters. 20

2.4 Experiments for defining storage integrity. 21

2.5 Scheme HASH-BINT. 23

2.6 Scheme SRAND-BINT. 25

2.7 Scheme RAND-BINT. 28

2.8 Scheme COMP-BINT. 29

2.9 The entropy test $\texttt{IsRand}_{b,\mathsf{TH}_{\mathsf{Ent}}}(M)$. 31

2.10 Block access distribution. 34

2.11 Client storage for integrity for the three replay-secure algorithms. 35

3.1 Merkle tree for a file with 6 blocks on the left; after block 7 is appended on the right. 39

3.2 The UpdateTree, CheckTree, AppendTree and DeleteTree algorithms for a Merkle tree T. 40

3.3 Cumulative distribution of number of writes per block. 44

3.4 Cumulative distribution of number of counter intervals in files. 44

3.5 The AuthCtr and CheckCtr algorithms for counter intervals. 46

3.6 The Update, Check, Append and Delete algorithms for the MT-FINT construction. . . . 47
3.7 The Update, Check, Append and Delete algorithms for the RAND-FINT construction. . . . 49
3.8 The DelIndexTree algorithm for a tree T deletes the hash of block i from T and moves the last leaf in its position, if necessary, to not allow holes in the tree. . . . 50
3.9 The Update, Check, Append and Delete algorithms for the COMP-FINT construction. . . . 52
3.10 Prototype architecture. . . . 54
3.11 Micro-benchmarks for text and compressed files. . . . 57
3.12 Evaluation for low-entropy files (text, object and executable files). . . . 59
3.13 Evaluation for high-entropy files (image and compressed files). . . . 60
3.14 Running time, average integrity bandwidth and storage for integrity of CMC-Entropy and AES-Compression relative to AES-Merkle. Labels on the graphs represent percentage of random-looking blocks. . . . 61
3.15 Number of files for which the counter intervals are stored in the untrusted storage space. . . . 64

4.1 Experiments defining the security of pseudorandom generator G. . . . 70
4.2 Experiments defining the CPA-security of encryption. . . . 72
4.3 Experiments defining the security of key-updating schemes. . . . 76
4.4 The additive composition of KU_1 and KU_2. . . . 78
4.5 The multiplicative composition of KU_1 and KU_2. . . . 81
4.6 The $\mathsf{Derive}(t, \mathsf{CS}_t)$ and $\mathsf{Extract}(t, \mathsf{UK}_t, i)$ algorithms of the chaining construction. . . . 84
4.7 The $\mathsf{Update}(t, (P_t, L_t))$ algorithm. . . . 89
4.8 The $\mathsf{Extract}(t, \mathsf{UK}_t, i)$ algorithm. . . . 91
4.9 Evaluation of the chaining scheme. . . . 95
4.10 Evaluation of the trapdoor scheme. . . . 95
4.11 Evaluation of the tree scheme. . . . 95

5.1 Classes of consistency conditions. 119
5.2 Linearizable history. 119
5.3 Causal consistent history. 119
5.4 FORK-Linearizable history. 119
5.5 A forking tree for the history. 120
5.6 A history that is not key-monotonic. 124
5.7 A history that does not satisfy condition (KM_2). 126
5.8 First case in which read r_1 cannot be inserted into S. 129
5.9 Second case in which read r_1 cannot be inserted into S. 129
5.10 The encrypted file protocol for client u. 135
5.11 The file access protocol for client u. 137
5.12 The code for the block store S_{BS}. 138
5.13 The code for the consistency server $\mathsf{S}_{\mathsf{CONS}}$. 138

List of Tables

3.1 Average storage per file for two counter representation methods. 45
3.2 File set characteristics. 56
3.3 NFS Harvard trace characteristics. 61
3.4 Statistics on file contents. 63

4.1 Worst-case time and space complexities of the constructions for T time intervals. *Note: the amortized complexity of $\mathsf{Update}(t, \mathsf{CS}_t)$ in the binary tree scheme is constant. 93
4.2 Evaluation of the three constructions for 1024 revocations. 96

Chapter 1

Introduction

As enterprise storage needs grow at approximately 50% per year, companies have to invest in storage resources that are often under-utilized. Moreover, there is a large increase in the cost of managing huge amounts of data. To overcome these difficulties, many small and medium-size companies choose to outsource their data to third party storage service providers that provide on-demand storage space and storage management services. Data is transferred from the storage service provider using a high-speed networked storage solution such as a storage-area network (i.e., SAN) or network-attached storage (i.e., NAS). The advantage of using a storage service provider is that companies only pay for the amount of storage used at a certain time and it is easy to gradually increase the amount of storage used.

Recently, companies also have to comply with a growing number of federal and state regulations that require that, for instance, financial information and patient records are handled securely. One obstacle in the adoption of outsourced storage is the need to protect the data stored remotely from disclosure or corruption. Thus, it is necessary to develop new security mechanisms to protect the data stored remotely in a networked storage system. As the clients cannot rely on the storage servers for security guarantees, security should be implemented on the client side in a manner transparent to the storage servers.

For securing the data stored remotely, we consider a model depicted in Figure 1.1 in which clients have access to a small amount of *trusted storage*, which could either be local to each client or, alternatively, could be provided by a client's organization through a dedicated server. A similar architecture that uses a dedicated meta-data server to store all the filesystem meta-data, as well as some integrity information and cryptographic keys for security, has been prototyped in a secure version of the IBM StorageTank distributed file system (Pletka and Cachin [2006]). A major challenge in designing a client-side security

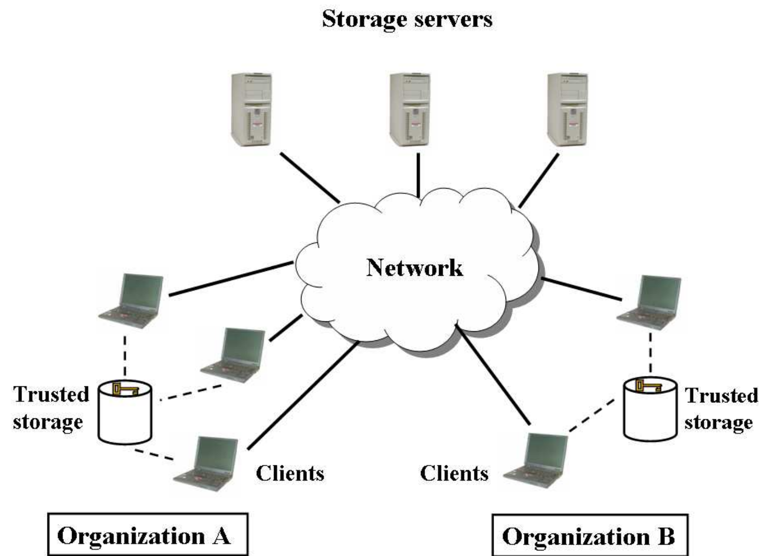

Figure 1.1: Network storage architecture.

solution for such a networked storage architecture is to preserve the traditional storage interface and minimize the amount of trusted storage used, while not incurring a prohibitive performance overhead.

1.1 Securing Networked Storage Systems

Security can be added at different layers and granularity levels in a networked storage system. One approach is to directly encrypt individual disk blocks in a SAN, for instance, but it is more common to implement security at the layer of files (in file systems) or objects (in object-based storage devices).

Cryptographic file systems. Cryptographic file systems (e.g., (Fu [1999], Cattaneo et al. [2001], Miller et al. [2002], Goh et al. [2003], Kallahalla et al. [2003], Li et al. [2004])) augment the file system interface with client-side cryptographic operations that protect the file secrecy and integrity against the compromise of the file store and attacks on the network. Several cryptographic file systems support only encryption of files, but not integrity protection (e.g., CFS (Blaze [1993]), NCryptFS (Wright et al. [2003b], Windows EFS (Russinovich [1999])), while others protect the integrity of files, but store data in clear on the storage servers (e.g., SFS (Fu et al. [2002]), SUNDR (Li et al. [2004])). Some

cryptographic file systems are local (e.g., CFS (Blaze [1993]), EncFS (Gough [2003])) and others are distributed (e.g., SFS (Fu et al. [2002]), Plutus (Kallahalla et al. [2003])). Distributed cryptographic file systems enable file sharing among their users and need to implement a key management scheme for the distribution of the cryptographic keys to authorized users.

Object-based storage. Object-based storage (or OSD) (T10 [2004]) is a new storage standard that evolves the storage interface from fixed size disk blocks to variable size objects. A storage object is a set of disk blocks that are aggregated for ease of management. The OSD standard also includes a security model (Factor et al. [2005]) based on capabilities for restricting access to storage only to authorized users. Future extensions of the standard will also include privacy protection and integrity.

Even when security is added at the layer of files or objects, it is a common practice to add cryptographic protections at the granularity of fixed-size data blocks with typical values of 1KB or 4KB. Several mechanisms are needed to implement security at either the block, object or file layer, in particular primitives for data confidentiality and integrity, access control methods to enable sharing of information, key management schemes that support user revocation, and methods for guaranteeing consistency of encrypted data. We detail below each of these security mechanisms.

Data Confidentiality. In order to prevent leakage of information to unauthorized parties, clients should encrypt the data stored remotely. For that, standard block cipher algorithms (e.g., AES (FIPS197 [2001])) or tweakable cipher implementations (e.g., CMC (Halevi and Rogaway [2003]), EME (Halevi and Rogaway [2004]), XCB (McGrew and Fluhrer [2004])) are suitable. Both the encryption and the decryption algorithms of a tweakable cipher take as input (in addition to the secret key and message) a public parameter, called a *tweak*, used for variability.

The block size of a standard cipher algorithm (e.g., AES) is usually small (16-32 bytes) compared to the size of a data block secured by clients. To encrypt a large data block, a block cipher in one of the standard modes of operations (e.g., CBC (FIPS81 [1980])) could be used, but this has the effect of increasing the length of the encrypted block. In contrast, tweakable ciphers are length-preserving, so that block boundaries do not shift or need to be adjusted as a result of encryption. As such, they are compliant with the requirement of adding security transparently to the storage servers. In fact, they were designed following a call for algorithms for block-level encryption by the IEEE Security In Storage Working Group and are currently considered for standardization.

Data Integrity. The data stored on the storage servers is vulnerable to unauthorized modification. A particularly interesting attack against data integrity is a *replay attack*, in which stale versions of data are returned to clients' read operations. To protect against modification of data, secure hashes of data blocks could be stored in the trusted storage space. A standard way of reducing the amount of trusted storage required for integrity is to construct a Merkle tree (Merkle [1989]) over multiple data blocks and authenticate only the root of the tree in trusted storage.

Access Control. An access control mechanism needs to be employed to restrict access to the stored data to only authorized users. A natural access control method for a storage system is to distribute the appropriate cryptographic keys (e.g., for encryption, message-authentication codes or digital signatures) only to users that have the appropriate access permissions. However, the use of cryptography for access control does not prevent against data destruction and overwriting by malicious clients. An alternative method uses capabilities (Gobioff et al. [1997]), but assumes intelligent storage devices that can check the validity of the capabilities presented by users before performing operations on data.

Key Management and User Revocation. Key management solutions in storage systems range from fully centralized key distribution using a trusted key server (Fu [1999]) to completely decentralized key distribution done by the storage system users (Kubiatowicz et al. [2000], Kallahalla et al. [2003]). Cryptographic keys are usually aggregated for multiple blocks into *a set of blocks* such that access permissions and ownership are managed at the level of these sets. The users who have access to the blocks in a set form a group, managed by the *group owner*. In the majority of cryptographic file system implementations, keys are assigned at the granularity of individual files or directories, and in object-based storage systems keys are assigned at the granularity of objects.

Initially, the cryptographic keys for a set of blocks are distributed to all users that have access permissions to the set. Upon revocation of a user from the group, the keys for the set needs to be changed to prevent access by revoked users to the data. Additionally, the cryptographic information for each block in the set (either an encryption of the block or some integrity information) has to be recomputed with the new cryptographic key.

There are two revocation models, depending on when the cryptographic information for the blocks in a set is updated, following a user revocation. In an *active revocation* model, all cryptographic information for the blocks in a set is immediately recomputed after a revocation takes place. This is expensive and might cause disruptions in the normal operation of the storage system. In the alternative model of *lazy revocation*, the informa-

tion for each data block in the set is recomputed only when that block is modified for the first time after a revocation (Fu [1999]).

Consistency of Encrypted Data. When multiple users are allowed concurrent access and modification of the remotely stored data, consistency of the encrypted data needs to be maintained. Many different consistency models have been defined and implemented for shared objects, ranging from strong conditions such as linearizability (Herlihy and Wing [1990]) and sequential consistency (Lamport [1979]), to loose consistency guarantees such as causal consistency (Ahamad et al. [1995]) and PRAM consistency (Lipton and Sandberg [1988]). When files are encrypted in cryptographic file systems, their consistency depends not only on the consistency of the file objects, but also on the consistency of the key object used to encrypt file contents. New challenges arise in defining consistency of encrypted files (starting from existing consistency conditions) and providing consistent implementations.

1.2 Contributions

In this thesis, we demonstrate that storage systems can be secured using novel, provably secure cryptographic constructions that incur low performance overhead and minimize the amount of trusted storage used. For that, we propose new approaches for two different mechanisms used in securing networked storage systems. First, we provide new, space-efficient integrity algorithms for both block-level encrypted storage systems and cryptographic file systems. Second, we construct new key management schemes for cryptographic file systems adopting lazy revocation. In addition, we study the consistency of encrypted file objects when multiple clients are allowed concurrent access to files in a cryptographic file system. We describe in detail each of the three contributions below.

1.2.1 Efficient Integrity Algorithms for Encrypted Storage Systems

Integrity in block-level storage systems. In Chapter 2 we present new methods to provide block-level integrity in encrypted storage systems (Oprea et al. [2005]). To preserve the length of the encrypted blocks sent to the storage servers, clients encrypt the blocks with a *length-preserving stateful encryption scheme* (a type of encryption we define and for which we provide two constructions). We present cryptographic definitions for integrity in this setting extending the security definitions for authenticated encryption (Bellare and Namprempre [2000]). One of the definitions formally expresses the notion that if the client

returns a data block in response to a read request for an address, then the client previously wrote that data block to that address. The second definition is stronger by providing defense against replay attacks; informally, it expresses the notion that if the client returns a data block in response to a read request for an address, then that data block is the content most recently written to that address.

We construct novel integrity algorithms that are based on the experimental observation that contents of blocks written to disk are usually efficiently distinguishable from random blocks. In two of our constructions, SRAND-BINT and RAND-BINT, data blocks are encrypted with a tweakable cipher. In this case, the integrity of the blocks that are efficiently distinguishable from random blocks can be checked by performing a randomness test on the block contents. The security properties of tweakable ciphers imply that if the block contents after decryption do not look random, then it is very likely that the contents are authentic. This idea permits a reduction in the amount of trusted storage needed for checking block integrity: a hash is stored in trusted storage only for those (usually few) blocks that are indistinguishable from random blocks (or, in short, *random-looking blocks*).

An example of a statistical test, called *the entropy test*, that can be used to distinguish block contents from random-looking blocks evaluates the entropy of a block and considers random those blocks that have an entropy higher than a threshold chosen experimentally. We provide a theoretical bound of the false negative rate for this statistical test. The bound is derived using Chernoff bounds and is useful in the security analysis of the algorithm.

A third algorithm, COMP-BINT, uses again the intuition that many workloads feature low-entropy files to reduce the space needed for integrity. In COMP-BINT the block content written to a disk block is compressed before encryption. If the compression level of the block content is high enough, then a message-authentication code of the block can be stored in the block itself, reducing the amount of storage necessary for integrity. In this construction, any length-preserving encryption scheme (and not necessarily a tweakable cipher) can be used to encrypt disk blocks.

The new integrity constructions are provably secure assuming the second preimage resistance of the hash function used to authenticate the random-looking blocks. We confirm through a month-long empirical evaluation in a Linux environment that the amount of trusted storage required by the new constructions is reduced compared to existing solutions.

Integrity in cryptographic file systems. We extend these ideas to construct two algorithms that protect the integrity of files in a cryptographic file system using only a constant amount of trusted storage per file (on the order of several hundred bytes). In the first algo-

rithm, called RAND-FINT, file blocks are encrypted with a tweakable cipher and integrity information is stored only for the random-looking blocks. To achieve a constant amount of trusted storage per file, a Merkle tree is built over the hashes of the random-looking blocks and the root of the Merkle tree is authenticated in trusted storage.

To protect against replay attacks, a possible solution is to make the tweak used in encrypting and decrypting a block dependent on a write counter for the block that denotes the total number of write operations performed to that block. The write counters for all blocks in a file should be authenticated using a small amount of trusted storage. We present a representation method for the block write counters that optimizes the amount of storage needed by the counters in the common case in which the majority of the files are written sequentially.

A second algorithm COMP-FINT proposed for file integrity in cryptographic file systems is an extension of the COMP-BINT construction for block storage systems. For the file blocks that can be compressed enough, their hash is stored inside the block. Like in the RAND-FINT construction, a Merkle tree is built over the hashes of the blocks in a file that cannot be compressed enough, and the root of the tree is authenticated in trusted storage.

Chapter 3 describes the two algorithms RAND-FINT and COMP-FINT that provide file integrity in cryptographic file systems (Oprea and Reiter [2006c]). We have also implemented our integrity constructions and the standard integrity algorithm based on Merkle trees in EncFS (Gough [2003]), an open-source user-level file system that transparently provides file block encryption on top of FUSE (FUSE). We provide an extensive evaluation of the three alogorithms with respect to the performance overhead, the bandwidth used for integrity and the amount of additional storage required for integrity, demonstrating how file contents, as well as file access patterns, have a great influence on the performance of the three schemes. Our experiments demonstrate that there is not a clear winner among the three integrity algorithms for *all* workloads, in that different integrity constructions are best suited to particular workloads.

1.2.2 Key Management Schemes for Lazy Revocation

In Chapter 4 we address the problem of efficient key management in cryptographic file systems using lazy revocation (Backes et al. [2005a, 2006, 2005b]). In systems adopting the lazy revocation model, the blocks in a set that initially share the same cryptographic keys might use different versions of the key after several user revocations. Storing and distributing these keys becomes more difficult in such systems than in systems using active revocation. An immediate solution to this problem, adopted by the first cryptographic file

systems using lazy revocation (Fu [1999]), is to store all keys for the blocks in a set at a trusted key server, but this results in a linear amount of trusted storage in the number of revocations. However, we are interested in more storage-efficient methods, in which the number of stored keys is not proportional to the number of revocations.

We formalize the notion of *key-updating schemes for lazy revocation* and give a rigorous security definition. The pseudorandom keys generated by our key-updating schemes can be used with a symmetric encryption algorithm to encrypt files for confidentiality or with a message-authentication code to authenticate files for integrity protection. In our model, a *center* (or *trusted entity*) initially generates some state information, which takes the role of the master secret key. The center state is updated at every revocation. We call the period of time between two revocations a *time interval*. Upon a user request, the center uses its current local state to derive a *user key* and gives that to the user. From the user key of some time interval, a user must be able to extract the key for any previous time interval efficiently. Security for key-updating schemes requires that any polynomial-time adversary with access to the user key for a particular time interval does not obtain any information about the keys for future time intervals.

We also describe two generic composition methods that combine two secure key updating schemes into a new scheme in which the number of time intervals is either the sum or the product of the number of time intervals of the initial schemes. Additionally, we investigate three constructions of key-updating schemes. The chaining scheme (CKU) uses a chain of pseudorandom generator applications and is related to existing methods using one-way hash chains. It has constant update cost for the center, but the complexity of the user-key derivation is linear in the total number of time intervals. The trapdoor permutation construction (TDKU) scheme can be based on arbitrary trapdoor permutations and generalizes the key rotation construction of the Plutus file system (Kallahalla et al. [2003]). It has constant update and user-key derivation times, but the update algorithm uses a relatively expensive public-key operation. These two constructions require that the total number of time intervals is polynomial in the security parameter.

Our third scheme, the binary tree scheme (TreeKU), uses a novel construction. It relies on a tree to derive the keys at the leaves from the master key at the root. The tree can be seen as resulting from the iterative application of the additive composition method and supports a practically unbounded number of time intervals. The binary-tree construction balances the tradeoff between the center-state update and user-key derivation algorithms (both of them have logarithmic complexity in the number of time intervals), at the expense of increasing the sizes of the user key and center state by a logarithmic factor in the number of time intervals.

We have implemented the three constructions and measured the maximum and average

computation time of the center at every revocation, as well as the maximum and average time of a client to extract a cryptographic key of a previous time interval. We demonstrate through our experiments that the binary-tree construction performs several orders of magnitude better than the existing constructions of key-updating schemes.

In Chapter 4, we also formalize three cryptographic primitives that can be used in a cryptographic file system with lazy revocation: symmetric encryption schemes, message-authentication codes and signature schemes (Backes et al. [2005b]). For each of these cryptographic primitives, we give a rigorous definition of security and describe generic constructions from key-updating schemes and known secure cryptographic primitives. Finally, we show how our primitives can be applied in cryptographic file systems adopting lazy revocation.

1.2.3 Consistency of Encrypted Files

In Chapter 5 we address the problem of consistency for encrypted file objects used to implement a cryptographic file system (Oprea and Reiter [2006a,b]). 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. The key and file objects may be implemented via completely different protocols and infrastructures. We are interested in the impact of the consistency of each on the consistency of the encrypted file object that they are used to implement. The consistency of the file object is 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 (Herlihy and Wing [1990]), 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 consider. 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' access permissions are revoked. 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.

An independent contribution of this work is the definition of two generic classes of consistency conditions. The class of *orderable consistency conditions* includes and generalizes well-known conditions such as linearizability (Herlihy and Wing [1990]), causal consistency (Ahamad et al. [1995]) and PRAM consistency (Lipton and Sandberg [1988]). The class of *forking consistency conditions* is particularly tailored to systems with untrusted shared storage and it extends fork consistency (Mazieres and Shasha [2002]) to other new, unexplored consistency conditions. These generic definitions of consistency might find other useful applications in the area of distributed shared memory consistency.

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 prove in our main result of this chapter (Theorem 15) 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. 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.

Finally, 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.

1.3 Organization

This thesis lies at the intersection of several distinct research areas: cryptography, storage systems and distributed systems. Each chapter describes a different contribution and can be read independently from the others (with the exception of Chapter 3 that uses some ideas from Chapter 2). We include the relevant background material at the beginning of each chapter and discuss the detailed comparison with related work at the end of each chapter. The beginning of each chapter contains a detailed outline of its contents.

Chapter 2 describes novel space-efficient integrity algorithms for block-level storage systems. In Chapter 3, we give the new integrity algorithms for cryptographic file systems that use only a constant amount of trusted storage per file. Chapter 3 uses some of the ideas developed in Chapter 2 to reduce the amount of storage for integrity, in particular the observation that typical block contents written to disks or files are not randomly distributed and the fact that the integrity of the file blocks efficiently distinguishable from random blocks can be easily checked with an efficient randomness test if file blocks are encrypted with a tweakable cipher. Chapter 4 formalizes a model of key management schemes for lazy revocation and some cryptographic primitives that can be used in a cryptographic file system adopting lazy revocation. Chapter 5 provides a framework for analyzing the consistency of an encrypted file object, starting from the consistency of the underlying file and key objects. Finally, Chapter 6 contains our conclusions and several thoughts for future work.

Chapter 2

Block-Level Integrity in Encrypted Storage Systems

In this chapter, we address the problem of designing efficient integrity algorithms for *encrypted* block-level storage systems. We present two new definitions of integrity for encrypted storage systems and give new constructions that are provably secure. The constructions reduce the additional space needed for integrity used by existing integrity methods by exploiting the high redundancy of the block contents usually written to disks. We confirm the space efficiency of the proposed integrity algorithms using some disk traces collected during a one-month period.

We start in Section 2.1 by describing some preliminary material needed for our definitions and proofs, in particular the definition of tweakable enciphering schemes and their security, everywhere second preimage resistance of hash functions, pseudorandom functions, message-authentication codes and Chernoff bounds. After describing our system model in Section 2.2, we define block write counters in Section 2.3 useful for preventing against replay attacks and for implementing a length-preserving encryption scheme (as shown in Section 2.4). We introduce the new integrity definitions in Section 2.5. The new integrity algorithms are given in Section 2.6 and an example of a statistical test used to distinguish block contents written to disk from uniformly random blocks is presented in Section 2.7. Our performance evaluation is in Section 2.8. We conclude this chapter by a discussion of related work in Section 2.9.

2.1 Preliminaries

2.1.1 Tweakable Enciphering Schemes

In this section, we review the definitions and security notions for tweakable enciphering schemes (Halevi and Rogaway [2003]). A tweakable enciphering scheme is a function of a tweak and has the property that for a fixed tweak, it is a strong, length-preserving pseudorandom permutation. More formally, a tweakable enciphering scheme is a function $\mathsf{E} : \mathcal{K}_\mathsf{E} \times \mathcal{T} \times \mathcal{M} \to \mathcal{M}$, where $\mathcal{K}_\mathsf{E}$ is the key space, $\mathcal{T}$ is the tweak set, $\mathcal{M}$ is the plaintext space (strings of length l bits), and for every $k \in \mathcal{K}_\mathsf{E}, T \in \mathcal{T}$, $\mathsf{E}(k, T, \cdot) = \mathsf{E}_k^T(\cdot)$ is a length preserving permutation. The inverse of the enciphering scheme E is the enciphering scheme $\mathsf{D} : \mathcal{K}_\mathsf{E} \times \mathcal{T} \times \mathcal{M} \to \mathcal{M}$, where $X = \mathsf{D}_k^T(Y)$ if and only if $\mathsf{E}_k^T(X) = Y$.

We define $\mathsf{Perm}(\mathcal{M})$ the set of all permutations $\pi : \mathcal{M} \to \mathcal{M}$ and $\mathsf{Perm}^\mathcal{T}(\mathcal{M})$ the set of all functions $\pi : \mathcal{T} \times \mathcal{M} \to \mathcal{M}$ such that $\pi(T) \in \mathsf{Perm}(\mathcal{M})$ for any $T \in \mathcal{T}$. For a function $\pi : \mathcal{T} \times \mathcal{M} \to \mathcal{M}$, we define $\pi^{-1} : \mathcal{T} \times \mathcal{M} \to \mathcal{M}$ such that $\pi^{-1}(T, y) = x$ if and only if $\pi(T, x) = y$.

Definition 1 (PRP security of tweakable enciphering schemes) *Let* $\mathsf{E} : \mathcal{K}_\mathsf{E} \times \mathcal{T} \times \mathcal{M} \to \mathcal{M}$ *be a tweakable enciphering scheme and* $\mathcal{A}$ *an adversary algorithm.* $\mathcal{A}$ *has access to oracles* $\mathsf{E}_k(\cdot,\cdot)$ *and* $\mathsf{D}_k(\cdot,\cdot)$ *that take as input a tweak and a plaintext, and a tweak and a ciphertext, respectively. The PRP-advantage of adversary* $\mathcal{A}$ *is defined as:*

$$\mathsf{Adv}^{\mathsf{prp}}_{\mathsf{E},\mathcal{A}} = |\Pr[k \xleftarrow{R} \mathcal{K}_\mathsf{E} : \mathcal{A}^{\mathsf{E}_k,\mathsf{D}_k} = 1] - \Pr[\pi \xleftarrow{R} \mathsf{Perm}^\mathcal{T}(\mathcal{M}) : \mathcal{A}^{\pi,\pi^{-1}} = 1]|.$$

We denote by $\mathsf{Adv}^{\mathsf{prp}}_{\mathsf{E}}(\tau, q_1, q_2)$ *the maximum advantage* $\mathsf{Adv}^{\mathsf{prp}}_{\mathsf{E},\mathcal{A}}$ *over all adversaries* $\mathcal{A}$ *that run in time at most* τ*, make* q_1 *queries to* E_k *and* q_2 *queries to* D_k*.*

The definition of PRP-security is a natural extension of the strong pseudorandom permutation definition by Naor and Reignold (Naor and Reingold [1997]).

2.1.2 Everywhere Second Preimage Resistant Hash Functions

Let $\mathsf{HMAC} : \mathcal{K}_\mathsf{HMAC} \times \mathcal{M} \to \{0,1\}^s$ be a family of hash functions. Intuitively, everywhere second preimage resistance requires that for any message $m \in \mathcal{M}$, it is hard to find a collision for a function chosen at random from the family, i.e., $m' \neq m$ such that $\mathsf{HMAC}_k(m') = \mathsf{HMAC}_k(m)$, with $k \xleftarrow{R} \mathcal{K}_\mathsf{HMAC}$. This definition has been formalized by Rogaway and Shrimpton (Rogaway and Shrimpton [2004]) and it is stronger than the standard definition of second preimage resistance, but weaker than collision resistance.

Definition 2 (Everywhere second preimage resistance of a hash function) *For an adversary algorithm $\mathcal{A}$, we define the advantage of $\mathcal{A}$ for the everywhere second preimage resistance of hash function family* HMAC *as:*

$$\begin{aligned} \mathsf{Adv}^{\mathsf{spr}}_{\mathsf{HMAC},\mathcal{A}} &= \max_{m \in \mathcal{M}} \{\Pr[k \xleftarrow{R} \mathcal{K}_{\mathsf{HMAC}}, m' \leftarrow \mathcal{A}(k, m) : (m \neq m') \\ &\wedge \ \ \mathsf{HMAC}_k(m') = \mathsf{HMAC}_k(m)]\}. \end{aligned}$$

$\mathsf{Adv}^{\mathsf{spr}}_{\mathsf{HMAC}}(\tau)$ *denotes the maximum advantage* $\mathsf{Adv}^{\mathsf{spr}}_{\mathsf{HMAC},\mathcal{A}}$ *for all adversaries $\mathcal{A}$ running in time at most τ.*

An unkeyed hash function $h : \mathcal{M} \rightarrow \{0,1\}^s$ can be viewed as a particular case of a hash function family with the key space of size 1. The definition of everywhere second preimage resistance for an unkeyed hash function h follows immediately from Definition 2.

2.1.3 Pseudorandom Functions

A pseudorandom function is a family of functions $\mathsf{PRF} : \mathcal{K}_{\mathsf{PRF}} \times D \rightarrow R$ with the property that the behavior of a random instance from the family is computationally indistinguishable from that of a random function with the same domain and codomain. A formal definition is below.

$\mathsf{Exp}^{\mathtt{prf}\text{-}0}_{\mathsf{PRF},\mathcal{A}}$	$\mathsf{Exp}^{\mathtt{prf}\text{-}1}_{\mathsf{PRF},\mathcal{A}}$
$k \xleftarrow{R} \mathcal{K}_{\mathsf{PRF}}$	$g \xleftarrow{R} \mathsf{Rand}^{D \rightarrow R}$
$d \leftarrow A^{\mathsf{PRF}_k}$	$d \leftarrow A^g$
return d	return d

Figure 2.1: Experiments defining the security of pseudorandom function family PRF.

Definition 3 (Pseudorandom functions) *Let* PRF *be a family of functions as above and $\mathcal{A}$ an adversary algorithm that has access to an oracle $g : D \rightarrow R$ and participates in the experiments from Figure 2.1. Let* $\mathsf{Rand}^{D \rightarrow R}$ *be the set of all functions from D to R. We define the* `prf`*-advantage of $\mathcal{A}$ in attacking the security of the family* PRF *as:*

$$\mathsf{Adv}^{\mathtt{prf}}_{\mathsf{PRF},\mathcal{A}} = |\Pr[\mathsf{Exp}^{\mathtt{prf}\text{-}1}_{\mathsf{PRF},\mathcal{A}} = 1] - \Pr[\mathsf{Exp}^{\mathtt{prf}\text{-}0}_{\mathsf{PRF},\mathcal{A}} = 1]|.$$

We denote by $\mathsf{Adv}^{\mathtt{prf}}_{\mathsf{PRF}}(\tau, q)$ *the maximum advantage* $\mathsf{Adv}^{\mathtt{prf}}_{\mathsf{PRF},\mathcal{A}}$ *of all adversary algorithms $\mathcal{A}$ running in time at most τ and making q queries to its oracle.*

2.1.4 Security of Message-Authentication Codes

A message-authentication code (MAC) consists of three algorithms: a key generation algorithm $\mathsf{Gen}(1^\kappa)$ that outputs a key taking as input a security parameter, a tagging algorithm $\mathsf{Tag}_k(m)$ that outputs the authentication tag v of a given message m with key k, and a verification algorithm $\mathsf{Ver}_k(m, v)$ that outputs a bit. A tag v is said to be *valid* on a message m for a key k if $\mathsf{Ver}_k(m, v) = 1$. The first two algorithms might be probabilistic, but Ver is deterministic. The correctness property requires that $\mathsf{Ver}_k(m, \mathsf{Tag}_k(m)) = 1$, for all keys k generated with the Gen algorithm and all messages m from the message space.

CMA-security for a message-authentication code (Bellare et al. [1994]) requires that any polynomial-time adversary with access to a tagging oracle $\mathsf{Tag}_k(\cdot)$ and a verification oracle $\mathsf{Ver}_k(\cdot, \cdot)$ is not able to generate a message and a valid tag for which it did not query the tagging oracle.

Definition 4 (CMA-Security of Message-Authentication Codes) *Let* MA *be a message-authentication code and $\mathcal{A}$ an adversary algorithm. We define the* cma-*advantage of $\mathcal{A}$ for* MA *as:*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{cma}}_{\mathsf{MA},\mathcal{A}} &= \Pr[k \leftarrow \mathsf{Gen}(1^\kappa), (m, v) \leftarrow \mathcal{A}^{\mathsf{Tag}_k(\cdot),\mathsf{Ver}_k(\cdot,\cdot)}() : \mathsf{Ver}_k(m, v) = 1 \text{ AND} \\ &\quad\ m \textit{ was not a query to } \mathsf{Tag}_k(\cdot)].\end{aligned}$$

We define $\mathsf{Adv}^{\mathsf{cma}}_{\mathsf{MA}}(\tau, q_1, q_2)$ to be the maximum advantage $\mathsf{Adv}^{\mathsf{cma}}_{\mathsf{MA},\mathcal{A}}$ over all adversary algorithms $\mathcal{A}$ running in time at most τ and making q_1 queries to the tagging oracle and q_2 queries to the verification oracle.

The tagging algorithm of a message-authentication code can also be viewed as a family of keyed hash functions $\mathsf{HMAC} : \mathcal{K}_{\mathsf{HMAC}} \times \{0,1\}^* \rightarrow \{0,1\}^s$ that given a key from the key space $\mathcal{K}_{\mathsf{HMAC}}$ and a message, outputs a tag of length s bits. Sometimes we refer to a message-authentication code using only its tagging algorithm and omitting explicit reference to the key generation and verification algorithms.

2.1.5 Chernoff Bounds

Let $X_1, \ldots, X_n$ be independent Poisson trials with $\Pr[X_i = 1] = p_i$, $\Pr[X_i = 0] = 1 - p_i$, for $0 < p_i < 1$. Denote $X = \sum_{i=1}^n X_i$ and $\mu = \mathbf{E}(X) = \sum_{i=1}^n p_i$. Then the following bounds hold:

1. For any $\epsilon > 0$, $\Pr[X > (1+\epsilon)\mu] < \left(\frac{e^\epsilon}{(1+\epsilon)^{1+\epsilon}}\right)^\mu$.

2. For any $0 < \epsilon \leq 1$, $\Pr[X < (1-\epsilon)\mu] < e^{-\frac{\mu\epsilon^2}{2}}$.

These bounds are taken from Motwani and Raghavan [1995].

2.2 System Model

We consider a limited storage client that keeps its data on an untrusted storage device, denoted by "storage server". The data is partitioned into fixed-length sectors or blocks. In our model, the server can behave maliciously, by mounting attacks against the confidentiality and integrity of the client's data. We assume that the server is available, i.e., it responds to client's read and write queries. However, no guarantees are given about the correctness of its replies.

For data confidentiality, we assume that blocks are encrypted by clients using a *length-preserving encryption scheme* that can be implemented by either a tweakable enciphering scheme or an encryption scheme constructed from a block cipher in CBC mode, as described in Section 2.4. The client is responsible for protecting its data integrity from malicious server behavior by using a small amount of additional *trusted storage* denoted TS. As trusted storage is expensive, our goal is to design schemes that minimize the amount of trusted storage for integrity and provide provable integrity.

The storage interface provides two basic operations to the clients: $\mathsf{WriteBlock}(bid, C)$ stores content C at block address bid and $C \leftarrow \mathsf{ReadBlock}(bid)$ reads (encrypted) content from block address bid. In addition, in an integrity scheme for encrypted storage defined below, a client can compute or check the integrity of a data block.

Definition 5 (Integrity schemes for encrypted block-level storage) *An integrity scheme for encrypted block-level storage is a tuple of algorithms* $\mathsf{IntS} = (\mathsf{Init},\mathsf{Write},\mathsf{Check})$ *where:*

1. *The initialization algorithm* Init *runs the key generation algorithm of the length-preserving stateful encryption scheme and outputs a secret key k for the client that can be used in the encryption and decryption algorithms. It also initializes the client trusted state* $\mathsf{TS} \leftarrow \emptyset$*;*

2. *The write algorithm* $\mathsf{Write}(k, bid, B)$ *takes as input the secret key generated by the* Init *algorithm, block content B and block identifier bid. The client encrypts the block B and updates the integrity information stored in* TS*. The client also invokes the* $\mathsf{WriteBlock}$ *operation to send the encrypted block content to the server. The* Write *algorithm returns the encrypted block content sent to the storage server.*

3. *When performing a* $\mathsf{Check}(k, bid, C)$ *operation, the client checks the integrity of the block content* C *read from block* bid *in a* $\mathsf{ReadBlock}$ *operation using client state* TS. *It outputs either the decrypted block if the client believes that it is authentic, or* $\perp$, *otherwise.*

2.3 Block Write Counters

The integrity constructions for encrypted storage described in Section 2.6 use write counters for disk blocks to keep track of the number of writes done to a particular block. Besides their use to prevent against replay attacks, block write counters can be used to implement length-preserving stateful encryption schemes, as shown in the next section.

We define several operations for the write counters of disk blocks:

- The $\mathsf{UpdateCtr}(bid)$ algorithm either initializes the value of the counter for block bid to 1, or it increments the counter for block bid if it has already been initialized.

- Function $\mathsf{GetCtr}(bid)$ returns the value of the counter for block bid.

To store the counters for disk blocks, we keep a flag (one bit for each block) that is initially set to 0 for all blocks and becomes 1 when the block is written first. We do not need to store counters for blocks that are written once or not written at all, as the counter could be inferred in these cases from the flags. An implementation of the UpdateCtr and GetCtr algorithms is given in Figure 2.2. We denote by L_C the associative array of (block identifiers, counter) pairs and by $F(bid)$ the flag of block identifier bid.

```
UpdateCtr(bid):                    GetCtr(bid):
 if F(bid) = 1                      if (bid, w) ∈ L_C
   if (bid, w) ∈ L_C                  return w
     remove (bid, w) from L_C       else
     w ← w + 1                        if F(bid) = 1
   else                                 return 1
     w ← 2                            else
   insert (bid, w) into L_C             return 0
 else
   F(bid) ← 1
```

Figure 2.2: The UpdateCtr and GetCtr algorithms.

2.4 Length-Preserving Stateful Encryption

Secure encryption schemes are usually not length-preserving. However, one of our design goals is to add security (and, in particular, encryption) to storage systems in a manner transparent to the storage servers. For this purpose, we introduce here the notion of *length-preserving stateful encryption scheme*, an encryption scheme that constructs encrypted blocks that preserve the length of original blocks (storing additional information in either trusted or untrusted storage). We define a length-preserving stateful encryption scheme to consist of a key generation algorithm $\mathsf{Gen}^{\mathsf{len}}$ that generates an encryption key at random from the key space, an encryption algorithm $\mathsf{E}^{\mathsf{len}}$ that encrypts block content B for block identifier bid with key k and outputs ciphertext C, and a decryption algorithm $\mathsf{D}^{\mathsf{len}}$ that decrypts the encrypted content C of block bid with key k and outputs the plaintext B. We present two different constructions of length-preserving stateful encryption schemes, one based on tweakable ciphers and one derived from any block cipher in CBC mode using write counters for disk blocks.

Construction from tweakable ciphers. Let $\mathsf{E} : \mathcal{K}_{\mathsf{E}} \times \mathcal{T} \times \mathcal{M} \rightarrow \mathcal{M}$ be a tweakable enciphering scheme and D its inverse. A length-preserving stateful encryption scheme can be immediately implemented as follows:

- The $\mathsf{Gen}^{\mathsf{len}}$ algorithm selects a key k at random from the key space $\mathcal{K}_{\mathsf{E}}$;
- Both the $\mathsf{E}^{\mathsf{len}}$ and $\mathsf{D}^{\mathsf{len}}$ algorithms use the block identifier (or any function of the block identifier) as a tweak in E, and D, respectively (e.g., $\mathsf{E}^{\mathsf{len}}_k(bid, B) = \mathsf{E}^{bid}_k(B)$ and $\mathsf{D}^{\mathsf{len}}_k(bid, B) = \mathsf{D}^{bid}_k(B)$).

Construction using block write counters. Write counters for disk blocks can be used to construct a length-preserving stateful encryption scheme. Let $(\mathsf{Gen}, \mathsf{E}, \mathsf{D})$ be an encryption scheme constructed from a block cipher in CBC mode. To encrypt a n-block message in the CBC encryption mode, a random initialization vector is chosen. The ciphertext consists of $n + 1$ blocks, with the first being the initialization vector. We denote by $\mathsf{E}_k(B, iv)$ the output of the encryption of B (excluding the initialization vector) using key k and initialization vector iv, and similarly by $\mathsf{D}_k(C, iv)$ the decryption of C using key k and initialization vector iv.

We replace the random initialization vectors for encrypting a block in the file in CBC mode with a pseudorandom function application of the block index concatenated with the write counter for the block. This is intuitively secure because different initialization

vectors are used for different encryptions of the same block, and moreover, the properties of pseudorandom functions imply that the initialization vectors are indistinguishable from random. It is thus enough to store the write counters for the disk blocks, and the initialization vectors can be easily inferred.

The $\mathsf{Gen}^{\mathsf{len}}$, $\mathsf{E}^{\mathsf{len}}$ and $\mathsf{D}^{\mathsf{len}}$ algorithms for a length-preserving stateful encryption scheme are described in Figure 2.3. Here $\mathsf{PRF} : \mathcal{K}_{\mathsf{PRF}} \times \mathcal{C} \rightarrow \mathcal{K}_{\mathcal{B}}$ denotes a pseudorandom function family with key space $\mathcal{K}_{\mathsf{PRF}}$, message space $\mathcal{C}$ (i.e., the set of all block indices concatenated with block counter values), and output space $\mathcal{K}_{\mathcal{B}}$ (i.e., the space of encryption blocks for E).

$\mathsf{Gen}^{\mathsf{len}}()$:	$\mathsf{E}^{\mathsf{len}}_{<k_1,k_2>}(bid, B)$:	$\mathsf{D}^{\mathsf{len}}_{<k_1,k_2>}(bid, C)$:
$k_1 \stackrel{R}{\leftarrow} \mathcal{K}_{\mathsf{PRF}}$	$\mathsf{UpdateCtr}(bid)$	$iv \leftarrow \mathsf{PRF}_{k_1}(bid \vert\vert \mathsf{GetCtr}(bid))$
$k_2 \leftarrow \mathsf{Gen}()$	$iv \leftarrow \mathsf{PRF}_{k_1}(bid \vert\vert \mathsf{GetCtr}(bid))$	$B \leftarrow \mathsf{D}_{k_2}(C, iv)$
return $<k_1, k_2>$	$C \leftarrow \mathsf{E}_{k_2}(B, iv)$	return B
	return C	

Figure 2.3: Implementing a length-preserving stateful encryption scheme with write counters.

2.5 Notions of Integrity for Encrypted Block-Level Storage

In formally defining integrity for encrypted storage, we consider adversary algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ with access to two oracles: a write oracle $\mathsf{Write}(k, \cdot, \cdot)$ and a check oracle $\mathsf{Check}(k, \cdot, \cdot)$. Both oracles are stateful in the sense that they maintain an internal state across invocations. The write oracle takes as input a block identifier and a block content. It computes the encryption of the block content given as input and its integrity information, invokes a WriteBlock query to the storage interface and returns the encrypted block content written to disk. The check oracle takes as input a block identifier and a ciphertext. It decrypts the encrypted block, checks its integrity information using its internal state and outputs either the decrypted block if its integrity is verified or $\perp$, otherwise. Adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$ play the storage server's role in our model.

In the first notion of integrity we define, an adversary wins if, intuitively, it outputs a block identifier and a ciphertext whose decryption is considered valid by the Check algorithm, but which has not been generated through a call to the Write oracle for that particular block identifier. Our second notion of integrity is stronger than the first one and incorpo-

rates defense against replay attacks. An adversary succeeds if it outputs a block identifier and a ciphertext whose decryption is valid, and which has not been generated in the *latest* Write query to that block identifier. These two notions of integrity are straightforward generalizations of the notions of integrity for symmetric encryption schemes defined by Bellare and Namprempre (Bellare and Namprempre [2000]) (i.e., *integrity of plaintexts* and *integrity of ciphertexts*) and we define them formally below.

$\mathsf{Exp}^{\mathsf{int\text{-}st}}_{\mathsf{IntS},\mathcal{A}_1}$:	$\mathsf{Exp}^{\mathsf{int\text{-}st\text{-}rep}}_{\mathsf{IntS},\mathcal{A}_2}$:
$k \leftarrow \mathsf{Init}()$ $\mathcal{A}_1$ adaptively queries $\mathsf{Write}(k,\cdot,\cdot)$ and $\mathsf{Check}(k,\cdot,\cdot)$. If $\mathcal{A}_1$ outputs (bid, C) such that: 1. $\mathsf{Check}(k, bid, C)$ returns $B \neq \perp$; 2. $\mathcal{A}_1$ did not receive C from a query to $\mathsf{Write}(k, bid, \cdot)$, then return 1, else return 0.	$k \leftarrow \mathsf{Init}()$ $\mathcal{A}_2$ adaptively queries $\mathsf{Write}(k,\cdot,\cdot)$ and $\mathsf{Check}(k,\cdot,\cdot)$. If $\mathcal{A}_2$ outputs (bid, C) such that: 1. $\mathsf{Check}(k, bid, C)$ returns $B \neq \perp$; 2. $\mathcal{A}_2$ did not receive C from the **latest** query to $\mathsf{Write}(k, bid, \cdot)$, then return 1, else return 0.

Figure 2.4: Experiments for defining storage integrity.

Definition 6 (Security of integrity schemes for encrypted block-based storage) *Let* $\mathsf{IntS} = (\mathsf{Init},\mathsf{Write},\mathsf{Check})$ *be an integrity schemes for encrypted block-based storage and* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *two adversary algorithms that participate in the experiments defined in Figure 2.4.*

We define the advantages of the adversaries for the integrity of the scheme IntS *as:*

$$\mathsf{Adv}^{\mathsf{int\text{-}st}}_{\mathsf{IntS},\mathcal{A}_1} = \Pr[\mathsf{Exp}^{\mathsf{int\text{-}st}}_{\mathsf{IntS},\mathcal{A}_1} = 1];$$

$$\mathsf{Adv}^{\mathsf{int\text{-}st\text{-}rep}}_{\mathsf{IntS},\mathcal{A}_2} = \Pr[\mathsf{Exp}^{\mathsf{int\text{-}st\text{-}rep}}_{\mathsf{IntS},\mathcal{A}_2} = 1].$$

We denote by $\mathsf{Adv}^{\mathsf{int\text{-}st}}_{\mathsf{IntS}}(\tau, q_1, q_2)$ *and* $\mathsf{Adv}^{\mathsf{int\text{-}st\text{-}rep}}_{\mathsf{IntS}}(\tau, q_1, q_2)$ *the maximum advantages* $\mathsf{Adv}^{\mathsf{int\text{-}st}}_{\mathsf{IntS},\mathcal{A}}$ *and* $\mathsf{Adv}^{\mathsf{int\text{-}st\text{-}rep}}_{\mathsf{IntS},\mathcal{A}}$*, respectively, over all adversaries* $\mathcal{A}$ *that run in time at most* τ *and make* q_1 *queries to* $\mathsf{Write}(k,\cdot,\cdot)$ *and* q_2 *queries to* $\mathsf{Check}(k,\cdot,\cdot)$.

The two notions of integrity require different correctness properties:

1. int-st: If the client performs $\mathsf{Write}(k, bid, B)$, then block B is accepted as valid, i.e., if $C \leftarrow \mathsf{ReadBlock}(bid)$, then $\mathsf{Check}(k, bid, C)$ returns B.

2. int-st-rep: If $\mathsf{Write}(k, bid, B)$ is the *most recent* write operation to block bid, then block content B is accepted as valid, i.e., if $C \leftarrow \mathsf{ReadBlock}(bid)$, then $\mathsf{Check}(k, bid, C)$ returns B.

Remark. The attack model considered here is stronger than the one in the real storage scenario. In our definitions, the adversary has adaptive access to the Write and Check oracles, which is infeasible for the storage server. The storage server only receives encrypted blocks from the client, for which it does not know the actual plaintexts.

2.6 Constructions of Integrity Schemes for Encrypted Block-Level Storage

We describe four constructions of integrity schemes for encrypted block-level storage and analyze their security and efficiency tradeoffs. We first describe a very simple int-st-rep secure construction HASH-BINT, which is used in many storage systems. We include this basic scheme here to compare its client-side storage to the more sophisticated schemes that we propose. Second, we propose three new constructions SRAND-BINT, RAND-BINT, and COMP-BINT that use the redundancy in typical block contents to reduce the storage space needed for integrity. The integrity of SRAND-BINT and RAND-BINT relies on the properties of tweakable ciphers that are used for block encryption. COMP-BINT has the advantage that any length-preserving stateful encryption scheme as defined in Section 2.4 can be used for block encryption.

2.6.1 Hash scheme HASH-BINT

The hash scheme is very simple: for each block written at a particular address, the client computes and stores the block identifier and a hash of the block in trusted storage. For a given address, the stored hash corresponds to the last written block, thus preventing the adversary in succeeding with a replay attack. The amount of additional storage kept by the client is linear in the number of blocks written to the server, i.e., 20 bytes per block if a cryptographically secure hash function such as SHA-1 is used plus the block identifiers (e.g., 2 or 4 bytes, depending on the implementation).

In order to fully specify the scheme, we need a length-preserving encryption scheme $(\mathsf{Gen}^{\mathsf{len}}, \mathsf{E}^{\mathsf{len}}, \mathsf{D}^{\mathsf{len}})$ and an everywhere second preimage resistant unkeyed hash function defined on the plaintext space $\mathcal{M}$ of $\mathsf{E}^{\mathsf{len}}$, $h : \mathcal{M} \rightarrow \{0,1\}^s$. The client keeps in the local trusted storage TS a list of pairs (block identifier, block hash), that is initialized to the empty set. The scheme HASH-BINT is detailed in Figure 2.5. Below we prove the security of the hash scheme HASH-BINT based on the everywhere second preimage resistance property of the hash function used.

$\mathsf{Init}()$:	$\mathsf{Write}(k, bid, B)$:	$\mathsf{Check}(k, bid, C)$:
$k \xleftarrow{R} \mathsf{Gen}^{\mathsf{len}}()$ $\mathsf{TS} \leftarrow \emptyset$	remove $(bid, *)$ from TS insert $(bid, h(B))$ into TS $C \leftarrow \mathsf{E}_k^{\mathsf{len}}(bid, B)$ $\mathsf{WriteBlock}(bid, C)$	$B \leftarrow \mathsf{D}_k^{\mathsf{len}}(bid, C)$ if $(bid, h(B)) \in \mathsf{TS}$ return B else return $\perp$

Figure 2.5: Scheme HASH-BINT.

Proposition 1 *If h is an everywhere second preimage resistant hash function, then the integrity scheme* HASH-BINT *is* int-st-rep *secure:*

$$\mathsf{Adv}^{\mathsf{int\text{-}st\text{-}rep}}_{\mathsf{HASH\text{-}BINT}}(\tau, q_1, q_2) \leq \mathsf{Adv}^{\mathsf{spr}}_{h}(\tau).$$

Proof: Assume there is an adversary $\mathcal{A}$ for HASH-BINT with advantage $\mathsf{Adv}^{\mathsf{int\text{-}st\text{-}rep}}_{\mathsf{HASH\text{-}BINT},\mathcal{A}}$ that runs in time τ. From Definition 6, it follows that $\mathcal{A}$ outputs a block identifier bid and an encrypted block content C such that: (1) $\mathsf{Check}(k, bid, C)$ returns $B \neq \perp$, i.e., $(bid, h(B)) \in \mathsf{TS}$, with $B = \mathsf{D}_k^{\mathsf{len}}(bid, C)$; and (2) C was not received by $\mathcal{A}$ in the last query to $\mathsf{Write}(k, bid,$
$\cdot)$. There are two possibilities:

- C was never written by $\mathcal{A}$ at block identifier bid. Since $(bid, h(B)) \in \mathsf{TS}$, there exists a different query $\mathsf{Write}(k, bid, B')$ with $B' \neq B$ and $h(B') = h(B)$. But then an adversary $\mathcal{B}$ with the same advantage and running time as $\mathcal{A}$ can be constructed for the everywhere second preimage resistance of h.
- C was written at block identifier bid, but it was overwritten by a subsequent query $\mathsf{Write}(k, bid, B')$. Assume that B' is the last block written in a $\mathsf{Write}(k, bid, \cdot)$ query. Then $h(B) = h(B')$ and in this case again an adversary $\mathcal{B}$ with the same advantage and running time for the everywhere second preimage resistance of h can be constructed.

The claim of the proposition follows immediately. □

2.6.2 Schemes Based on a Randomness Test

We design new, storage-efficient constructions to obtain int-st and int-st-rep integrity, respectively. Our constructions are based on two observations. The first one is that blocks

written to disk do not look random in practice; in fact they tend to have very low entropy. And, second, if an adversary tries to modify ciphertexts encrypted with a tweakable enciphering scheme, the resulting plaintext is indistinguishable from random with very high probability. The second property derives immediately from the PRP security of a tweakable enciphering scheme defined in Section 2.1.1.

We start by giving a simple and very space-efficient construction that is only int-st secure and then extend it to an int-st-rep secure algorithm.

A Simple Scheme SRAND-BINT

In construction SRAND-BINT, we need a statistical test IsRand that can distinguish uniformly random blocks from non-random ones. More explicitly, $\texttt{IsRand}(M), M \in \mathcal{M}$ returns 1 with high probability if M is a uniformly random block in $\mathcal{M}$, and 0 otherwise. Of course, the statistical test is not perfect. It is characterized by the false negative rate α, defined as the probability that a uniformly random element is considered not random by the test, i.e:

$$\alpha = \Pr[M \xleftarrow{R} \mathcal{M} : \texttt{IsRand}(M) = 0].$$

In designing such a statistical test, the goal is to have a very small false negative rate. We will discuss more in Section 2.7 about a particular instantiation for IsRand.

The idea of our new construction is very intuitive: before encrypting a block M with tweakable enciphering scheme E, the client computes $\texttt{IsRand}(M)$. If this returns 1, then the client keeps a hash of that block for authenticity in a list L_R in the trusted storage space. Otherwise, the client stores nothing, as the fact that the block is efficiently distinguishable from random blocks will be used to verify its integrity upon return. The block is then encrypted with a tweak equal to the block identifier and sent over to the server. When reading a ciphertext from an address, the client first decrypts it to obtain a plaintext M and then computes $\texttt{IsRand}(M)$. If $\texttt{IsRand}(M) = 1$ and its hash is not stored in the hash list, then the client knows that the server has tampered with the ciphertext. Otherwise, the block is authentic.

We denote by L_R the associative array of (block identifier, block hash) pairs for random-looking blocks. The new construction SRAND-BINT is detailed in Figure 2.6. We give the following theorem that guarantees the int-st integrity of SRAND-BINT.

Theorem 2 *If* E *is a PRP-secure tweakable enciphering scheme with the plaintext size l bits,* D *is its inverse, h is an everywhere second preimage resistant hash function and α*

Init():	Write(k, bid, B):	Check(k, bid, C):

$k \stackrel{R}{\leftarrow} \mathcal{K}_{\mathsf{E}}$
$L_R \leftarrow \emptyset$
	remove $(bid, *)$ from L_R
	if $\texttt{IsRand}(B) = 1$
	  insert $(bid, h(B))$ into L_R
	$T \leftarrow bid$
	WriteBlock$(bid, C \leftarrow \mathsf{E}_k^T(B))$
		$T \leftarrow bid$
		$B \leftarrow \mathsf{D}_k^T(C)$
		if $\texttt{IsRand}(B) = 0$
		  return B
		else
		  if $(bid, h(B)) \in L_R$
		    return B
		  else
		    return $\perp$

Figure 2.6: Scheme SRAND-BINT.

(the false negative rate of IsRand*) is small, then* SRAND-BINT *is* int-st *secure:*

$$\mathsf{Adv}^{\mathsf{int\text{-}st}}_{\mathsf{SRAND\text{-}BINT}}(\tau, q_1, q_2) \leq \mathsf{Adv}^{\mathsf{prp}}_{\mathsf{E}}(\tau, q_1, q_2) + \mathsf{Adv}^{\mathsf{spr}}_{h}(\tau) + \frac{(q_2+1)\alpha 2^l}{2^l - q_1}.$$

Proof: Assume there exists an attacker $\mathcal{A}$ for the int-st integrity of RAND-BINT that runs in time τ and makes q_1 queries to the Write oracle and q_2 queries to the Check oracle. Since $T = bid$ in the description of the construction from Figure 2.6, we use T instead of bid in the Write and Check queries. We construct a distinguisher $\mathcal{D}$ for the PRP-security of E. $\mathcal{D}$ has access to oracles G and G^{-1}, which are either $\mathsf{E}_k, \mathsf{D}_k$ with $k \stackrel{R}{\leftarrow} \mathcal{K}_{\mathsf{E}}$ or π, π^{-1} with $\pi \stackrel{R}{\leftarrow} \mathsf{Perm}^{\mathcal{T}}(\mathcal{M})$.

$\mathcal{D}$ simulates the oracles $\mathsf{Write}(k, \cdot, \cdot)$ and $\mathsf{Check}(k, \cdot, \cdot)$ for $\mathcal{A}$ and keeps a list L_R of block identifiers and hashes for the randomly looking block contents that $\mathcal{A}$ queries to the Write oracle. $\mathcal{D}$ replies to a $\mathsf{Write}(k, T, B)$ query with $C = G(T, B)$ and stores $(T, h(B))$ in L_R if $\texttt{IsRand}(B) = 1$. $\mathcal{D}$ replies to a $\mathsf{Check}(k, T, C)$ query with $B = G^{-1}(T, C)$ if $\texttt{IsRand}(B) = 0$ or $(T, h(B)) \in L_R$, and $\perp$, otherwise. $\mathcal{D}$ makes the same number of queries to its G and G^{-1} oracles as $\mathcal{A}$ makes to the Write and Check oracles, respectively. The running time of $\mathcal{D}$ is τ.

If $\mathcal{A}$ succeeds, i.e., outputs a block identifier T and an encrypted block content C such that $\mathsf{Check}(k, T, C)$ returns $B \neq \perp$ and C was not output to $\mathcal{A}$ in a $\mathsf{Write}(k, T, \cdot)$ query, then $\mathcal{D}$ outputs 1. Otherwise, $\mathcal{D}$ outputs 0. We express the advantage of distinguisher $\mathcal{D}$ as a function of the advantage of adversary $\mathcal{A}$.

From Definition 1, the PRP-advantage of adversary $\mathcal{D}$ is:

$$\mathsf{Adv}^{\mathsf{prp}}_{\mathsf{E},\mathcal{D}} = \Pr[k \stackrel{R}{\leftarrow} \mathcal{K}_{\mathsf{E}}, \mathcal{D}^{\mathsf{E}_k, \mathsf{D}_k} = 1] - \Pr[\pi \stackrel{R}{\leftarrow} \mathsf{Perm}^{\mathcal{T}}(\mathcal{M}), \mathcal{D}^{\pi, \pi^{-1}} = 1].$$

It is immediate that the probability of $\mathcal{D}$ outputting 1 in the case when the oracles G and G^{-1} are $\mathsf{E}_k, \mathsf{D}_k$, respectively, is equal to the probability of success of $\mathcal{A}$:

$$\Pr[k \stackrel{R}{\leftarrow} \mathcal{K}_{\mathsf{E}}, \mathcal{D}^{\mathsf{E}_k,\mathsf{D}_k} = 1] = \Pr[\mathcal{A} \text{ succeeds}] = \mathsf{Adv}^{\mathsf{int\text{-}st}}_{\mathsf{SRAND\text{-}BINT},\mathcal{A}}.$$

In the case when the oracles G and G^{-1} are π and π^{-1}, with π drawn randomly from $\mathsf{Perm}^{\mathcal{T}}(\mathcal{M})$, we can express the probability of $\mathcal{D}$ outputting 1 as a function of the false negative rate of `IsRand` and the everywhere second preimage resistance of h.

$$\begin{aligned}
& \Pr[\pi \stackrel{R}{\leftarrow} \mathsf{Perm}^{\mathcal{T}}(\mathcal{M}), \mathcal{D}^{\pi,\pi^{-1}} = 1] \\
= \; & \Pr[\mathcal{A} \text{ succeeds } | \mathcal{A} \text{ receives random } C_1, \dots, C_{q_1} \text{ from } \mathsf{Write}(k, \cdot, \cdot) \\
& \qquad \text{and random } B_1, \dots, B_{q_2} \text{ from } \mathsf{Check}(k, \cdot, \cdot)].
\end{aligned}$$

Making the notation $\mathcal{A} \sim (T, B, C)$ for $\mathcal{A}$ outputting block identifier T and encrypted block content C such that $B = G^{-1}(T, C)$, the last probability can be written as:

$$\begin{aligned}
& \Pr[\pi \stackrel{R}{\leftarrow} \mathsf{Perm}^{\mathcal{T}}(\mathcal{M}), \mathcal{D}^{\pi,\pi^{-1}} = 1] \\
= \; & \Pr[\mathcal{A} \sim (T, B, C) : C \text{ was not received in a } \mathsf{Write}(k, T, \cdot) \text{ query by } \mathcal{A} \text{ AND} \\
& \qquad \mathsf{Check}(k, T, C) = B] \\
= \; & \Pr[\mathcal{A} \sim (T, B, C) : (T, B) \text{ was not a query to } \mathsf{Write}(k, \cdot, \cdot) \text{ AND} \\
& \qquad (\mathtt{IsRand}(B) = 0 \text{ OR } (T, h(B)) \in L_R)] \\
\leq \; & \Pr[\mathcal{A} \sim (T, B, C) : (T, B) \text{ was not a query to } \mathsf{Write}(k, \cdot, \cdot) \text{ AND } \mathtt{IsRand}(B) = 0] + \\
& \Pr[\mathcal{A} \sim (T, B, C) : (T, B) \text{ was not a query to } \mathsf{Write}(k, \cdot, \cdot) \text{ AND } (T, h(B)) \in L_R].
\end{aligned}$$

Denote the last of these probabilities by p_1 and p_2, respectively:

$$p_1 = \Pr[\mathcal{A} \sim (T, B, C) : (T, B) \text{ was not a query to } \mathsf{Write}(k, \cdot, \cdot) \text{ AND } \mathtt{IsRand}(B) = 0];$$

$$p_2 = \Pr[\mathcal{A} \sim (T, B, C) : (T, B) \text{ was not a query to } \mathsf{Write}(k, \cdot, \cdot) \text{ AND } (T, h(B)) \in L_R].$$

We try to upper bound each of these two probabilities. In order to bound p_1, let's compute the probability that $\mathcal{A}$ outputs a block identifier T and a ciphertext C for which $\pi^{-1}(T, C)$ is considered not random by the entropy test. $\mathcal{A}$ makes q_1 queries to $\mathsf{Write}(k, \cdot, \cdot)$. From the $\mathsf{int\text{-}st}$ definition of adversary's success, $\mathcal{A}$ cannot output (T, C) such that C was received from a query to $\mathsf{Write}(k, T, \cdot)$. If $\mathcal{A}$ picks a $C \in \mathcal{M}$, then $\mathtt{IsRand}(\pi_k^{-1}(T, C)) = 0$ with probability α, the false negative rate of the entropy test. So, in $\mathcal{M}$, there are $\alpha|\mathcal{M}| = \alpha 2^l$ ciphertexts for which $\mathtt{IsRand}(\pi_k^{-1}(T, C)) = 0$. $\mathcal{A}$ makes q_2 queries to Check, and it can make one more guess to output if the decryption of none of those resulting plaintexts B satisfies $\mathtt{IsRand}(B) = 0$. Thus:

$$p_1 \leq \frac{(q_2 + 1)\alpha 2^l}{2^l - q_1}.$$

For bounding p_2, if $(T, h(B)) \in L_R$ and (T, B) was not a query to $\mathsf{Write}(k, \cdot, \cdot)$, then there exists $B' \in \mathcal{M}$ such that (T, B') was a query to $\mathsf{Write}(k, \cdot, \cdot)$ and $h(B) = h(B')$. Then there exists an adversary $\mathcal{B}$ for h such that $p_2 \leq \mathsf{Adv}^{\mathsf{spr}}_{h,\mathcal{B}}$.

To conclude, we have:

$$\Pr[\pi \xleftarrow{R} \mathsf{Perm}^{\mathcal{T}}(\mathcal{M}), \mathcal{D}^{\pi,\pi^{-1}} = 1] \leq p_1 + p_2 \leq \frac{(q_2+1)\alpha 2^l}{2^l - q_1} + \mathsf{Adv}^{\mathsf{spr}}_h(\tau),$$

and

$$\begin{aligned} \mathsf{Adv}^{\mathsf{int\text{-}st}}_{\mathsf{SRAND\text{-}BINT},\mathcal{A}} &= \mathsf{Adv}^{\mathsf{prp}}_{\mathsf{E},\mathcal{D}} + \Pr[\pi \xleftarrow{R} \mathsf{Perm}^{\mathcal{T}}(\mathcal{M}), \mathcal{D}^{\pi,\pi^{-1}} = 1] \\ &\leq \mathsf{Adv}^{\mathsf{prp}}_{\mathsf{E},\mathcal{D}} + \mathsf{Adv}^{\mathsf{spr}}_h(\tau) + \frac{(q_2+1)\alpha 2^l}{2^l - q_1}. \end{aligned}$$

The statement of the theorem follows from the last relation. ▯

Scheme Secure Against Replay Attacks RAND-BINT

The construction we propose here stems from the observation that the block access distribution in practice is not uniformly random, in fact it follows a Zipf-like distribution. More specifically, there are few blocks that are written more than once, with the majority of the blocks being written just once. If all the blocks were written only once, then scheme SRAND-BINT would suffice to defend against replays, as well. If the blocks were written uniformly, then scheme HASH-BINT could be used. The solution we give here is a hybrid scheme, that combines the previous two constructions.

Briefly, the solution is to keep in trusted storage a counter for each block that is written more than once. The counter denotes the number of writes to a particular block. We compute the tweak as a function of the block identifier and the counter, so that if a block is written more than once, it is encrypted every time with a different tweak. After computing the tweak as indicated, the scheme proceeds as in SRAND-BINT: at each Write operation, if the message has high entropy, then its hash is stored in a list L_R in trusted storage. A message is considered valid if either it has low entropy or its hash is stored in L_R. The intuition for the correctness of this scheme is that decryptions of the same ciphertext using the same key, but different tweaks, are independent. Thus, if the server replies with an older version of an encrypted block, the client uses a different tweak for decrypting it than the one with which it was originally encrypted. Then, the chances that it still yields a low-entropy plaintext are small. The detailed scheme RAND-BINT is given in Figure 2.7.

Init():	Write(k, bid, B):	Check(k, bid, C):
$k \xleftarrow{R} \mathcal{K}_\mathsf{E}$	UpdateCtr(bid)	$T \leftarrow bid \Vert$GetCtr(bid)
$L_R \leftarrow \emptyset$	$T \leftarrow bid \Vert$GetCtr(bid)	$B \leftarrow \mathsf{D}_k^T(C)$
	if $(bid, hval) \in L_R$	if $\mathtt{IsRand}(B) = 0$
	remove $(bid, hval)$ from L_R	return$B$
	if $\mathtt{IsRand}(B) = 1$	else
	insert $(bid, h(B))$ into L_R	if $(bid, h(B)) \in L_R$
	WriteBlock($bid, C \leftarrow \mathsf{E}_k^T(B)$)	return B
		else
		return $\perp$

Figure 2.7: Scheme RAND-BINT.

The proof for the integrity of RAND-BINT is similar to that of the integrity of SRAND-BINT. Here we just state the theorem that relates the security of RAND-BINT to the PRP security of E, the everywhere second preimage resistance of the hash function and the false negative rate of the entropy test.

Theorem 3 *If* E *is a PRP-secure tweakable enciphering scheme with the plaintext size l bits,* D *is its inverse, h is an everywhere second preimage resistant hash function and α (the false negative rate of* `IsRand`*) is small, then* RAND-BINT *is* int-st-rep *secure:*

$$\mathsf{Adv}^{\mathsf{int\text{-}st}}_{\mathsf{SRAND\text{-}BINT}}(\tau, q_1, q_2) \leq \mathsf{Adv}^{\mathsf{prp}}_{\mathsf{E}}(\tau, q_1, q_2) + \mathsf{Adv}^{\mathsf{spr}}_{h}(\tau) + \frac{(q_2+1)\alpha 2^l}{2^l - q_1}.$$

2.6.3 The Compression Scheme COMP-BINT

This construction is again based on the intuition that many workloads feature low-entropy blocks, but attempts to exploit this in a different way. In this construction, the block is compressed and padded before encryption. If the compression level of the block content is high enough, then a message-authentication code of the block can be stored in the block itself, reducing the amount of storage necessary for integrity. In order to prevent replay attacks, it is necessary that the message-authentication code is computed over block contents and block write counters that need to be authenticated in trusted storage. For blocks that can not be compressed enough, the block identifier together with the block hash is stored in a list L_R in trusted storage.

The authentication information (i.e., tags) for blocks that can be compressed is stored on untrusted storage, and consequently a message-authentication code is required. In con-

trast, the hashes of the blocks that cannot be compressed enough are stored in trusted storage, and thus an unkeyed hash is enough to authenticate these blocks.

The advantage of this construction compared to the SRAND-BINT and RAND-BINT constructions is that here blocks can be encrypted with any length-preserving encryption scheme, and not necessarily with a tweakable cipher.

In the description of the integrity scheme, we assume we are given a compression algorithm compress and a decompression algorithm decompress such that decompress(compress$(m)) = m$, for any message m. We can also pad messages up to a certain fixed length by using the pad function, and unpad a padded message with the unpad function such that unpad(pad$(m)) = m$, for all messages m of length less than the output of the pad function. We can use standard padding methods for implementing these algorithms (Black and Urtubia [2002]). We also need a hash function $h : \{0,1\}^* \rightarrow \{0,1\}^s$ and a message-authentication code HMAC : $\mathcal{K}_{\mathsf{HMAC}} \times \{0,1\}^* \rightarrow \{0,1\}^s$ with key space $\mathcal{K}_{\mathsf{HMAC}}$ that outputs messages of length s bits.

The Init, Write, and Check algorithms of the COMP-BINT construction are detailed in Figure 2.8. Here Z is the byte length of the largest plaintext size for which the ciphertext is of length at most the file block length less the size of a hash function output. For example, if the block size is 4096 bytes, SHA-1 is used for hashing (whose output is 20 bytes) and 16-byte AES is used for encryption, then Z is the largest multiple of the AES block size (i.e., 16 bytes) less than $4096 - 20 = 4076$ bytes. The value of Z in this case is 4064 bytes.

Init():	Write($<k_1, k_2>, bid, B$):	Check($<k_1, k_2>, bid, C$):
$k_1 \leftarrow \mathsf{Gen}^{\mathsf{len}}()$ $k_2 \leftarrow \mathcal{K}_{\mathsf{HMAC}}$ $L_R \leftarrow \emptyset$	UpdateCtr(bid) if $(bid, hval) \in L_R$ remove $(bid, hval)$ from L_R $B^c \leftarrow$ compress(B) if $\|B^c\| \leq Z$ $C \leftarrow \mathsf{E}^{\mathsf{len}}_{k_1}(bid, \mathsf{pad}(B^c))$ WriteBlock($bid, C\|\|\mathsf{HMAC}_{k_2}(bid\|\|$ GetCtr(bid)$\|\|B$)) else insert $(bid, h(bid\|\|\mathsf{GetCtr}(bid)\|\|B)))$ into L_R $C \leftarrow \mathsf{E}^{\mathsf{len}}_{k_1}(bid, B)$ WriteBlock(bid, C)	if $(bid, hval) \in L_R$ $B \leftarrow \mathsf{D}^{\mathsf{len}}_{k_1}(bid, C)$ if $h(bid\|\|\mathsf{GetCtr}(bid)\|\|B) = hval$ return B else return $\perp$ else parse C as $C'\|\|hval$ $B^c \leftarrow \mathsf{unpad}(\mathsf{D}^{\mathsf{len}}_{k_1}(bid, C'))$ $B \leftarrow$ decompress(B^c) if $\mathsf{HMAC}_{k_2}(bid\|\|\mathsf{GetCtr}(bid)\|\|B) = hval$ return B else return $\perp$

Figure 2.8: Scheme COMP-BINT.

Theorem 4 *If h is an everywhere second preimage resistant hash function and* HMAC *is a* cma*-secure message-authentication code, then* COMP-BINT *is* int-st-rep *secure:*

$$\mathsf{Adv}^{\mathsf{int\text{-}st\text{-}rep}}_{\mathsf{COMP\text{-}BINT}}(\tau, q_1, q_2) \leq \mathsf{Adv}^{\mathsf{spr}}_{h}(\tau) + \mathsf{Adv}^{\mathsf{cma}}_{\mathsf{HMAC}}(\tau, q_1, q_2).$$

Proof: Assume there is an adversary $\mathcal{A}$ for COMP-BINT with advantage $\mathsf{Adv}^{\mathsf{int\text{-}st\text{-}rep}}_{\mathsf{COMP\text{-}BINT},\mathcal{A}}$ running in time τ and making q_1 queries to the Write oracle and q_2 queries to the Check oracle. From Definition 6, it follows that $\mathcal{A}$ outputs a block identifier bid and an encrypted block content C such that: (1) $\mathsf{Check}(<k_1, k_2>, bid, C)$ returns $B \neq \perp$; and (2) C was not received by $\mathcal{A}$ in the last query to $\mathsf{Write}(<k_1, k_2>, bid, \cdot)$.

For these conditions to be satisfied, one of the following two cases occurs:

1. Assuming that the last block written by $\mathcal{A}$ to block bid cannot be compressed, $\mathcal{A}$ finds a different block content that hashes to the value stored in trusted storage in the list L_R. The probability of this event is upper bounded by $\mathsf{Adv}^{\mathsf{spr}}_{h}(\tau)$.

2. Assuming that the last block written to block bid can be compressed, $\mathcal{A}$ finds a different block content that hashes under key k_2 to the value stored at the end of the block. The probability of this event is upper bounded by $\mathsf{Adv}^{\mathsf{cma}}_{\mathsf{HMAC}}(\tau, q_1, q_2)$.

From the two cases, it follows that the probability of success of $\mathcal{A}$ is upper bounded by $\mathsf{Adv}^{\mathsf{spr}}_{h}(\tau) + \mathsf{Adv}^{\mathsf{cma}}_{\mathsf{HMAC}}(\tau, q_1, q_2)$, and thus the conclusion of the theorem follows. ◻

2.7 The Entropy Statistical Test

In this section we give an example of a statistical test `IsRand` that can distinguish between random and non-random blocks. $\texttt{IsRand}(M), M \in \mathcal{M}$ returns 1 with high probability if M is a uniformly random block in $\mathcal{M}$, and 0 otherwise. Consider a block M divided into n parts of fixed length $M = M_1 M_2 \dots M_n$ with $M_i \in \{1, 2, \dots, b\}$. For example, a 1024-byte block could be either divided into 1024 8-bit parts (for $b = 256$), or alternatively into 2048 4-bit parts (for $b = 16$). The empirical entropy of M is defined as $\mathsf{Ent}(M) = -\sum_{i=1}^{b} p_i \log_2(p_i)$, where p_i is the frequency of symbol i in the sequence $M_1, \dots, M_n$.

If we fix a threshold $\mathsf{TH}_{\mathsf{Ent}}$ depending on n and b (we will discuss later how to choose $\mathsf{TH}_{\mathsf{Ent}}$), then the entropy test parameterized by b and $\mathsf{TH}_{\mathsf{Ent}}$ is defined in Figure 2.9. In the following, we call $\texttt{IsRand}_{8,\mathsf{TH}_{\mathsf{Ent}}}(\cdot)$ the *8-bit entropy test* and $\texttt{IsRand}_{4,\mathsf{TH}_{\mathsf{Ent}}}(\cdot)$ the *4-bit entropy test*.

Write M as $M = M_1 M_2 \ldots M_n$ with $M_i \in \{1, 2, \ldots, b\}$ Compute p_i = the frequency of symbol i in M, $i = 1, \ldots, b$ Compute $\mathsf{Ent}(M) = -\sum_{i=1}^{b} p_i \log_2(p_i)$ If $\mathsf{Ent}(M) < \mathsf{TH}_{\mathsf{Ent}}$, then return 0 Else return 1

Figure 2.9: The entropy test $\mathtt{IsRand}_{b,\mathsf{TH}_{\mathsf{Ent}}}(M)$.

Analysis. We give an analytical bound for the false negative rate, as a function of n, b and $\mathsf{TH}_{\mathsf{Ent}}$.

Theorem 5 *For a given threshold* $\mathsf{TH}_{\mathsf{Ent}}$*, if we denote* δ *the solution of the following equation:*

$$\mathsf{TH}_{\mathsf{Ent}} = -(1-\delta)\log_2\left(\frac{1-\delta}{b}\right), \tag{2.1}$$

then the false negative rate α *of* $\mathtt{IsRand}_{b,\mathsf{TH}_{\mathsf{Ent}}}(\cdot)$ *satisfies:*

$$\alpha \leq b e^{-\frac{n}{2b}\delta^2} + b\left(\frac{1}{e}\right)^{\frac{n}{b}}\left(\frac{4e}{b}\right)^{\frac{n}{4}}. \tag{2.2}$$

In the proof of Theorem 5, we use the following well-known lemma.

Lemma 1 (Union bound) *Assume* $A_1, \ldots, A_m$ *are some events, not necessarily independent. Then:*

1. $\Pr[A_1 \cup \cdots \cup A_m] \leq \sum_{i=1}^{m} \Pr[A_i]$;
2. $\Pr[A_1 \cap \cdots \cap A_m] \geq 1 - \sum_{i=1}^{m}(1 - \Pr[A_i])$.

We can now proceed to the proof of Theorem 5.

Proof: The false negative rate of `IsRand` is by definition:

$$\alpha = \Pr[\mathsf{Ent}(R = R_1 R_2 \ldots R_n) \leq \mathsf{TH}_{\mathsf{Ent}}],$$

for $R_1, R_2, \ldots, R_n$ uniformly random in $\{1, 2, \ldots, b\}$.

Consider a fixed $i \in \{1, 2, \ldots, b\}$. To each R_j we associate a 0-1 variable X_j with the property: $X_j = 1 \Leftrightarrow R_j = i$. $X_1, \ldots, X_n$ are independent and the mean of each X_j is $\mathbf{E}(X_j) = \frac{1}{b}$.

Define $f_i = \sum_{j=1}^{n} X_j$, i.e., f_i denotes the number of blocks that have value i. From the independence of $X_1, \ldots, X_n$, it follows that $\mathbf{E}(f_i) = \sum_{i=1}^{n} \mathbf{E}(X_j) = \frac{n}{b}$. We also define $p_i = \frac{f_i}{n}$. Then the entropy of R is $\mathsf{Ent}(R) = -\sum_{i=1}^{b} p_i \log_2(p_i)$.

Applying the Chernoff bounds for f_i, it follows that for any $\delta \in (0, 1]$:

$$\Pr\Big[p_i < \frac{1}{b}(1-\delta)\Big] = \Pr\Big[f_i < \frac{n}{b}(1-\delta)\Big] = \Pr\Big[f_i < \mathbf{E}(f_i)(1-\delta)\Big] < e^{\frac{-\mathbf{E}(f_i)\delta^2}{2}} = e^{-\frac{n}{2b}\delta^2}.$$

Assume that all p_i are less or equal to $\frac{1}{4}$. The function $x \to -x \log_2(x)$ is monotonically increasing on the interval $(0, \frac{1}{4}]$. Therefore, if $p_i \geq \frac{1}{b}(1-\delta)$, then:

$$\mathsf{Ent}(R) = -\sum_{i=1}^{b} p_i \log_2(p_i) \geq -(1-\delta)\log_2\Big(\frac{1-\delta}{b}\Big) = \mathsf{TH}_{\mathsf{Ent}}. \tag{2.3}$$

For a fixed i, it follows from the Chernoff bounds that $p_i \geq \frac{1}{b}(1-\delta)$ with probability at least $1 - e^{-\frac{n}{2b}\delta^2}$. Applying the union bound, it follows that:

$$\Pr\Big[p_i \geq \frac{1}{b}(1-\delta), \forall i = 1, \ldots, b\Big] \geq 1 - be^{-\frac{n}{2b}\delta^2}. \tag{2.4}$$

(2.3) and (2.4) imply that:

$$\Pr\Big[\mathsf{Ent}(R) \geq \mathsf{TH}_{\mathsf{Ent}} | p_i \leq \frac{1}{4}, i = 1, \ldots, b\Big] \geq 1 - be^{-\frac{n}{2b}\delta^2}. \tag{2.5}$$

Let's bound the probability $\Pr[p_i \leq \frac{1}{4}, i = 1, \ldots, b]$ using Chernoff bounds. Denote $\epsilon = \frac{b}{4} - 1$. Then for a fixed $i \in \{1, \ldots, b\}$, we have:

$$\begin{aligned}
\Pr\Big[p_i > \frac{1}{4}\Big] &= \Pr\Big[p_i > \frac{1}{b}(1 + \frac{b}{4} - 1)\Big] \\
&= \Pr\Big[f_i > \frac{n}{b}(1 + \frac{b}{4} - 1)\Big] < \left[\frac{e^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right]^{\frac{n}{b}} \\
&= \left(\frac{1}{e}\right)^{\frac{n}{b}} \left(\frac{4e}{b}\right)^{\frac{n}{4}}.
\end{aligned}$$

Using the union bound, it follows that:

$$\Pr\Big[p_i \leq \frac{1}{4}, i = 1, \ldots, b\Big] \geq 1 - b\left(\frac{1}{e}\right)^{\frac{n}{b}} \left(\frac{4e}{b}\right)^{\frac{n}{4}}. \tag{2.6}$$

Using (2.5) and (2.6), we get:

$$
\begin{aligned}
& \Pr[\mathsf{Ent}(R) \geq \mathsf{TH}_{\mathsf{Ent}}] \\
= \; & \Pr\Big[\mathsf{Ent}(R) \geq \mathsf{TH}_{\mathsf{Ent}} | p_i \leq \frac{1}{4}, i = 1, \ldots, b\Big] \cdot \Pr\Big[p_i \leq \frac{1}{4}, i = 1, \ldots, b\Big] \\
+ \; & \Pr\Big[\mathsf{Ent}(R) \geq \mathsf{TH}_{\mathsf{Ent}} | \exists\, i = 1, \ldots, b \text{ st } p_i \leq \frac{1}{4}\Big] \cdot \Pr\Big[\exists\, i = 1, \ldots, b \text{ st } p_i \leq \frac{1}{4}\Big] \\
\geq \; & \big[1 - b e^{-\frac{n}{2b}\delta^2}\big] \left[1 - b \left(\frac{1}{e}\right)^{\frac{n}{b}} \left(\frac{4e}{b}\right)^{\frac{n}{4}}\right].
\end{aligned}
$$

In conclusion:

$$
\begin{aligned}
\alpha &= \Pr[\mathsf{Ent}(R) \leq \tau] = 1 - \Pr[\mathsf{Ent}(R) \geq \tau] \\
&\leq 1 - \big[1 - b e^{-\frac{n}{2b}\delta^2}\big] \left[1 - b \left(\frac{1}{e}\right)^{\frac{n}{b}} \left(\frac{4e}{b}\right)^{\frac{n}{4}}\right] \\
&\leq b e^{-\frac{n}{2b}\delta^2} + b \left(\frac{1}{e}\right)^{\frac{n}{b}} \left(\frac{4e}{b}\right)^{\frac{n}{4}}.
\end{aligned}
$$

□

Discussion. The derived bound for the false negative rate of the entropy test is mostly of theoretical interest. It demonstrates that the false negative rate decreases exponentially with the increase of the block length. However, for small values of n such as 1KB or 4KB, the bound does not give tight constraints on the false negative rate of the entropy test.

2.8 Evaluation

In order to project the behavior of the integrity algorithms proposed in practice, we collected approximately 200 MB of block disk traces from a SuSe Linux environment. They represent a sampling of disk activity from one machine during one month. The applications used most frequently were: the Netscape browser and e-mail client, the g++ compiler, the Java compiler from Sun, the XMMS audio player, the image viewer GIMP, the text editor Kedit, the LATEX compiler, the Acrobat and GV viewers, and the VNC server. The reasons we collected traces only from one computer are two-fold: first, getting the traces proved to be cumbersome, as it required non-trivial kernel modification and the installation of a new drive to store them. Secondly, collecting disk traces from users has strong

implications for users' privacy. The block sizes used by the disk interface were 1024 and 4096 bytes.

We implemented the HASH-BINT, RAND-BINT and COMP-BINT integrity algorithms and evaluated the amount of storage necessary for integrity. The performance evaluation of the similar constructions for file systems is done in the context of a cryptographic file system in Section 3.6. We choose the threshold $\mathsf{TH}_{\mathsf{Ent}}$ experimentally as follows. We generated 100,000 uniformly random blocks and computed, for each, its entropy. Then, we pick $\mathsf{TH}_{\mathsf{Ent}}$ smaller than the the minimum entropy of all the random blocks generated. For 1KB blocks, the 8-bit entropy threshold is set to 7.68 and the 4-bit entropy threshold is 3.96. For 4KB blocks, the thresholds are set to 7.9 and 3.98 for the 8-bit and 4-bit entropy tests, respectively.

In the trace we collected, there were 813,124 distinct block addresses, from which only 276,560 were written more than once. Therefore, a counter has to be stored for about 35% of the block addresses. The block access distribution is represented in Figure 2.10. In this graph, the fraction of blocks written i times is represented for $i = 1, 2, \ldots, 14$. For $i = 15$, the graph accumulates all the blocks that have been written at least 15 times.

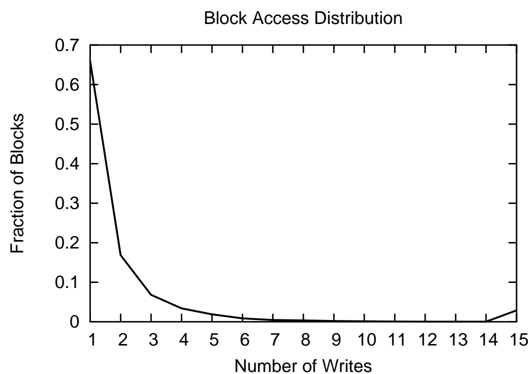

Figure 2.10: Block access distribution.

From the block contents in the trace, 0.56% are random-looking by using the 8-bit entropy test, 0.87% are random-looking by using the 4-bit entropy test and 0.68% are not compressible enough to be able to fit a message-authentication code inside the block. The amount of storage that the client needs to keep for the three schemes secure against replay attacks is given in Figure 2.11. We only considered here the amount of storage for RAND-BINT using the better 8-bit entropy test. The amount of storage of RAND-BINT and COMP-BINT are 1.86% and 1.98%, respectively, of that needed by HASH-BINT.

Of course, the client storage increases with the lifetime of the system, as more blocks

are overwritten. One solution to prevent the indefinite expansion of client state is to periodically change the encryption key, re-encrypt all the data under the new key, recompute all the integrity information and reset all the block flags.

Storage for HASH-BINT	Storage for RAND-BINT	Storage for COMP-BINT
854.43 KB	15.92 KB	16.98 KB

Figure 2.11: Client storage for integrity for the three replay-secure algorithms.

2.9 Related Work

Encryption algorithms for secure storage have received much attention in recent years, leading to the development of *tweakable* block ciphers (Liskov et al. [2002]). In addition to a key and a plaintext, a tweakable block cipher has as input a *tweak*, for variability. In the secure storage model, a tweak might be the address of a disk block or the block identifier. This notion has been extended to that of tweakable enciphering schemes (Halevi and Rogaway [2003]) that operate on larger plaintexts (e.g., 512 or 1024 bytes). Constructions of tweakable enciphering schemes designed recently include CMC (Halevi and Rogaway [2003]), the parallelizable EME (Halevi and Rogaway [2004]) and XCB (McGrew and Fluhrer [2004]).

Adopting one of these tweakable encryption schemes for confidentiality, our goal is to augment it to provide efficient integrity for the storage scenario. Therefore, there are two main orthogonal fields related to our work: authenticated encryption and storage security.

Authenticated encryption. Authenticated encryption (e.g., (Bellare and Namprempre [2000], Katz and Yung [2001], Krawczyk [2001])) is a primitive that provides privacy and message authenticity at the same time. That is, in addition to providing some notion of encryption scheme privacy (e.g., Bellare et al. [1998]), authenticated encryption ensures either integrity of plaintexts or integrity of ciphertexts. The traditional approach for constructing authenticated encryption is by generic composition, i.e., the combination of a secure encryption scheme and an unforgeable message-authentication code (MAC). However, Bellare and Namprempre (Bellare and Namprempre [2000]) analyze the security of the composition and provide proofs that some of the widely believed secure compositions are actually insecure. Krawczyk (Krawczyk [2001]) proves that the generic composition method used in the Secure Socket Layer (SSL) protocol is insecure, but the particular standard implementation is secure with respect to both privacy and integrity. The authenticated

encryption in SSH is also insecure, as demonstrated by Bellare et al. (Bellare et al. [2002]). There, a new definition for integrity is given, that protects against replay and out-of-order delivery attacks; Kohno et al. (Kohno et al. [2003]) also supply such definitions. While we also define integrity against replay attacks, our definitions are particularly tailored to the storage scenario, and are thus different from the network case.

A different approach to obtain integrity is to add redundancy to plaintexts. Bellare and Rogaway (Bellare and Rogaway [2000]) and Ann and Bellare (Ann and Bellare [2001]) give necessary and sufficient conditions for the redundancy code such that the composition of the encryption scheme and the redundancy code provide integrity.

Integrity in block-level storage systems. We elaborate on methods for data integrity in storage systems in Section 3.8 in the next chapter.

Chapter 3

Integrity in Cryptographic File Systems with Constant Trusted Storage

In this chapter, we construct new algorithms for integrity protection of files in cryptographic file systems that use only a constant amount of trusted storage (Oprea and Reiter [2006c]). The new integrity algorithms exploit two characteristics of many file-system workloads, namely low entropy of file contents and high sequentiality of file block writes. The algorithms extend the RAND-BINT and COMP-BINT constructions from Section 2.6.2 to the setting of a cryptographic file system where individual file blocks need to be authenticated with a small amount of trusted storage and a low integrity bandwidth (i.e., the amount of information needed to update or check the integrity of a file block). Both algorithms achieve their storage efficiency by using counters for file blocks (that denote the number of writes to that block) for replay protection. Based on real NFS traces collected at Harvard University (Ellard et al. [2003]), we propose an efficient representation method for counters that reduces the additional space needed for integrity. We evaluate the performance and storage requirements of our new constructions compared to those of the standard integrity algorithm based on Merkle trees. We conclude with guidelines for choosing the best integrity algorithm depending on typical application workload.

We describe the well known Merkle tree algorithm in Section 3.1. After elaborating on our system model in Section 3.2, we investigate two counter representation methods and perform an analysis of their storage requirements in Section 3.3. We detail the two new integrity algorithms for encrypted file systems and analyze their security in Section 3.4. Our prototype architecture is given in Section 3.5 and our performance evaluation in Section 3.6. We conclude the chapter in Section 3.7 with a discussion of the suitability of each integrity algorithm to particular classes of workloads. We also discuss there an alter-

native application of the integrity algorithms proposed. Lastly, we provide in Section 3.8 a comparison of the different integrity methods used in existing storage systems.

3.1 Preliminaries

In this section we review the use of Merkle trees for authenticating a set of data items, and show how they can be applied to provide integrity of files in file systems. We describe several algorithms on a Merkle tree for a file that will be used later in our constructions for integrity in encrypted file systems.

Merkle trees (Merkle [1989]) are used to authenticate n data items with constant-size trusted storage. A Merkle tree for data items $M_1, \ldots, M_n$, denoted $\mathsf{MT}(M_1, \ldots, M_n)$, is a binary tree that has $M_1, \ldots, M_n$ as leaves. An interior node of the tree with children C_L and C_R is the hash of the concatenation of its children (i.e., $h(C_L || C_R)$, for h a collision-resistant hash function). The value stored in the root of the tree thus depends on all the values stored in the leaves of the tree and can be used to authenticate all the leaf values. If the root of the tree is stored in trusted storage, then all the leaves of the tree can be authenticated by reading $\mathcal{O}(\log(n))$ hashes from the tree. The size of the Merkle tree that authenticates a set of data items is double the number of items authenticated, but it can be reduced using a higher degree Merkle tree instead of a binary tree. However, increasing the degree of the Merkle tree might increase the number of hash values needed to authenticate a data item. In our work, we only consider binary Merkle trees.

We define the Merkle tree for a file F with n blocks $B_1, \ldots, B_n$ to be the binary tree $\mathsf{MT}_F = \mathsf{MT}(h(1||B_1), \ldots, h(n||B_n))$. The file block index is hashed together with the block content in order to prevent block swapping attacks in some of our constructions, in which a client considers valid the content of a different file block than the one he intended to read. A Merkle tree with a given set of leaves can be constructed in multiple ways. Here we choose to append a new block in the tree as a right-most child, so that the tree has the property that all the left subtrees are complete. Before giving the details of the algorithms used to append a new leaf in the tree, update the value stored at a leaf, check the integrity of a leaf and delete a leaf from the tree, we give first a small example.

Example. We give an example in Figure 3.1 of a Merkle tree for a file that initially has six blocks: $B_1, \ldots, B_6$. We also show how the Merkle tree for the file is modified when block B_7 is appended to the file.

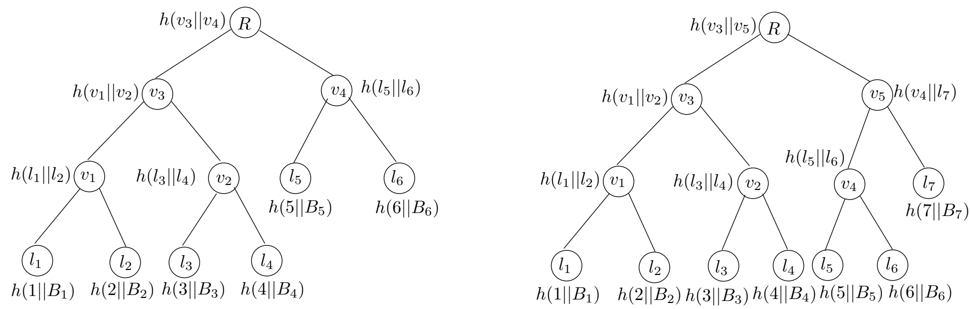

Figure 3.1: Merkle tree for a file with 6 blocks on the left; after block 7 is appended on the right.

Notation. For a tree T, $T.\mathsf{root}$ is the root of the tree, $T.\mathsf{no_leaves}$ is the number of leaves in T and $T.\mathsf{leaf}[i]$ is the i-th leaf in the tree counting from left to right. For a node v in the tree, $v.\mathsf{hash}$ is the hash value stored at the node and for a non-leaf node v, $v.\mathsf{left}$ and $v.\mathsf{right}$ are pointers to the left and right children of v, respectively. $v.\mathsf{sib}$ and $v.\mathsf{parent}$ denote the sibling and parent of node v, respectively.

Merkle tree algorithms. The algorithms UpdatePathRoot, CheckPathRoot, UpdateTree, CheckTree, AppendTree and DeleteTree on a Merkle tree T are described in Figure 3.2.

- The $\mathsf{UpdatePathRoot}(\mathsf{R}, v)$ algorithm for tree T is called after a modification of the hash value stored at node v. In this algorithm, the hashes stored in the nodes on the path from node v to the root of the tree are updated. For that, all the hashes stored in the siblings of those nodes are read. Finally, the hash from the root of T is stored in R.

- In the $\mathsf{CheckPathRoot}(\mathsf{R}, v)$ algorithm for tree T, the hashes stored in the nodes on the path from node v to the root of the tree are computed, by reading all the hashes stored in the siblings of those nodes. Finally, the hash of the root of T is checked to match the parameter R. If the hash stored in the root matches the parameter R, then the client is assured that all the siblings read from the tree are authentic.

- In the $\mathsf{UpdateTree}(\mathsf{R}, i, hval)$ algorithm for tree T, the hash stored at the i-th leaf of T is updated to $hval$. This triggers an update of all the hashes stored on the path from the i-th leaf to the root of the tree and an update of the value stored in R with algorithm UpdatePathRoot. It is necessary to first check that all the siblings of the

```
T.UpdatePathRoot(R, v):
  while v ≠ T.root
    s ← v.sib
    if v.parent.left = v
      v.parent.hash ← h(v.hash||s.hash)
    else
      v.parent.hash ← h(s.hash||v.hash)
    v ← v.parent
  R ← T.root.hash
```

```
T.CheckPathRoot(R, v):
  hval ← v.hash
  while v ≠ T.root
    s ← v.sib
    if v.parent.left = v
      hval ← h(hval||s.hash)
    else
      hval ← h(s.hash||hval)
    v ← v.parent
  if R = hval
    return true
  else
    return false
```

```
T.UpdateTree(R, i, hval):
  if T.CheckPathRoot(R, T.leaf[i]) = true
    T.leaf[i].hash ← hval
    T.UpdatePathRoot(R, T.leaf[i])
```

```
T.CheckTree(R, i, hval):
  if T.leaf[i].hash ≠ hval
    return false
  return T.CheckPathRoot(R, T.leaf[i])
```

```
T.AppendTree(R, hval):
  u.hash ← hval
  u.left ← null; u.right ← null
  depth ← ⌈log_2(T.no_leaves)⌉
  d ← 0; v ← T.root
  while v.right and d < depth
    v ← v.right; d ← d + 1
  if d = depth
    p ← T.root
  else
    p ← v.parent
  if T.CheckPathRoot(R, p) = true
    p.left ← p
    p.right ← u
    p.hash ← h(p.hash||u.hash)
    T.UpdatePathRoot(R, p)
```

```
T.DeleteTree(R):
  v ← T.root
  while v.right
    v ← v.right
  if v = T.root
    T ← null
  else
    p ← v.parent
    if T.CheckPathRoot(R, p) = true
      if p = T.root
        T.root ← p.left
      else
        p.parent.right ← p.left
      T.UpdatePathRoot(R, p)
```

Figure 3.2: The UpdateTree, CheckTree, AppendTree and DeleteTree algorithms for a Merkle tree T.

nodes on the path from the updated leaf to the root of the tree are authentic. This is done by calling the algorithm CheckPathRoot.

- The CheckTree$(\mathsf{R}, i, hval)$ algorithm for tree T checks that the hash stored at the i-th leaf matches $hval$. Using algorithm CheckPathRoot, all the hashes stored at the nodes on the path from the i-th leaf to the root are computed and the root of T is

checked finally to match the value stored in R.

- Algorithm $\mathsf{AppendTree}(\mathsf{R}, hval)$ for tree T appends a new leaf u that stores the hash value $hval$ to the tree. The right-most node v in the tree is searched first. If the tree is already a complete tree, then the new leaf is added as the right child of the root of the tree and the existing tree becomes the left child of the new root. Otherwise, the new leaf is appended as the right child of the parent p of v, and the subtree rooted at p becomes the left child of p. This way, for any node in the Merkle tree, the left subtree is complete and the tree determined by a set of leaves is unique. Upon completion, the parameter R keeps the hash from the root of T.

- The $\mathsf{DeleteTree}(\mathsf{R})$ algorithm for tree T deletes the last leaf from the tree. First, the right-most node v in the tree is searched. If this node is the root itself, then the tree becomes null. Otherwise, node v is removed from the tree and the subtree rooted at the sibling of v is moved in the position of v's parent. Upon completion, the parameter R keeps the hash from the root of T.

3.2 System Model

We consider the model of a cryptographic file system that provides random access to files. Encrypted data is stored on untrusted storage servers and there is a mechanism for distributing the cryptographic keys to authorized parties. A small (on the order of several hundred bytes), fixed-size per file, *trusted storage* is available for authentication data.

We assume that the storage servers are actively controlled by an adversary. The adversary can adaptively alter the data stored on the storage servers or perform any other attack on the stored data, but it cannot modify or observe the trusted storage. A particularly interesting attack that the adversary can mount is a *replay attack*, in which stale data is returned to read requests of clients. Using the trusted storage to keep some constant-size information per file, and keeping more information per file on untrusted storage, our goal is to design and evaluate different integrity algorithms that allow the update and verification of individual blocks in files and that detect data modification and replay attacks.

In our framework, a file F is divided into n fixed-size blocks $B_1 B_2 \dots B_n$ (the last block B_n might be shorter than the first $n-1$ blocks), each encrypted individually with the encryption key of the file and stored on the untrusted storage servers (n differs per file). The constant-size, trusted storage for file F is denoted TS_F. Additional storage for file F, which can reside in untrusted storage, is denoted US_F.

Similar to the model in the previous chapter, the storage interface provides two basic operations to the clients: $F.\mathsf{WriteBlock}(i, C)$ stores content C at block index i in file F and $C \leftarrow F.\mathsf{ReadBlock}(i)$ reads (encrypted) content from block index i in file F. In addition, in an integrity scheme for an encrypted file, a client can compute or check the integrity of a file block.

Definition 7 (Integrity schemes for encrypted files) *An integrity scheme for an encrypted file F consists of five operations $F.\mathsf{IntS} = (F.\mathsf{Init}, F.\mathsf{Update}, F.\mathsf{Append}, F.\mathsf{Check}, F.\mathsf{Delete})$, where:*

1. *In the initialization algorithm* Init *for file F, the encryption key for the file is generated.*
2. *In the update operation* $\mathsf{Update}(i, B)$ *for file F, an authorized client updates the i-th block in the file with the length-preserving encryption of block content B and updates the integrity information for the i-th block stored in TS_F and US_F.*
3. *In the append operation* $\mathsf{Append}(B)$ *for file F, a new block that contains the length-preserving encryption of B is appended to the encrypted file. Integrity information for the new block is stored in TS_F and US_F.*
4. *In the check operation* $\mathsf{Check}(i, C)$ *for file F, an authorized client checks that the (decrypted) block content C read from the i-th block in file F (by calling $C \leftarrow F.\mathsf{ReadBlock}(i)$) is authentic, using the additional storage TS_F and US_F for file F. The check operation returns the decrypted block if the client accepts the block content as authentic and $\perp$ otherwise.*
5. *The* Delete *operation deletes the last block in a file and updates the integrity information for the file.*

Remark. An integrity scheme for encrypted block-level storage defined in Definition 5 consists of only three algorithms: Init, Write and Check. In defining an integrity scheme for an encrypted file, we divide the Write algorithm into the Update and Append algorithms. The Update algorithm is called when an existing file block is modified, and the Append algorithm is used to append new file blocks to the file. We also added a new Delete algorithm useful for shrinking file sizes, which is not necessary in block-level storage where the space of available block addresses is fixed over time.

Using the algorithms we have defined for an integrity scheme for an encrypted file, a client can read or write at any byte offset in the file. For example, to write to a byte offset

that is not at a block boundary, the client first reads the block the byte offset belongs to, decrypts it and checks its integrity using algorithm Check. Then, the client constructs the new data blocks by replacing the appropriate bytes in the decrypted block, encrypts the new block and updates its integrity information using algorithm Update.

In designing an integrity algorithm for a cryptographic file system, we consider the following metrics. First is the size of the untrusted storage US_F; we will always enforce that the trusted storage TS_F is of constant size, independent of the number of blocks. Second is the integrity bandwidth for updating and checking individual file blocks, defined as the number of bytes from US_F accessed (updated or read) when accessing a block of file F, averaged over either: all blocks in F when we speak of a per-file integrity bandwidth; all blocks in all files when we speak of the integrity bandwidth of the file system; or all blocks accessed in a particular trace when we speak of one trace. Third is the performance cost of writing and reading files.

3.3 Write Counters for File Blocks

In Chapter 2, block write counters are used for replay protection in the new block-level integrity algorithms proposed and for constructing length-preserving stateful encryption schemes. The integrity algorithms for cryptographic file systems are immediate extensions of the corresponding algorithms for block storage systems, and as such they make use of file block counters in the same two ways. The write counters of the blocks in a file F support two basic operations: $F.\mathsf{UpdateCtr}(i)$ and $F.\mathsf{GetCtr}(i)$, which are similar to the operations defined in Section 2.3.

When counters are used for encryption, they can be safely stored in the untrusted storage space for a file. However, in the case in which counters protect against replay attacks, they need to be authenticated with a small amount of trusted storage. We define two algorithms for authenticating block write counters, both of which are invoked by an authorized client:

- Algorithm AuthCtr modifies the trusted storage space TS_F of file F to contain the trusted authentication information for the write counters of F.
- Algorithm CheckCtr checks the authenticity of the counters stored in US_F using the trusted storage TS_F for file F and returns true if the counters are authentic and false, otherwise.

A problem that needs to be addressed in the design of the various integrity algorithms

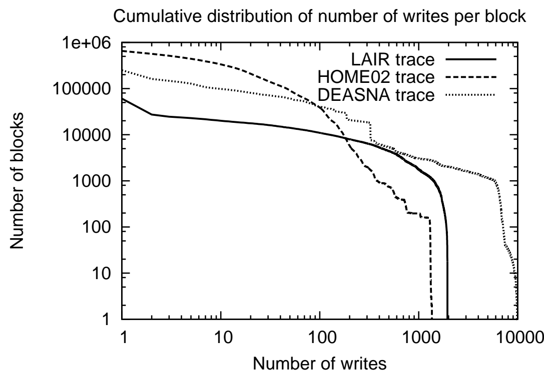

Figure 3.3: Cumulative distribution of number of writes per block.

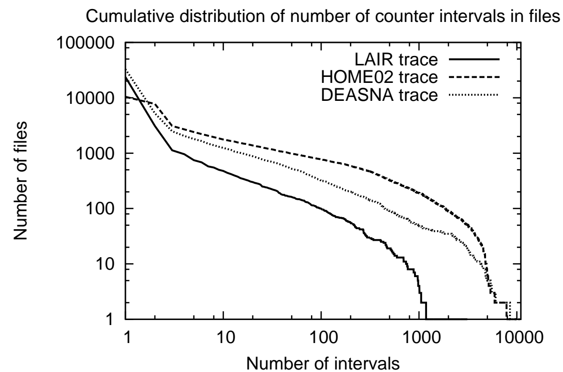

Figure 3.4: Cumulative distribution of number of counter intervals in files.

described below is the storage and authentication of the block write counters. If a counter per file block were used, this would result in significant additional storage for counters. We investigate more efficient methods of storing the block write counters, based on analyzing the file access patterns in NFS traces collected at Harvard University (Ellard et al. [2003]).

Counter intervals. We performed some experiments on NFS Harvard traces (Ellard et al. [2003]) in order to analyze the file access patterns. We considered three different traces (LAIR, DEASNA and HOME02) for a period of one week. The LAIR trace consists from research workload traces from Harvard's computer science department. The DEASNA trace is a mix of research and email workloads from the division of engineering and applied sciences at Harvard. HOME02 is mostly the email workload from the campus general purpose servers.

We first plotted the cumulative distribution of the number of writes per block for each trace in Figure 3.3 in a log-log scale (i.e., for x number of writes on the x-axis, the y axis represents the number of blocks that have been written at least x times).

Ellard et al. (Ellard et al. [2003]) make the observation that a large number of file accesses are sequential. This leads to the idea that the values of the write counters for adjacent blocks in a file might be correlated. To test this hypothesis, we represent counters for blocks in a file using *counter intervals*. A counter interval is defined as a sequence of consecutive blocks in a file that all share the same value of the write counter. For a counter interval, we need to store only the beginning and end of the interval, and the value of the write counter. We plot the cumulative distribution of the number of counter intervals per files for the three Harvard traces in Figure 3.4 (i.e., for x number of intervals on the x-axis,

the y axis represents the number of files that have at least x counter intervals).

Figure 3.4 validates the hypothesis that for the large majority of files in the three traces considered, a small number of counter intervals needs to be stored and only for few files the number of counter intervals is large (i.e., over 1000). The average storage per file using counter intervals is several orders of magnitude smaller than that used by storing a counter per block, as shown in Table 3.1. This justifies our design choice to use counter intervals for representing block counters in the integrity algorithms proposed.

	LAIR	DEASNA	HOME02
Counter per block	1.79 MB	2.7 MB	8.97 MB
Counter intervals	5 bytes	9 bytes	163 bytes

Table 3.1: Average storage per file for two counter representation methods.

Counter representation. The counter intervals for file F are represented by two arrays: $\mathsf{IntStart}_F$ keeps the block indices where new counter intervals start and CtrVal_F keeps the values of the write counter for each interval. The trusted storage TS_F for file F includes either the arrays $\mathsf{IntStart}_F$ and CtrVal_F if they fit into TS_F or their hashes, otherwise. In the limit, to reduce the bandwidth for integrity, we could build a Merkle tree to authenticate each of these arrays and store the root of these trees in TS_F, but we have not seen in the Harvard traces files that would warrant this.

We omit here the implementation details for the UpdateCtr, GetCtr and DelCtr operations on counters (which are immediate), but describe the algorithms for authenticating counters with a constant amount of trusted storage. Assume that the length of available trusted storage for counters for file F is $\mathsf{L}_{\mathsf{ctr}}$. For an array A, A.size is the number of bytes needed for all the elements in the array and $h(\mathsf{A})$ is the hash of concatenated elements in the array. We also store in trusted storage a flag ctr-untr whose value is true if the counter arrays $\mathsf{IntStart}_F$ and CtrVal_F are stored in the untrusted storage space of F and false otherwise. The AuthCtr and CheckCtr algorithms are described in Figure 3.5.

If the counter intervals for a file get too dispersed, then the size of the arrays $\mathsf{IntStart}_F$ and CtrVal_F might increase significantly. To keep the untrusted storage for integrity low, we could periodically change the encryption key for the file, re-encrypt all blocks in the file, and reset the block write counters to 0.

F.AuthCtr():	F.CheckCtr():

if $\mathsf{IntStart}_F$.size + CtrVal_F.size > $\mathsf{L}_{\mathsf{ctr}}$
  store $h_1 = h(\mathsf{IntStart}_F)$ and $h_2 = h(\mathsf{CtrVal}_F)$ in TS_F
  store $\mathsf{IntStart}_F$ and CtrVal_F in US_F
  ctr-untr = true
else
  store $\mathsf{IntStart}_F$ and CtrVal_F in TS_F
  ctr-untr = false
	if ctr-untr = true
	  get $\mathsf{IntStart}_F$ and CtrVal_F from US_F
	  get h_1 and h_2 from TS_F
	  if $h_1 = h(\mathsf{IntStart}_F)$ and $h_2 = h(\mathsf{CtrVal}_F)$
	    return true
	  else
	    return false
	else
	  get $\mathsf{IntStart}_F$ and CtrVal_F from TS_F
	  return true

Figure 3.5: The AuthCtr and CheckCtr algorithms for counter intervals.

3.4 Integrity Constructions for Encrypted Storage

In this section, we first present a Merkle tree integrity construction MT-FINT for encrypted storage, used in file systems such as Cepheus (Fu [1999]), FARSITE (Adya et al. [2002]), and Plutus (Kallahalla et al. [2003]). Second, we extend the RAND-BINT and COMP-BINT constructions for integrity in block storage systems from Section 2.6 to algorithms RAND-FINT and COMP-FINT for integrity protection in cryptographic file systems. The challenge in constructing integrity algorithms for cryptographic file systems is to reduce the amount of trusted storage needed per file to a constant value.

In the MT-FINT and COMP-FINT algorithms, file blocks are encrypted with a length-preserving stateful encryption scheme. We extend the notion defined in Section 2.4 so that the encryption and decryption algorithms take as input parameters a file and a block index instead of a block identifier. The two implementations of a length-preserving stateful encryption scheme given in Section 2.4 can be easily extended to this case. In the RAND-FINT algorithm, a tweakable enciphering scheme as defined in Section 2.1.1 is used for encrypting file blocks.

3.4.1 The Merkle Tree Construction MT-FINT

In this construction, file blocks can be encrypted with any length-preserving stateful encryption scheme and they are authenticated with a Merkle tree. More precisely, if $F = B_1 \dots B_n$ is a file with n blocks, then the untrusted storage for integrity for file F is $\mathsf{US}_F = \mathsf{MT}(h(1||B_1), \dots, h(n||B_n))$ (for h an everywhere second preimage resistant hash function), and the trusted storage TS_F is the root of this tree.

The algorithm Init runs the key generation algorithm $\mathsf{Gen}^{\mathsf{len}}$ of the length-preserving stateful encryption scheme for file F. The algorithms Update, Check, Append and Delete of the $\mathsf{MT\text{-}FINT}$ construction are given in Figure 3.6. We denote here by $F.\mathsf{enc_key}$ the encryption key for file F (generated in the Init algorithm) and $F.\mathsf{blocks}$ the number of blocks in file F.

- In the $\mathsf{Update}(i, B)$ algorithm for file F, the i-th leaf in MT_F is updated with the hash of the new block content using the algorithm $\mathsf{UpdateTree}$ and the encryption of B is stored in the i-th block of F.
- To append a new block B to file F with algorithm $\mathsf{Append}(B)$, a new leaf is appended to MT_F with the algorithm $\mathsf{AppendTree}$, and then an encryption of B is stored in the $(n+1)$-th block of F (for n the number of blocks of F).
- In the $\mathsf{Check}(i, C)$ algorithm for file F, block C is decrypted, and its integrity is checked using the $\mathsf{CheckTree}$ algorithm.
- To delete the last block from a file F with algorithm Delete, the last leaf in MT_F is deleted with the algorithm $\mathsf{DeleteTree}$.

$F.\mathsf{Update}(i, B)$: $k \leftarrow F.\mathsf{enc_key}$ $\mathsf{MT}_F.\mathsf{UpdateTree}(\mathsf{TS}_F, i, h(i\|\|B))$ $C \leftarrow \mathsf{E}_k^{\mathsf{len}}(F, i, B)$ $F.\mathsf{WriteBlock}(i, C)$	$F.\mathsf{Check}(i, C)$: $k \leftarrow F.\mathsf{enc_key}$ $B_i \leftarrow \mathsf{D}_k^{\mathsf{len}}(F, i, C)$ if $\mathsf{MT}_F.\mathsf{CheckTree}(\mathsf{TS}_F, i, h(i\|\|B_i)) = \mathsf{true}$ return B_i else return $\perp$
$F.\mathsf{Append}(B)$: $k \leftarrow F.\mathsf{enc_key}$ $n \leftarrow F.\mathsf{blocks}$ $\mathsf{MT}_F.\mathsf{AppendTree}(\mathsf{TS}_F, h(n+1\|\|B))$ $C \leftarrow \mathsf{E}_k^{\mathsf{len}}(F, n+1, B)$ $F.\mathsf{WriteBlock}(n+1, C)$	$F.\mathsf{Delete}()$: $n \leftarrow F.\mathsf{blocks}$ $\mathsf{MT}_F.\mathsf{DeleteTree}(\mathsf{TS}_F)$ delete B_n from file F

Figure 3.6: The Update, Check, Append and Delete algorithms for the $\mathsf{MT\text{-}FINT}$ construction.

The $\mathsf{MT\text{-}FINT}$ construction detects data modification and block swapping attacks, as file block contents are authenticated by the root of the Merkle tree for each file. The $\mathsf{MT\text{-}FINT}$ construction is also secure against replay attacks, as the tree contains the hashes of the latest version of the data blocks and the root of the Merkle tree is authenticated in trusted storage.

3.4.2 The Randomness Test Construction RAND-FINT

Whereas in the Merkle tree construction any length-preserving stateful encryption algorithm can be used to individually encrypt blocks in a file, the randomness test construction uses the observation from Chapter 2 that the integrity of the blocks that are efficiently distinguishable from random blocks can be checked with a randomness test if a tweakable cipher is used to encrypt them. As such, integrity information is stored only for random-looking blocks.

In this construction, a Merkle tree per file that authenticates the contents of the random-looking blocks is built. The untrusted storage for integrity US_F for file $F = B_1 \dots B_n$ includes this tree $\mathsf{RTree}_F = \mathsf{MT}(h(i||B_i) : i \in \{1, \dots, n\}$ and $\mathtt{IsRand}(B_i) = 1)$, and, in addition, the set of block indices that are random-looking $\mathsf{RArr}_F = \{i \in \{1, \dots, n\} : \mathtt{IsRand}(B_i) = 1\}$, ordered the same as the leaves in the previous tree RTree_F. The root of the tree RTree_F is kept in the trusted storage TS_F for file F.

To prevent against replay attacks, clients need to distinguish different writes of the same block in a file. A simple idea (similar to that used in construction RAND-BINT for encrypted block-level storage in Section 2.6.2) is to use a counter per file block that denotes the number of writes of that block, and make the counter part of the encryption tweak. The block write counters need to be authenticated in the trusted storage space for the file F to prevent clients from accepting valid older versions of a block that are considered not random by the randomness test (see Section 3.3 for a description of the algorithms AuthCtr and CheckCtr used to authenticate and check the counters, respectively). To ensure that file blocks are encrypted with different tweaks, we define the tweak for a file block to be a function of the file, the block index and the block write counter. We denote by F.Tweak the tweak-generating function for file F that takes as input a block index and a block counter and outputs the tweak for that file block. The properties of tweakable ciphers imply that if a block is decrypted with a different counter, then it will look random with high probability.

The algorithm Init selects a key at random from the key space of the tweakable encryption scheme E. The Update, Check, Append and Delete algorithms of RAND-FINT are detailed in Figure 3.7. For the array RArr_F, RArr_F.items denotes the number of items in the array, RArr_F.last denotes the last element in the array, and the function RArr_F.SearchIndex(i) gives the position in the array where index i is stored (if it exists in the array).

- In the Update(i, B) algorithm for file F, the write counter for block i is incremented and the counter authentication information from TS_F is updated with the algorithm

F.Update(i, B) :
  $k \leftarrow F$.enc_key
  F.UpdateCtr(i)
  F.AuthCtr$()$
  if IsRand$(B) = 0$
    if $i \in$ RArr$_F$
      RTree$_F$.DelIndexTree(TS$_F$, RArr$_F$, i)
  else
    if $i \in$ RArr$_F$
      $j \leftarrow$ RArr$_F$.SearchIndex(i)
      RTree$_F$.UpdateTree(TS$_F$, j, $h(i\|\|B)$)
    else
      RTree$_F$.AppendTree(TS$_F$, $h(i\|\|B)$)
      append i at end of RArr$_F$
  F.WriteBlock$(i, \mathsf{E}_k^{F.\mathsf{Tweak}(i, F.\mathsf{GetCtr}(i))}(B))$
	F.Check(i, C):
	  $k \leftarrow F$.enc_key
	  if F.CheckCtr$()$ = false
	    return $\perp$
	  $B_i \leftarrow \mathsf{D}_k^{F.\mathsf{Tweak}(i, F.\mathsf{GetCtr}(i))}(C)$
	  if IsRand$(B_i) = 0$
	    return B_i
	  else
	    if $i \in$ RArr$_F$
	      $j \leftarrow$ RArr$_F$.SearchIndex(i)
	      if RTree$_F$.CheckTree(TS$_F$, j, $h(i\|\|B_i)$) = true
	        return B_i
	      else
	        return $\perp$
	    else
	      return $\perp$

F.Append(B):
  $k \leftarrow F$.enc_key
  $n \leftarrow F$.blocks
  F.UpdateCtr$(n+1)$
  F.AuthCtr$()$
  if IsRand$(B) = 1$
    RTree$_F$.AppendTree(TS$_F$, $h(n+1\|\|B)$)
    append $n+1$ at end of RArr$_F$
  F.WriteBlock$(n+1, \mathsf{E}_k^{F.\mathsf{Tweak}(n+1, F.\mathsf{GetCtr}(n+1))}(B))$
	F.Delete$()$:
	  $n \leftarrow F$.blocks
	  F.DelCtr(n)
	  F.AuthCtr$()$
	  if $n \in$ RArr$_F$
	    RTree$_F$.DelIndexTree(TS$_F$, RArr$_F$, n)
	  delete B_n from file F

Figure 3.7: The Update, Check, Append and Delete algorithms for the RAND-FINT construction.

AuthCtr. Then, the randomness test IsRand is applied to block content B. If B is not random looking, then the leaf corresponding to block i (if it exists) has to be removed from RTree$_F$. This is done with the algorithm DelIndexTree, described in Figure 3.8. On the other hand, if B is random-looking, then the leaf corresponding to block i has to be either updated with the new hash (if it exists in the tree) or appended in RTree$_F$. Finally, the tweakable encryption of B is stored in the i-th block of F.

- When a new block B is appended to file F with algorithm Append(B), the write counter for the new block is initialized and the authentication information for counters is updated. Furthermore, the hash of the block index concatenated with the block content is added to RTree$_F$ only if the block is random-looking. In addition, the index $n+1$ (where n is the current number of blocks in F) is added to RArr$_F$ in this case. Finally, the tweakable encryption of B is stored in the $(n+1)$-th block

```
T.DelIndexTree(TS_F, RArr_F, i):
  j ← RArr_F.SearchIndex(i)
  l ← RArr_F.last
  if j ≠ l
    T.UpdateTree(TS_F, j, h(l||B_l))
    RArr_F[j] ← l
    RArr_F.items ← RArr_F.items − 1
  T.DeleteTree(TS_F)
```

Figure 3.8: The DelIndexTree algorithm for a tree T deletes the hash of block i from T and moves the last leaf in its position, if necessary, to not allow holes in the tree.

of F.

- In the Check(i, C) algorithm for file F, the authentication information from TS_F for the block counters is checked first. Then block C is decrypted, and checked for integrity. If the content of the i-th block is not random-looking, then by the properties of tweakable ciphers we can infer that the block is valid. Otherwise, the integrity of the i-th block is checked using the tree RTree_F. If i is not a block index in the tree, then the integrity of block i is unconfirmed and the block is rejected.

- To delete the last block in file F with algorithm Delete, the write counter for the last block is deleted and the authentication information for the counters is updated. If the n-th block is authenticated through RTree_F, then its hash has to be removed from the tree by calling the algorithm DelIndexTree, described in Figure 3.8.

It is not necessary to authenticate in trusted storage the array RArr_F of indices of the random-looking blocks in a file. The reason is that the root of RTree_F is authenticated in trusted storage and this implies that an adversary cannot modify the order of the leaves in RTree_F without being detected in the AppendTree, UpdateTree or CheckTree algorithms.

The construction RAND-FINT protects against unauthorized modification of data written to disk and block swapping attacks by authenticating the root of RTree_F in the trusted storage space for each file. By using write counters in the encryption of block contents and authenticating the values of the counters in trusted storage, this construction provides defense against replay attacks and provides all the security properties of the MT-FINT construction.

3.4.3 The Compress-and-Hash Construction COMP-FINT

This construction is an extension of the integrity construction COMP-BINT for block storage systems and uses the ideas from integrity algorithm RAND-FINT to reduce the amount of trusted storage per file to a constant value. For the blocks that can be compressed enough, a message-authentication is stored inside the block. A Merkle tree RTree_F is built over the hashes of the blocks in file F that cannot be compressed enough, and the root of the tree is kept in trusted storage. To prevent against replay attacks, block indices and contents are hashed together with the block write counter and the write counters for a file F need to be authenticated in the trusted storage space TS_F (see Section 3.3 for a description of the algorithms AuthCtr and CheckCtr used to authenticate and check the counters, respectively).

The algorithm Init runs the key generation algorithm $\mathsf{Gen}^{\mathsf{len}}$ of the length-preserving stateful encryption scheme for file F to generate key k_1 and selects at random a key k_2 from the key space $\mathcal{K}_{\mathsf{HMAC}}$ of HMAC. It outputs the tuple $<k_1, k_2>$. The Update, Append, Check and Delete algorithms of the COMP-FINT construction are detailed in Figure 3.9.

- In the $\mathsf{Update}(i, B)$ algorithm for file F, the write counter for block i is incremented and the counter authentication information from TS_F is updated with the algorithm AuthCtr. Then block content B is compressed to B^c. If the length of B^c (denoted $|B^c|$) is at most the threshold Z defined in Section 2.6.3, then there is room to store the message-authentication code of the block content inside the block. In this case, the hash of the previous block content stored at the same address is deleted from the Merkle tree RTree_F, if necessary. The compressed block is padded and encrypted, and then stored with its message-authentication code in the i-th block of F. Otherwise, if the block cannot be compressed enough, then its hash has to be inserted into the Merkle tree RTree_F. The block content B is then encrypted with a length-preserving stateful encryption scheme using the key for the file and is stored in the i-th block of F.

- To append a new block B to file F with n blocks using the $\mathsf{Append}(B)$ algorithm, the write counter for the new block is initialized to 1 and the authentication information for counters stored in TS_F is updated. Block B is then compressed. If it has an adequate compression level, then the compressed block is padded and encrypted, and a message-authentication code is concatenated at the end of the new block. Otherwise, a new hash is appended to the Merkle tree RTree_F and an encryption of B is stored in the $(n + 1)$-th block of F.

- In the $\mathsf{Check}(i, C)$ algorithm for file F, the authentication information from TS_F

<table>
<tr>
<td>
F.Update(i, B) :
$< k_1, k_2 > \leftarrow F$.enc_key
F.UpdateCtr(i)
F.AuthCtr$()$
$B^c \leftarrow \mathsf{compress}(B)$
if $|B^c| \leq Z$
if $i \in \mathsf{RArr}_F$
$\mathsf{RTree}_F.\mathsf{DelIndexTree}(\mathsf{TS}_F, \mathsf{RArr}_F, i)$
$C \leftarrow \mathsf{E}^{\mathsf{len}}_{k_1}(F, i, \mathsf{pad}(B^c))$
$F.\mathsf{WriteBlock}(i, C||\mathsf{HMAC}_{k_2}(i||F.\mathsf{GetCtr}(i)||B))$
else
if $i \in \mathsf{RArr}_F$
$j \leftarrow \mathsf{RArr}_F.\mathsf{SearchIndex}(i)$
$\mathsf{RTree}_F.\mathsf{UpdateTree}(\mathsf{TS}_F, j, h(i||F.\mathsf{GetCtr}(i)||B))$
else
$\mathsf{RTree}_F.\mathsf{AppendTree}(\mathsf{TS}_F, h(i||F.\mathsf{GetCtr}(i)||B))$
append i at end of RArr_F
$C \leftarrow \mathsf{E}^{\mathsf{len}}_{k_1}(F, i, B)$
$F.\mathsf{WriteBlock}(i, C)$
</td>
<td>
F.Check(i, C):
$< k_1, k_2 > \leftarrow F$.enc_key
if $F.\mathsf{CheckCtr}() = \mathsf{false}$
return $\perp$
if $i \in \mathsf{RArr}_F$
$B_i \leftarrow \mathsf{D}^{\mathsf{len}}_{k_1}(F, i, C)$
$j \leftarrow \mathsf{RArr}_F.\mathsf{SearchIndex}(i)$
if $\mathsf{RTree}_F.\mathsf{CheckTree}(\mathsf{TS}_F, j,$
$h(i||F.\mathsf{GetCtr}(i)||B_i)) = \mathsf{true}$
return B_i
else
return $\perp$
else
parse C as $C'||hval$
$B_i^c \leftarrow \mathsf{unpad}(\mathsf{D}^{\mathsf{len}}_{k_1}(F, i, C'))$
$B_i \leftarrow \mathsf{decompress}(B_i^c)$
if $hval = \mathsf{HMAC}_{k_2}(i||F.\mathsf{GetCtr}(i)||B_i)$
return B_i
else
return $\perp$
</td>
</tr>
<tr>
<td>
F.Append(B) :
$< k_1, k_2 > \leftarrow F$.enc_key
$n \leftarrow F$.blocks
F.UpdateCtr$(n+1)$
F.AuthCtr$()$
$B^c \leftarrow \mathsf{compress}(B)$
if $|B^c| \leq Z$
$C \leftarrow \mathsf{E}^{\mathsf{len}}_{k_1}(F, n+1, \mathsf{pad}(B^c))$
$F.\mathsf{WriteBlock}(i, C||\mathsf{HMAC}_{k_2}(n+1||F.\mathsf{GetCtr}(n+1)||B))$
else
$\mathsf{RTree}_F.\mathsf{AppendTree}(\mathsf{TS}_F, h(n+1||F.\mathsf{GetCtr}(n+1)||B))$
append $n+1$ at end of RArr_F
$C \leftarrow \mathsf{E}^{\mathsf{len}}_{k_1}(F, n+1, B)$
$F.\mathsf{WriteBlock}(n+1, C)$
</td>
<td>
F.Delete$()$:
$n \leftarrow F$.blocks
F.DelCtr(n)
F.AuthCtr$()$
if $n \in \mathsf{RArr}_F$
$\mathsf{RTree}_F.\mathsf{DelIndexTree}(\mathsf{TS}_F, \mathsf{RArr}_F, n)$
delete B_n from file F
</td>
</tr>
</table>

Figure 3.9: The Update, Check, Append and Delete algorithms for the COMP-FINT construction.

for the block counters is checked first. There are two cases to consider. First, if the hash of the block content stored at the i-th block of F is authenticated through the Merkle tree RTree_F, then the block is decrypted and algorithm CheckTree is called. Otherwise, the message-authentication code of the block content is stored at the end of the block and we can thus parse the i-th block of F as $C'||hval$. C' has to be decrypted, unpadded and decompressed, in order to obtain the original

block content B_i. The hash value $hval$ stored in the block is checked to match the hash of the block index i concatenated with the write counter for block i and block content B_i.

- To delete the last block from file F with n blocks, algorithm Delete is used. The write counter for the block is deleted and the authentication information for counters is updated. If the n-th block is authenticated through RTree_F, then its hash has to be removed from the tree by calling the algorithm DelIndexTree.

The construction COMP-FINT prevents against replay attacks by hashing write counters for file blocks together with block indices and block contents and authenticating the write counters in trusted storage. It meets all the security properties of RAND-FINT and MT-FINT.

3.5 Implementation

Our integrity algorithms are very general and they can be integrated into any cryptographic file system in either the kernel or userspace. For the purpose of evaluating and comparing their performance, we implemented them in EncFS (Gough [2003]), an open-source user-level file system that transparently encrypts file blocks. EncFS uses the FUSE (FUSE) library to provide the file system interface. FUSE provides a simple library API for implementing file systems and it has been integrated into recent versions of the Linux kernel.

In EncFS, files are divided into fixed-size blocks and each block is encrypted individually. Several ciphers such as AES and Blowfish in CBC mode are available for block encryption. We implemented in EncFS the three constructions that provide integrity: MT-FINT, RAND-FINT and COMP-FINT. While any length-preseving encryption scheme can be used in the MT-FINT and COMP-FINT constructions, RAND-FINT is constrained to use a tweakable cipher for encrypting file blocks. We choose to encrypt file blocks in MT-FINT and COMP-FINT with the length-preserving stateful encryption derived from the AES cipher in CBC mode (as shown in Section 2.4), and use the CMC tweakable cipher (Halevi and Rogaway [2003]) as the encryption method in RAND-FINT. For compressing and decompressing blocks in COMP-FINT we used the zlib library (Zlib).

Our prototype architecture is depicted in Figure 3.10. We modified the user space of EncFS to include the CMC cipher for block encryption and the new integrity algorithms. The server uses the underlying file system (i.e., reiserfs) for the storage of the

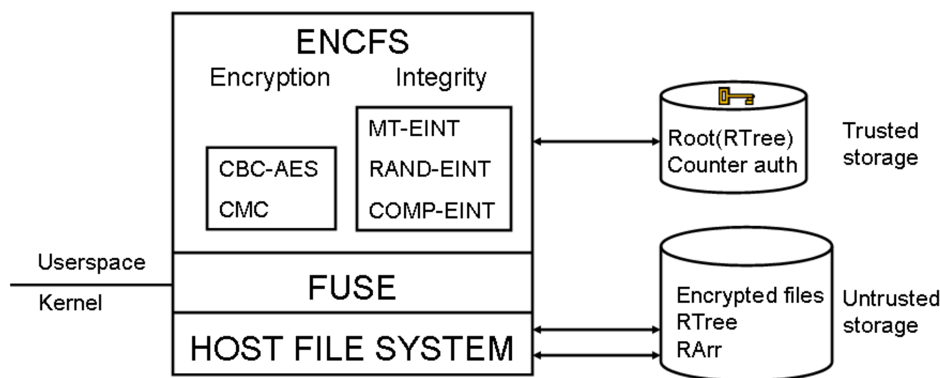

Figure 3.10: Prototype architecture.

encrypted files. The Merkle trees for integrity RTree_F and the index arrays of the random-looking blocks RArr_F are stored with the encrypted files in the untrusted storage space on the server. For faster integrity checking (in particular to improve the running time of the SearchIndex algorithm used in the Update and Check algorithms of the RAND-FINT and COMP-FINT constructions), we also keep the array RArr_F for each file, ordered by indices, in untrusted storage. The roots of the trees RTree_F, and the arrays $\mathsf{IntStart}_F$ and CtrVal_F or their hashes (if they are too large) are stored in the trusted storage space. In our current implementation, we use two extended attributes for each file F, one for the root of RTree_F and the second for the arrays $\mathsf{IntStart}_F$ and CtrVal_F, or their hashes (see the counter authentication algorithm from Section 3.3).

By default, EncFS caches the last block content written or read from the disk. In our implementation, we cached the last arrays RArr_F, $\mathsf{IntStart}_F$ and CtrVal_F used in a block update or check operation. Since these arrays are typically small (a few hundred bytes), they easily fit into memory. We also evaluate the effect of caching of Merkle trees in our system in Section 3.6.1.

3.6 Performance Evaluation

In this section, we evaluate the performance of the new integrity constructions for encrypted storage RAND-FINT and COMP-FINT compared to that of MT-FINT. We ran our experiments on a 2.8 GHz Intel D processor machine with 1GB of RAM, running SuSE Linux 9.3 with kernel version 2.6.11 and file system reiserfs. The hard disk used was an 80GB SATA 7200 RPM Maxtor. To obtain accurate results in our experiments, we use a cold cache at each run.

The main challenge we faced in evaluating the proposed constructions was to come up with representative file system workloads. While the performance of the Merkle tree construction is predictable independently of the workload, the performance of the new integrity algorithms is highly dependent on the file contents accessed, in particular on the randomness of block contents. To our knowledge, there are no public traces that contain file access patterns, as well as the contents of the file blocks read and written. Due to the privacy implications of releasing actual users' data, we expect it to be nearly impossible to get such traces from a widely used system. However, we have access to three public NFS Harvard traces (Ellard et al. [2003]) that contain NFS traffic from several of Harvard's campus servers. The traces were collected at the level of individual NFS operations and for each read and write operation they contain information about the file identifier, the accessed offset in the file and the size of the request, but not the actual block contents.

To evaluate the integrity algorithms proposed in this paper, we perform two sets of experiments. In the first one, we strive to demonstrate how the performance of the new constructions varies for different file contents. For that, we use representative files from a Linux distribution installed on one of our desktop machines, together with other files from the user's home directory, divided into several file types. We identify five file types of interest: text, object, executables, images, and compressed files, and divide the collected files according to these five classes. All files of a particular type are first encrypted and the integrity information for them is built; then they are decrypted and checked for integrity. We report the performance results for the files with the majority of blocks not random-looking (i.e., text, executable and object) and for those with mostly random-looking blocks (i.e., image and compressed). In this experiment, all files are written and read sequentially, and as such the access pattern is not a realistic one.

In the second set of experiments, we evaluate the effect of more realistic access patterns on the performance of the integrity schemes, using the NFS Harvard traces. As the Harvard traces do not contain information about actual file block contents written to the disks, we generate synthetic block contents for each block write request. We define two types of block contents: low-entropy and high-entropy, and perform experiments assuming that all block contents are either low or high entropy. These extreme workloads represent the "best" and "worst"-case for the new algorithms, respectively. We also consider a "middle"-case, in which a block is random-looking with a 50% probability and plot the performance results of the new schemes relative to the Merkle tree integrity algorithm for the best, middle and worst cases.

In our performance evaluation, it would be beneficial to know what is the percentage of random-looking blocks in practical filesystem workloads. To determine statistics on file contents, we perform a user study on several machines from our department running Linux

and report the results in Section 3.6.3. For each user machine, we measure the percent of random-looking blocks and the percent of blocks that cannot be compressed enough from users' home directories. The results show that on average, 28% percent of file blocks are random-looking and 32% percent of file blocks cannot be compressed enough to fit a MAC inside.

Finally, we also evaluate how the amount of trusted storage per file affects the storage of the counter intervals in the three NFS Harvard traces. As discussed in Section 3.3, the counter intervals are either stored in the trusted storage space of a file if enough space is available or they are stored in the untrusted storage space of the file and their hashes are stored in trusted storage. We show that several hundred bytes of trusted storage per file is enough to keep the counter intervals for a large majority of files from the NFS Harvard traces.

3.6.1 The Impact of File Block Contents on Integrity Performance

File sets. We consider a snapshot of the file system from one of our desktop machines. We gathered files that belong to five classes of interest: (1) *text files* are files with extensions .txt, .tex, .c, .h, .cpp, .java, .ps, .pdf; (2) *object files* are system library files from the directory /usr/local/lib; (3) *executable files* are system executable files from directory /usr/local/bin; (4) *image files* are JPEG files and (5) *compressed files* are gzipped tar archives. Several characteristics of each set, including the total size, the number of files in each set, the minimum, average and maximum file sizes and the fraction of file blocks that are considered random-looking by the entropy test are given in Table 3.2.

	Total size	No files	Min file size	Max file size	Avg file size	Fraction of random-looking blocks
Text	245 MB	808	27 bytes	34.94 MB	307.11 KB	0.0351
Objects	217 MB	28	15 bytes	92.66 MB	7.71 MB	0.0001
Executables	341 MB	3029	24 bytes	13.21 MB	112.84 KB	0.0009
Image	189 MB	641	17 bytes	2.24 MB	198.4 KB	0.502
Compressed	249 MB	2	80.44 MB	167.65 MB	124.05 MB	0.7812

Table 3.2: File set characteristics.

Experiments. We consider three cryptographic file systems: (1) MT-FINT with CBC-AES for encrypting file blocks; (2) RAND-FINT with CMC encryption; (3) COMP-FINT with CBC-AES encryption. For each cryptographic file system, we first write the files from each set; this has the effect of automatically encrypting the files, and running the Update algorithm of the integrity method for each file block. Second, we read all files

from each set; this has the effect of automatically decrypting the files, and running the Check algorithm of the integrity method for each file block. We use file blocks of size 4KB in the experiments.

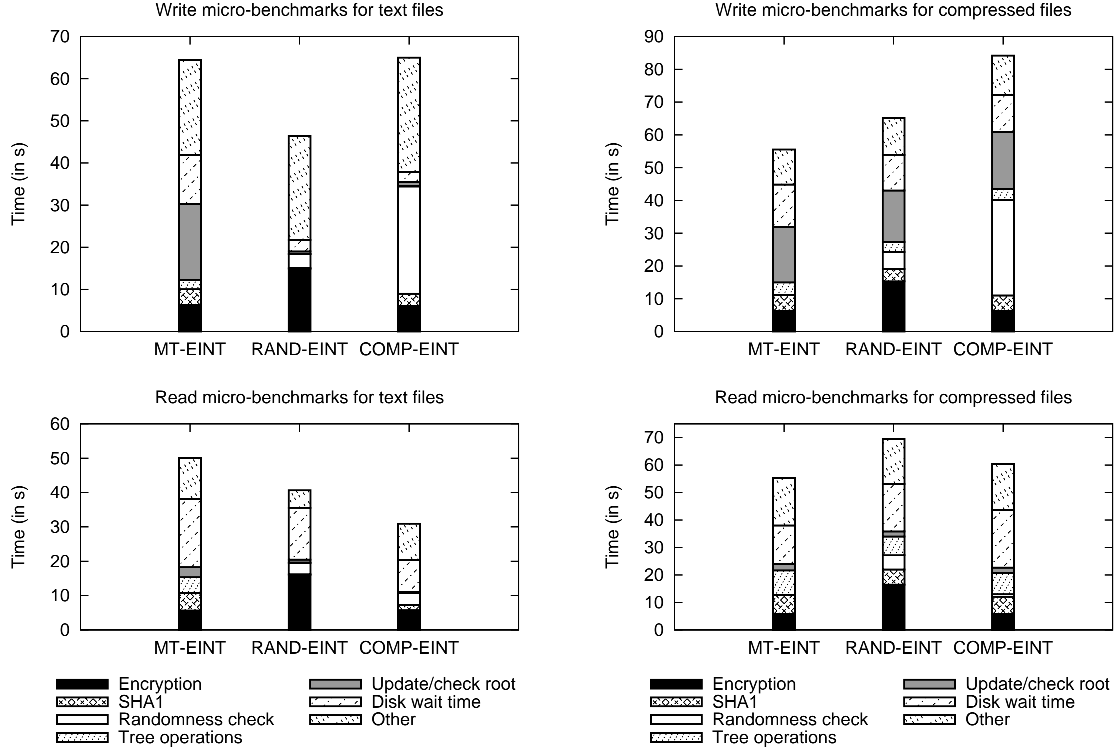

Figure 3.11: Micro-benchmarks for text and compressed files.

Micro-benchmarks. We first present a micro-benchmark evaluation for the text and compressed file sets in Figure 3.11. We plot the total time to write and read the set of text and compressed files, respectively. The write time for a set of files includes the time to encrypt all the files in the set, create new files, write the encrypted contents in the new files and build the integrity information for each file block with algorithm Update. The read time for a set of files includes the time to retrieve the encrypted files from disk, decrypt each file from the set and check the integrity of each file block with algorithm Check. We separate the total time incurred by the write and read experiments into the following components: encryption/decryption time (either AES or CMC); SHA1 hashing time; randomness check time (either the entropy test for CMC-Entropy or compression/decompression time for AES-Compression); Merkle tree operations (e.g., given a leaf index, find its index

in inorder traversal or given an inorder index of a node in the tree, find the inorder index of its sibling and parent); the time to update and check the root of the tree (the root of the Merkle tree is stored as an extended attribute for the file) and disk waiting time.

The results show that the cost of CMC encryption and decryption is about 2.5 times higher than that of AES encryption and decryption in CBC mode. Decompression is between 4 and 6 times faster than compression and this accounts for the good read performance of AES-Compression.

A substantial amount of the AES-Merkle overhead is due to disk waiting time (for instance, 39% at read for text files) and the time to update and check the root of the Merkle tree (for instance, 30% at write for compressed files). In contrast, due to smaller sizes of the Merkle trees in the CMC-Entropy and AES-Compression file systems, the disk waiting time and the time to update and check the root of the tree for text files are smaller. The results suggests that caching of the hash values stored in Merkle trees in the file system might reduce the disk waiting time and the time to update the root of the tree and improve the performance of all three integrity constructions, and specifically that of the AES-Merkle algorithm. We present our results on caching next.

Caching Merkle trees. We implemented a global cache that stores the latest hashes read from Merkle trees used to either update or check the integrity of file blocks. As an optimization, when we verify the integrity of a file block, we compute all the hashes on the path from the node up to the root of the tree until we reach a node that is already in the cache and whose integrity has been validated. We store in the cache only nodes that have been verified and that are authentic. When a node in the cache is written, all its ancestors on the path from the node to the root, including the node itself are evicted from the cache.

We plot the total file write and read time in seconds for the three cryptographic file systems as a function of different cache sizes. We also plot the average integrity bandwidth per block in a log-log scale. Finally, we plot the cumulative size of the untrusted storage US_F for all files from each set. We show the combined graphs for low-entropy files (text, object and executable files) in Figure 3.12 and for high-entropy files (compressed and image files) in Figure 3.13. The results represent averages over three independent runs of the same experiment.

The results show that AES-Merkle benefits mostly at read by implementing a cache of size 1KB, while the write time is not affected greatly by using a cache. The improvements for AES-Merkle using a cache of 1KB are as much as 25.22% for low-entropy files and 20.34% for high-entropy files in the read experiment. In the following, we compare the performance of the three constructions for the case in which a 1KB cache is used.

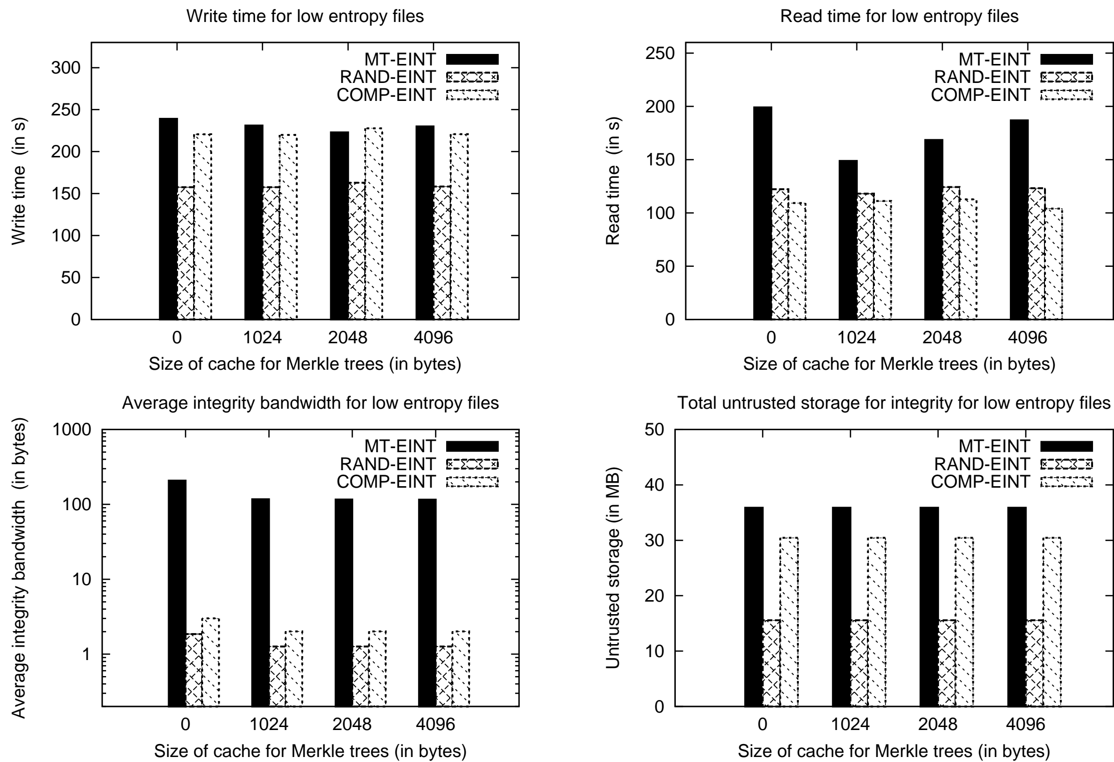

Figure 3.12: Evaluation for low-entropy files (text, object and executable files).

Results for low-entropy files. For sets of files with a low percent of random-looking blocks (text, object and executable files), CMC-Entropy outperforms AES-Merkle with respect to all the metrics considered. The performance of CMC-Entropy compared to that of AES-Merkle is improved by 31.77% for writes and 20.63% for reads. The performance of the AES-Compression file system is very different in the write and read experiments due to the cost difference of compression and decompression. The write time of AES-Compression is within 4% of the write time of AES-Merkle and in the read experiment AES-Compression outperforms AES-Merkle by 25.27%. The integrity bandwidth of CMC-Entropy and AES-Compression is 92.93 and 58.25 times, respectively, lower than that of AES-Merkle. The untrusted storage for integrity for CMC-Entropy and AES-Compression is reduced 2.3 and 1.17 times, respectively, compared to AES-Merkle.

Results for high-entropy files. For sets of files with a high percent of random-looking blocks (image and compressed files), CMC-Entropy adds a maximum performance overhead of 4.43% for writes and 18.15% for reads compared to AES-Merkle for a 1KB

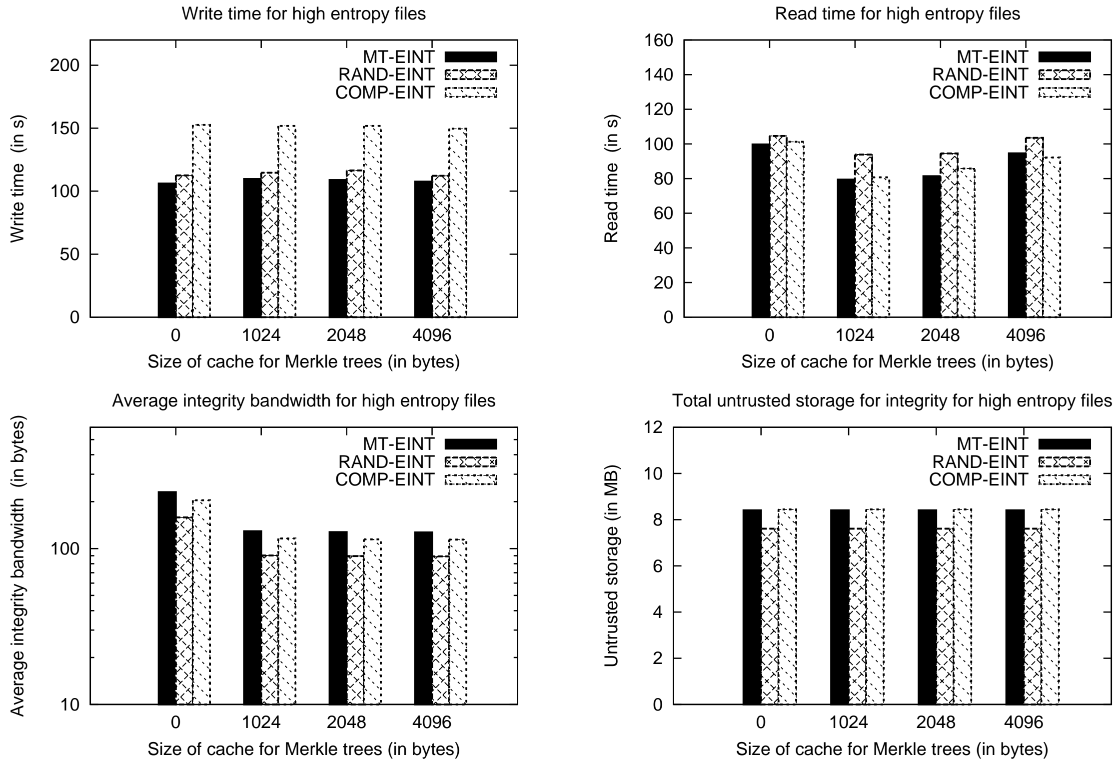

Figure 3.13: Evaluation for high-entropy files (image and compressed files).

cache. AES-Compression adds a write performance overhead of 38.39% compared to AES-Merkle, and performs within 1% of AES-Merkle in the read experiment. The average integrity bandwidth needed by CMC-Entropy and AES-Compression is lower by 30.15% and 10.22%, respectively, than that used by AES-Merkle. The untrusted storage for integrity used by CMC-Entropy is improved by 9.52% compared to AES-Merkle and that of AES-Compression is within 1% of the storage used by AES-Merkle. The reason that the average integrity bandwidth and untrusted storage for integrity are still reduced in CMC-Entropy compared to AES-Merkle is that in the set of high-entropy files considered only about 70% of the blocks have high entropy. We would expect that for files with 100% high-entropy blocks, these two metrics will exhibit a small overhead in both CMC-Entropy and AES-Compression compared to AES-Merkle (this is actually confirmed in the experiments from the next section). However, such workloads with 100% high entropy files are very unlikely to occur in practice.

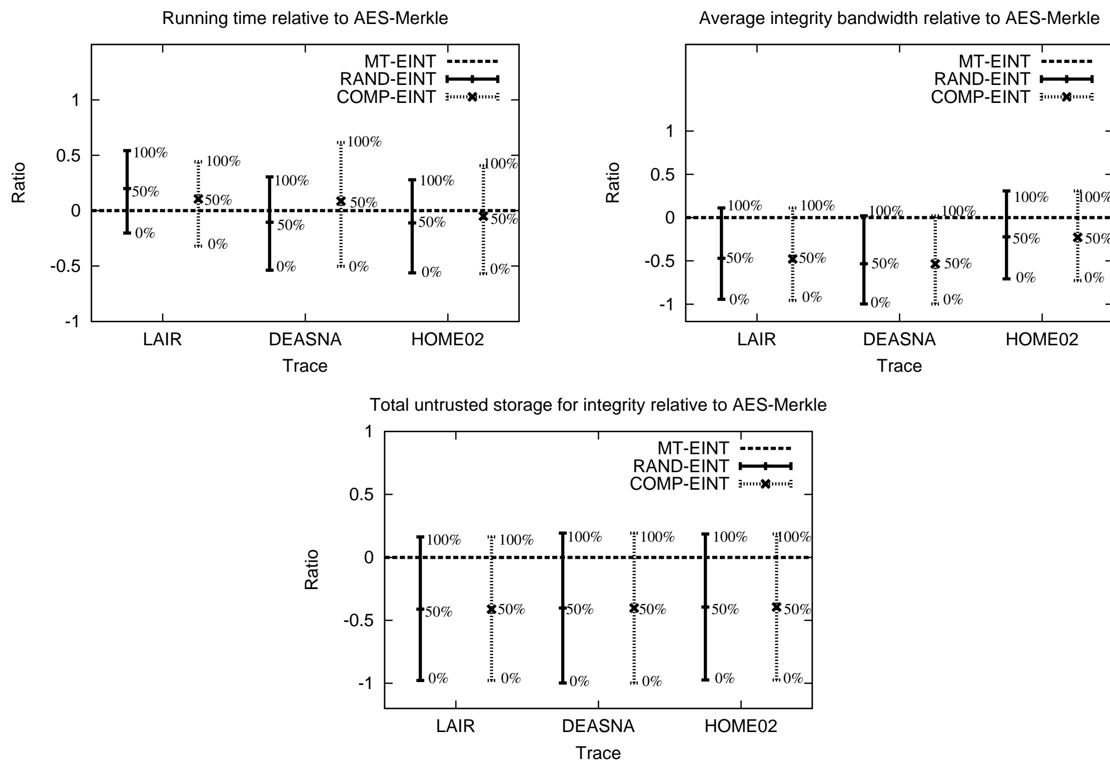

Figure 3.14: Running time, average integrity bandwidth and storage for integrity of CMC-Entropy and AES-Compression relative to AES-Merkle. Labels on the graphs represent percentage of random-looking blocks.

3.6.2 The Impact of File Access Patterns on Integrity Performance

File traces. We considered a subset of the three NFS Harvard traces(Ellard et al. [2003]) (LAIR, DEASNA and HOME02), each collected during one day. We show several characteristics of each trace, including the number of files and the total number of block write and read operations, in Table 3.3. The block size in these traces is 4096 bytes and we have implemented a 1KB cache for Merkle trees.

	Number of files	Number of writes	Number of reads
LAIR	7017	66331	23281
DEASNA	890	64091	521
HOME02	183	89425	11815

Table 3.3: NFS Harvard trace characteristics.

Experiments. We replayed each of the three traces with three types of synthetically-generated block contents: all low-entropy, all high-entropy and 50% high-entropy. For each experiment, we measured the total running time, the average integrity bandwidth and the total untrusted storage for integrity for CMC-Entropy and AES-Compression relative to AES-Merkle and plot the results in Figure 3.14. We represent the performance of AES-Merkle as the horizontal axis in these graphs and the performance of CMC-Entropy and AES-Compression relative to AES-Merkle. The points above the horizontal axis are overheads compared to AES-Merkle, and the points below the horizontal axis represent improvements relative to AES-Merkle. The labels on the graphs denote the percent of random-looking blocks synthetically generated.

Results. The performance improvements of CMC-Entropy and AES-Compression compared to AES-Merkle are as high as 56.21% and 56.85%, respectively, for the HOME02 trace for low-entropy blocks. On the other hand, the performance overhead for high-entropy blocks are at most 54.14% for CMC-Entropy (in the LAIR trace) and 61.48% for AES-Compression (in the DEASNA trace). CMC-Entropy performs better than AES-Compression when the ratio of read to write operations is small, as is the case for the DEASNA and HOME02 trace. As this ratio increases, AES-Compression outperforms CMC-Entropy.

For low-entropy files, both the average integrity bandwidth and the untrusted storage for integrity for both CMC-Entropy and AES-Compression are greatly reduced compared to AES-Merkle. For instance, in the DEASNA trace, AES-Merkle needs 215 bytes on average to update or check the integrity of a block, whereas CMC-Entropy and AES-Compression only require on average 0.4 bytes. The amount of additional untrusted storage for integrity in the DEASNA trace is 2.56 MB for AES-Merkle and only 7 KB for CMC-Entropy and AES-Compression. The maximum overhead added by both CMC-Entropy and AES-Compression compared to AES-Merkle for high-entropy blocks is 30.76% for the average integrity bandwidth (in the HOME02 trace) and 19.14% for the amount of untrusted storage for integrity (in the DEASNA trace).

3.6.3 File Content Statistics

We perform a user study to collect some statistics on file contents on several machines. We use eight machines from our department running Linux and we measure on each machine the percent of random-looking blocks (for the 8-bit entropy test) and the percent of blocks that cannot be compressed from users' home directories. We use a block size of 4KB. The results show that the the amount of random-looking blocks is between 8% and 36% of the

total number of blocks, with an average of 28% for all the machines. On the other hand, the amount of blocks that cannot be compressed is between 9% and 40%, with an average of 32% on the eight machines.

In Table 3.4 we show for each machine the percent of random-looking blocks and blocks that cannot be compressed enough to fit a MAC inside the block, as well as the percent of blocks on each machine that belong to the five file classes we have defined in Section 3.6.1.

	Total number of blocks	Random blocks	Blocks that can not be compressed	Text	Objects	Executables	Image	Compressed
Machine 1	4840810	10.53%	22.03%	9.66%	4.3%	3.62%	2.14%	52.96%
Machine 2	5203769	21.71%	23.82%	36.19%	7.56%	4.34%	.28%	11.05%
Machine 3	20489146	8.59%	9.04%	.48%	.13%	0	.05%	2.59%
Machine 4	1132932	21.61%	25.35%	6.69%	5.61%	6.77%	3.86%	4.12%
Machine 5	758939	21.6%	23.01%	23.48%	3.74%	5.35%	2.68%	3.41%
Machine 6	12314808	24.01%	26.13%	9.26%	5.8%	10.33%	.08%	10.87%
Machine 7	7354332	32.57%	38.45%	4.11%	.05%	3.14%	.3%	.21%
Machine 8	77706734	36.13%	40.32%	21.98%	0	0	0	0

Table 3.4: Statistics on file contents.

3.6.4 Amount of Trusted Storage

Finally, we perform some experiments to evaluate how the storage of the counter intervals, in particular the arrays $\mathsf{IntStart}_F$ and CtrVal_F, is affected by the amount of trusted storage per file. For that, we plot the number of files for which we need to keep the arrays $\mathsf{IntStart}_F$ and CtrVal_F in the untrusted storage space, as a function of the amount of trusted storage per file. The results for the three traces are in Figure 3.15. We conclude that a value of 200 bytes of constant storage per file (which we have used in our experiments) is enough to keep the counter intervals for all the files in the LAIR and DEASNA traces, and about 88% percent of the files in the HOME02 trace.

3.7 Discussion

From the evaluation of the three constructions, it follows that none of the schemes is a clear winner over the others with respect to all the three metrics considered. While the performance of AES-Merkle is not dependent on workloads, the performance of both CMC-Entropy and AES-Compression is greatly affected by file block contents and file

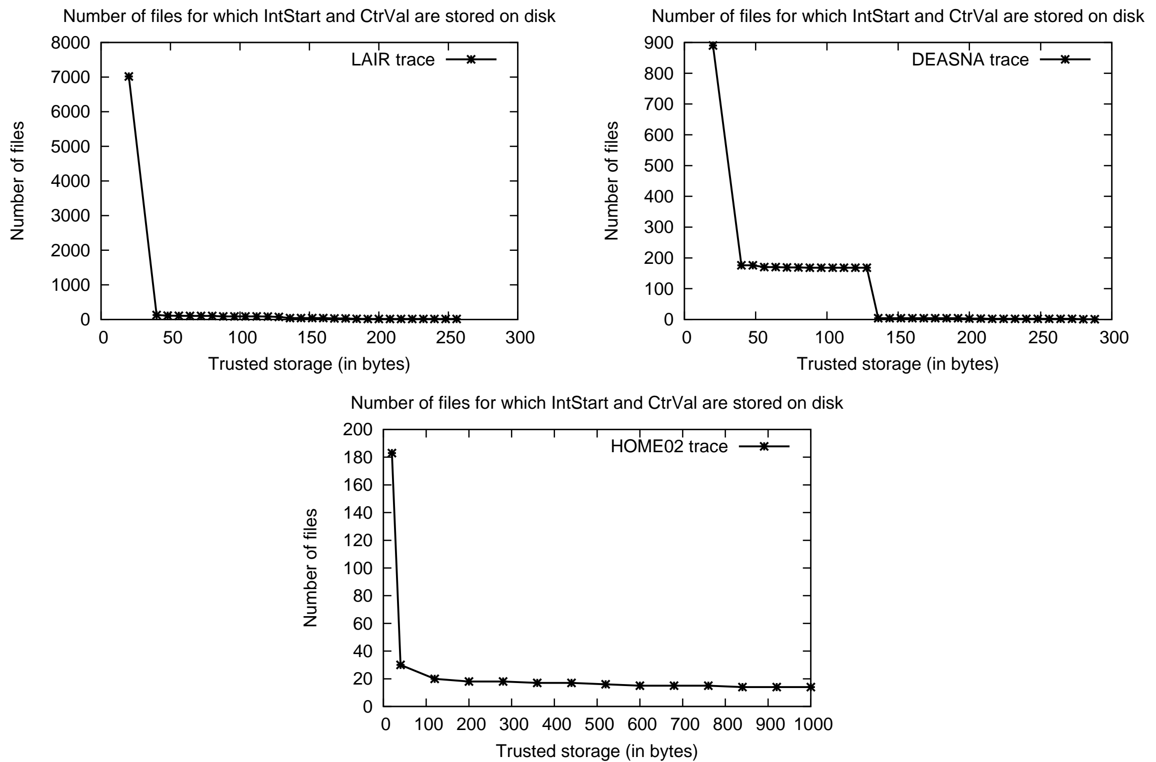

Figure 3.15: Number of files for which the counter intervals are stored in the untrusted storage space.

access patterns. We recommend that all three constructions be implemented in a cryptographic file system. An application can choose the best scheme based on its typical workload. For an application in which the large majority of files have high-entropy (e.g., a file sharing application in which users transfer mostly audio and video files), the standard AES-Merkle still remains the best option for integrity. On the other hand, for applications in which typical files have low entropy, CMC-Entropy is the best choice for most ratios of read to write operations, and AES-Compression is the best option for primarily read-only workloads when minimizing read latency is a priority.

The new algorithms that we propose can be applied in other settings in which authentication of data stored on untrusted storage is desired. One example is checking the integrity of arbitrarily-large memory in a secure processor using only a constant amount of trusted storage (Blum et al. [1994], Clarke et al. [2005]). In this setting, a trusted checker maintains a constant amount of trusted storage and, possibly, a cache of data blocks most recently read from the main memory. The goal is for the checker to verify the integrity of

the untrusted memory using a small bandwidth overhead.

The algorithms described in this chapter can be only used in applications where the data that needs to be authenticated is encrypted. However, the COMP-FINT integrity algorithm can be easily modified to fit into a setting in which data is only authenticated and not encrypted, and can thus replace Merkle trees in such applications. On the other hand, the RAND-FINT integrity algorithm is only suitable in a setting in which data is encrypted with a tweakable cipher, as the integrity guarantees of this algorithm are based on the security properties of such ciphers.

3.8 Related Work

Integrity in storage systems. Integrity in storage systems can be provided at different levels, such as the device driver level, the file system or virtual file system layers. Common cryptographic primitives used for integrity are collision-resistant hash functions, message-authentication codes and digital signatures.

Cryptographic file systems. Cryptographic file systems extend traditional file systems with support for security in untrusted storage environments. Most cryptographic file systems provide to their clients mechanisms for encrypting data and checking data integrity. The cryptographic operations are performed on the client side, as both the storage devices and the network are untrusted.

A common integrity method used in systems such as TCFS (Cattaneo et al. [2001]) and SNAD (Miller et al. [2002]) is to store a hash or message-authentication code for each file block for authenticity, but this results in a lot of additional storage for integrity. In an environment with all the storage servers untrusted, the use of public-key cryptography is necessary for integrity. In systems such as SFS (Mazieres et al. [1999]), SFSRO (Fu et al. [2002]), Cepheus (Fu [1999]), FARSITE (Adya et al. [2002]), Plutus (Kallahalla et al. [2003]), and SUNDR (Li et al. [2004]), a Merkle tree per file is built and the root of the tree is digitally signed for authenticity. In SiRiUS (Goh et al. [2003]) each file is digitally signed for authenticity, but in this approach the integrity bandwidth to update or check a block in a file is linear in the file size.

Pletka and Cachin (Pletka and Cachin [2006]) describes a security architecture for the distributed IBM StorageTank file system (also denoted the SAN.FS file system) (Menon et al. [2003]). SAN.FS achieves its scalability to many clients by separating the meta-data operations from the data path. As such, the meta-data is stored on trusted meta-data

servers, while the data is stored on the storage devices from a SAN. SAN.FS uses AES in CBC mode for confidentiality and Merkle trees (with degree 16) for integrity protection of files. The roots of the Merkle trees are stored together with the file system meta-data on the meta-data servers. Thus, the security architecture for the SAN.FS file system implements the MT-FINT scheme, with the difference that Merkle trees of degree 16 are used instead of binary Merkle trees.

Stackable file systems (Heidemann and Popek [1994]) are a means of adding new functionality to existing file systems in a portable way. They intercept calls from the virtual file system (VFS), perform some special operations (e.g., encryption, integrity checking) and finally, redirect them to lower level file systems. NCryptFS (Wright et al. [2003b]) and CryptFS (Zadok et al. [1998]) are stackable file systems that only guarantee confidentiality. They have been extended by I^3FS (Sivathanu et al. [2004]) and eCryptFS (Halcrow [2005]), respectively, to provide integrity by computing and storing page level hashes.

For journaling file systems, an elegant solution for integrity called *hash logging* is provided by PFS (Stein et al. [2001]). The hashes of file blocks together with the file system metadata are stored in the file system log, a protected memory area. However, in this solution the amount of storage for integrity for a file is linear in the number of blocks in the file.

User-level integrity. Tripwire (Kim and Spafford [1994]) is a user-level tool that computes a hash per file and stores it in trusted memory. While this approach achieves constant trusted storage for integrity per file, the integrity bandwidth is linear in the number of blocks in the file. Other tools for remote monitoring of file integrity are Samhain (Samhain) and Osiris (Osiris).

Integrity of data on the network. There exists several storage systems in which data is stored in clear on the storage disks. The confidentiality and integrity of data are protected only on the untrusted network, using network transport protocols such as IPSEC. In the NASD (Gibson et al. [1998], Gobioff et al. [1997]) object storage architecture, intelligent storage disks are directly connected to the network and security is integrated at the device driver level. To lower the bandwidth offered by the storage disks, NASD uses a new integrity scheme based on message-authentication codes in which the integrity information is precomputed and data integrity can be verified incrementally (Gobioff et al. [1998]). Other object-based storage systems in which security is provided at the device driver level are the ObjectStore (Azagury et al. [2003, 2002]) prototype from IBM Haifa and Snapdragon (Aguilera et al. [2003]) from HP.

Surveys. Riedel et al. (Riedel et al. [2002]) provides a framework for extensively evaluating the security of storage systems. Wright et al. (Wright et al. [2003a]) evaluates the performance of five cryptographic file systems, focusing on the overhead of encryption. Two other recent surveys about securing storage systems are by Sivathanu et al. (Sivathanu et al. [2004]) and Kher and Kimand (Kher and Kim [2005]).

Memory integrity checking. Another area related to our work is that of checking the integrity of arbitrarily large untrusted memory using only a small, constant-size trusted memory. A first solution to this problem (Blum et al. [1994]) is to check the integrity of each memory operation using a Merkle tree. This results in a logarithmic bandwidth overhead in the total number of memory blocks. Recent solutions try to reduce the logarithmic bandwidth to almost a constant value. The main idea by Clarke et al. (Clarke et al. [2005]) is to verify the integrity of the memory only when a *critical operation* is encountered. The integrity of sequences of operations between critical operations is checked by aggregating these operations in a log using incremental hash functions (Clarke et al. [2003]). In contrast, in our model we need to be able to check the integrity of *each* file block read from the storage servers.

Chapter 4

Lazy Revocation in Cryptographic File Systems

In this chapter, we consider the problem of efficient key management and user revocation in cryptographic file systems that allow shared access to files. We consider a model in which access control is done at the level of individual files. Each file is divided into fixed-size file blocks encrypted individually with the encryption key for the file. A performance-efficient solution to user revocation in such systems is lazy revocation, a method that delays the re-encryption of a file block until the next write to that block. In contrast, in active revocation, all blocks in a file are immediately re-encrypted at the moment of revocation, but the amount of work caused by a single revocation might be prohibitive for large files. In systems using lazy revocation, all blocks in a file are initially encrypted with the same file key, but after several user revocations different blocks might be written with different versions of the file key. The encryption keys for a file are generated, stored and distributed by a trusted entity called *center* (e.g., the owner of the file). The focus of this chapter is designing efficient, provable secure key management schemes for lazy revocation that require a minimum amount of trusted storage. This chapter is based on joint work with Michael Backes and Christian Cachin from IBM Zurich Research Lab (Backes et al. [2005a, 2006, 2005b]).

After introducing some preliminary material in Section 4.1, we formalize in Section 4.2 the notion of key-updating schemes for lazy revocation, an abstraction to manage cryptographic keys in file systems with lazy revocation, and give a security definition for such schemes. We give two composition methods that combine two secure key-updating schemes into a new secure scheme that permits a larger number of user revocations in Section 4.3. In Section 4.4, we prove the security of two slightly modified existing con-

structions and propose a novel binary tree construction that is also provably secure in our model. We analyze the computational and communication complexity of the three constructions in Section 4.5 and describe our performance evaluation of the constructions in Section 4.6. We provide in Section 4.7 a comprehensive formalization of the cryptographic primitives used in a file system with lazy revocation, in particular symmetric encryption schemes, message authentication codes and signature schemes. Finally, we describe the related literature in Section 4.8.

4.1 Preliminaries

4.1.1 Pseudorandom Generators

A pseudorandom generator is a function $G : \{0,1\}^s \to \{0,1\}^{b+s}$ that takes as input a s-bit seed and outputs a string of length $b+s$ bits that is computationally indistinguishable from a random string of the same length. A formal definition is below.

Definition 8 (Pseudorandom Generators) *Let G be a function as above and $\mathcal{D}$ a distinguishing algorithm that participates in the experiments from Figure 4.1. We define the* `prg`*-advantage of $\mathcal{D}$ for G as:*

$$\mathsf{Adv}^{\mathtt{prg}}_{G,\mathcal{D}} = |\Pr[\mathsf{Exp}^{\mathtt{prg}\text{-}1}_{G,\mathcal{D}} = 1] - \Pr[\mathsf{Exp}^{\mathtt{prg}\text{-}0}_{G,\mathcal{D}} = 1]|.$$

We define $\mathsf{Adv}^{\mathtt{prg}}_{G}(\tau)$ to be the maximum advantage $\mathsf{Adv}^{\mathtt{prg}}_{G,\mathcal{D}}$ over all distinguishing algorithms $\mathcal{D}$ running in time at most τ.

$\mathsf{Exp}^{\mathtt{prg}\text{-}0}_{G,\mathcal{D}}$	$\mathsf{Exp}^{\mathtt{prg}\text{-}1}_{G,\mathcal{D}}$
$u \xleftarrow{R} \{0,1\}^s$	$r \xleftarrow{R} \{0,1\}^{b+s}$
$r \leftarrow G(u)$	$b \leftarrow \mathcal{D}(r)$
$b \leftarrow \mathcal{D}(r)$	return b
return b	

Figure 4.1: Experiments defining the security of pseudorandom generator G.

4.1.2 Trapdoor Permutations

A trapdoor permutation family is a family of functions with the property that it is computationally hard to invert a function drawn at random from the family at a random point in the

domain, unless a *trapdoor* (i.e., some secret information) is known. A formal definition is below.

Definition 9 (Trapdoor Permutation) *Consider a family $\mathcal{F}$ of permutations $f : \{0,1\}^s \rightarrow \{0,1\}^s$. $\mathcal{F}$ is a trapdoor permutation family if the following two properties are satisfied:*

One-wayness: *For any adversary algorithm $\mathcal{A}$, we define its one-way advantage as:*

$$\mathsf{Adv}^{\mathsf{one\text{-}way}}_{\mathcal{F},\mathcal{A}} = \Pr[f \overset{R}{\leftarrow} \mathcal{F}, y \overset{R}{\leftarrow} \{0,1\}^s, x \leftarrow \mathcal{A}(f,y) : f(x) = y].$$

We define $\mathsf{Adv}^{\mathsf{one\text{-}way}}_{\mathcal{F}}(\tau)$ to be the maximum advantage $\mathsf{Adv}^{\mathsf{one\text{-}way}}_{\mathcal{F},\mathcal{A}}$ over all adversary algorithms $\mathcal{A}$ running in time at most τ.

Trapdoor: *Given the trapdoor information trap_f for a function $f \in \mathcal{F}$, there exists an efficient algorithm $\mathcal{I}$ such that $\mathcal{I}(y, \mathsf{trap}_f) = f^{-1}(y)$, for all $y \in \{0,1\}^s$.*

4.1.3 CPA-Security of Symmetric Encryption

We recall that a symmetric encryption scheme $\mathcal{E}$ consists of three algorithms: a key generation algorithm $\mathsf{Gen}(1^\kappa)$ that takes a security parameter as input and outputs a key, an encryption algorithm $\mathsf{E}_k(M)$ that outputs the encryption of a given message M with key k, and a decryption algorithm $\mathsf{D}_k(C)$ that decrypts a ciphertext C with key k. The first two algorithms might be probabilistic, but D is deterministic. The correctness property requires that $\mathsf{D}_k(\mathsf{E}_k(M)) = M$, for all keys k generated by Gen and all messages M from the encryption domain.

CPA-security of a symmetric encryption scheme $\mathcal{E} = (\mathsf{Gen}, \mathsf{E}, \mathsf{D})$ requires that any adversary $\mathcal{A}$ with access to an encryption oracle $\mathsf{E}_k(\cdot)$ is unable to distinguish between encryption of two messages M_0 and M_1 of its choice.

Definition 10 (CPA-Security of Symmetric Encryption) *Let $\mathcal{E}$ be a symmetric encryption scheme and $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ an adversary algorithm that participates in the experiments from Figure 4.2. We define the* cpa*-advantage of $\mathcal{A}$ for $\mathcal{E}$ as:*

$$\mathsf{Adv}^{\mathsf{cpa}}_{\mathcal{E},\mathcal{A}} = |\Pr[\mathsf{Exp}^{\mathsf{cpa\text{-}1}}_{\mathcal{E},\mathcal{A}} = 1] - \Pr[\mathsf{Exp}^{\mathsf{cpa\text{-}0}}_{\mathcal{E},\mathcal{A}} = 1]|.$$

We define $\mathsf{Adv}^{\mathsf{cpa}}_{\mathcal{E},\mathcal{A}}(\tau, q)$ to be the maximum advantage $\mathsf{Adv}^{\mathsf{cpa}}_{\mathcal{E},\mathcal{A}}$ over all adversary algorithms $\mathcal{A}$ running in time at most τ and making q queries to the encryption oracle.

W.l.o.g., we can relate the success probability of $\mathcal{A}$ and its advantage as

$$\Pr\big[\mathcal{A} \text{ succeeds}\big] = \frac{1}{2}\big[1 + \mathsf{Adv}^{\mathsf{cpa}}_{\mathcal{E},\mathcal{A}}\big]. \tag{4.1}$$

$$
\begin{array}{l}
\mathsf{Exp}^{\mathsf{cpa}\text{-}b}_{\mathcal{E},\mathcal{A}} \\
\quad k \stackrel{R}{\leftarrow} \mathsf{Gen}(1^{\kappa}) \\
\quad (M_0, M_1) \leftarrow \mathcal{A}_1^{\mathsf{E}_k(\cdot)}() \\
\quad b \stackrel{R}{\leftarrow} \{0,1\} \\
\quad b' \leftarrow \mathcal{A}_2^{\mathsf{E}_k(\cdot)}(\mathsf{E}_k(M_b)) \\
\quad \mathsf{if}\ b = b' \\
\quad\quad \mathsf{return}\ 1 \\
\quad \mathsf{else} \\
\quad\quad \mathsf{return}\ 0
\end{array}
$$

Figure 4.2: Experiments defining the CPA-security of encryption.

4.1.4 Signature Schemes and Identity-Based Signatures

Signature schemes. A signature scheme consists of three algorithms: a key generation algorithm $\mathsf{Gen}(1^{\lambda})$ that outputs a public key/secret key pair $(\mathsf{PK}, \mathsf{SK})$ taking as input a security parameter, a 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 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 first two algorithms might be probabilistic, but Ver is deterministic. 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.

CMA-security for a signature scheme (Goldwasser et al. [1988]) requires that an adversary with access to a signing oracle $\mathsf{Sign}_{\mathsf{SK}}(\cdot)$ is not able to generate a message and a valid signature for which it did not query the signing oracle.

Definition 11 (CMA-Security of Signature Schemes) *Let $\mathcal{S}$ be a signature scheme and $\mathcal{A}$ an adversary algorithm. We define the* cma*-advantage of $\mathcal{A}$ for $\mathcal{S}$ as:*

$$
\begin{aligned}
\mathsf{Adv}^{\mathsf{cma}}_{\mathcal{S},\mathcal{A}} \;=\; & \Pr[(\mathsf{SK}, \mathsf{PK}) \leftarrow \mathsf{Gen}(1^{\lambda}), (M, \sigma) \leftarrow \mathcal{A}^{\mathsf{Sign}_{\mathsf{SK}}(\cdot)}(\mathsf{PK}) : \mathsf{Ver}_{\mathsf{PK}}(M, \sigma) = 1 \text{ AND} \\
& M \textit{ was not a query to } \mathsf{Sign}_{\mathsf{SK}}(\cdot)].
\end{aligned}
$$

We define $\mathsf{Adv}^{\mathsf{cma}}_{\mathcal{S},\mathcal{A}}(\tau, q)$ to be the maximum advantage $\mathsf{Adv}^{\mathsf{cma}}_{\mathcal{S},\mathcal{A}}$ over all adversary algorithms $\mathcal{A}$ running in time at most τ and making q queries to the signing oracle.

Identity-based signatures. Identity-based signatures have been introduced by Shamir (Shamir [1985]). A trusted entity initially generates a *master secret key* and a *master public key*. Later the trusted entity can generate the signing key for a user from the master secret key and the user's identity, which is an arbitrary bit string. In order to verify

a signature, it is enough to know the master public key and the signer's identity, which is a public string.

Definition 12 (Identity-Based Signatures) *An identity-based signature scheme consists of a tuple of four probabilistic polynomial-time algorithms* $\mathcal{S} = (\mathsf{MKGen}, \mathsf{UKGen}, \mathsf{Sign}, \mathsf{Ver})$ *with the following properties:*

- *The master key generation algorithm,* MKGen*, takes as input the* security parameter 1^λ*, and outputs the* master public key MPK *and* master secret key MSK *of the scheme.*
- *The user key generation algorithm,* UKGen*, takes as input the* master secret key MSK *and the* user's identity ID*, and outputs the* secret key $\mathsf{SK_{ID}}$ *for the user.*
- *The signing algorithm,* Sign*, takes as input the* user's secret key $\mathsf{SK_{ID}}$ *and a* message M*, and outputs a* signature σ.
- *The verification algorithm,* Ver*, takes as input the* master public key MPK*, the signer's identity* ID*, a* message M *and a* signature σ *and outputs a bit. The signature* σ *generated by the user with identity* ID *is said to be* valid *on message* M *if* $\mathsf{Ver}(\mathsf{MPK}, \mathsf{ID}, M, \sigma) = 1$.

Correctness of IBS. The correctness property requires that, if $(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{MKGen}(1^\kappa)$ is a pair of master public and secret keys for the scheme, $\mathsf{SK_{ID}} \leftarrow \mathsf{UKGen}(\mathsf{MSK}, \mathsf{ID})$ is the signing key for the user with identity ID, then $\mathsf{Ver}(\mathsf{MPK}, \mathsf{ID}, M, \mathsf{Sign}(\mathsf{SK_{ID}}, M)) = 1$, for all messages M and all identities ID.

Security of IBS. Consider an adversary $\mathcal{A}$ that participates in the following experiment:

Initialization: The master public key MPK and master secret key MSK are generated with MKGen. MPK is given to $\mathcal{A}$.

Oracle queries: The adversary has access to three oracles: $\mathsf{InitID}(\cdot)$ that allows it to generate the secret key for a new identity, $\mathsf{Corrupt}(\cdot)$ that gives the adversary the secret key for an identity of its choice, and $\mathsf{Sign}(\cdot, \cdot)$ that generates the signature on a particular message and identity.

Output: The adversary outputs the identity of an uncorrupted user, a message and a signature.

The adversary succeeds if the signature it outputs is valid and the adversary did not query the message to the signing oracle. We denote by $\mathsf{Adv}^{\mathsf{ibs}}_{\mathcal{S},\mathcal{A}}$ the probability of success of $\mathcal{A}$ and by $\mathsf{Adv}^{\mathsf{ibs}}_{\mathcal{S}}(\tau, q_1, q_2, q_3)$ the maximum advantage $\mathsf{Adv}^{\mathsf{ibs}}_{\mathcal{S},\mathcal{A}}$ over all adversary algorithms $\mathcal{A}$ running in time at most τ and making q_1 queries to the $\mathsf{InitID}(\cdot)$ oracle, q_2 queries to the $\mathsf{Corrupt}(\cdot)$ oracle and q_3 queries to the $\mathsf{Sign}(\cdot,\cdot)$ oracle.

4.2 Formalizing Key-Updating Schemes

4.2.1 Definition of Key-Updating Schemes

In our model, we divide time into intervals, not necessarily of fixed length, and each time interval is associated with a new key that can be used in a symmetric-key cryptographic algorithm. In a key-updating scheme, the trusted center generates initial state information that is updated at each time interval, and from which the center can derive a user key. The user key for interval t permits a user to derive the keys of previous time intervals (k_i for $i \leq t$), but it should not give any information about keys of future time intervals (k_i for $i > t$).

For simplicity, we assume that all the keys are bit strings of length a security parameter denoted κ. The number of time intervals and the security parameter are given as input to the initialization algorithm. We define formally key-updating schemes below.

Definition 13 (Key-Updating Schemes) *A key-updating scheme consists of four deterministic polynomial time algorithms* $\mathsf{KU} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Extract})$ *with the following properties:*

- *The initialization algorithm,* Init*, takes as input the* security parameter 1^κ*, the* number of time intervals T *and a* random seed $s \in \{0,1\}^{l(\kappa)}$ *for a polynomial* $l(\kappa)$*, and outputs a bit string* CS_0*, called the initial center* state.
- *The key update algorithm,* Update*, takes as input the current* time interval $0 \leq t \leq T-1$*, the current center* state CS_t*, and outputs the center* state CS_{t+1} *for the next time interval.*
- *The user key derivation algorithm,* Derive*, is given as input a* time interval $1 \leq t \leq T$ *and the center* state CS_t*, and outputs the* user key UK_t*. The user key can be used to derive all keys* k_i *for* $1 \leq i \leq t$*.*

- *The key extraction algorithm,* $\mathsf{Extract}$*, is executed by the user and takes as input a* time interval $1 \leq t \leq T$*, the* user key UK_t *for interval* t *as received from the center, and a* target time interval i *with* $1 \leq i \leq t$*. The algorithm outputs the* key k_i *for interval* i*.*

W.l.o.g., we assume that the Update algorithm is run at least once after the Init algorithm, before any user keys can be derived. The first time the Update algorithm is run, it is given as input time interval $t = 0$. User keys and keys are associated with the time intervals between 1 and T.

4.2.2 Security of Key-Updating Schemes

The definition of security for key-updating schemes requires, informally, that a polynomial-time adversary with access to the user key for a time interval t is able to distinguish the key of the next time interval from a randomly generated key only with very small probability. The definition we give here is related to the definition of forward-secure pseudorandom generators given by Bellare and Yee (Bellare and Yee [2003]). Formally, consider an adversary $\mathcal{A} = (\mathcal{A}_{\mathcal{U}}, \mathcal{A}_{\mathcal{G}})$ that participates in the following experiment:

Initialization: The initial center state is generated with the Init algorithm.

Key updating: The adversary adaptively picks a time interval t such that $0 \leq t \leq T - 1$ as follows. Starting with $t = 0, 1, \ldots$, algorithm $\mathcal{A}_{\mathcal{U}}$ is given the user keys UK_t for all consecutive time intervals until $\mathcal{A}_{\mathcal{U}}$ decides to output stop or t becomes equal to $T-1$. We require that $\mathcal{A}_{\mathcal{U}}$ outputs stop at least once before halting. $\mathcal{A}_{\mathcal{U}}$ also outputs some additional information $z \in \{0,1\}^*$ that is given as input to algorithm $\mathcal{A}_{\mathcal{G}}$.

Challenge: A challenge for the adversary is generated, which is either the key for time interval $t+1$ generated with the Update, Derive and $\mathsf{Extract}$ algorithms, or a random bit string of length κ.

Guess: $\mathcal{A}_{\mathcal{G}}$ takes the challenge and z as inputs and outputs a bit b.

The adversary succeeds if it distinguishes between the properly generated key for time interval $t + 1$ and a randomly generated key. More formally, the definition of a secure key-updating scheme is the following:

$\mathsf{Exp}^{\mathtt{sku}\text{-}0}_{\mathsf{KU},\mathcal{A}}(T)$:	$\mathsf{Exp}^{\mathtt{sku}\text{-}1}_{\mathsf{KU},\mathcal{A}}(T)$:
$\quad \mathsf{CS}_0 \leftarrow \mathsf{Init}(1^\kappa, T)$	$\quad \mathsf{CS}_0 \leftarrow \mathsf{Init}(1^\kappa, T)$
$\quad t \leftarrow 0$	$\quad t \leftarrow 0$
$\quad (d, z) \leftarrow \mathcal{A}_\mathcal{U}(t, \perp, \perp)$	$\quad (d, z) \leftarrow \mathcal{A}_\mathcal{U}(t, \perp, \perp)$
$\quad \mathsf{while}(d \neq \mathsf{stop})\ \mathsf{and}\ (t < T-1)$	$\quad \mathsf{while}(d \neq \mathsf{stop})\ \mathsf{and}\ (t < T-1)$
$\qquad t \leftarrow t+1$	$\qquad t \leftarrow t+1$
$\qquad \mathsf{CS}_t \leftarrow \mathsf{Update}(t-1, \mathsf{CS}_{t-1})$	$\qquad \mathsf{CS}_t \leftarrow \mathsf{Update}(t-1, \mathsf{CS}_{t-1})$
$\qquad \mathsf{UK}_t \leftarrow \mathsf{Derive}(t, \mathsf{CS}_t)$	$\qquad \mathsf{UK}_t \leftarrow \mathsf{Derive}(t, \mathsf{CS}_t)$
$\qquad (d, z) \leftarrow \mathcal{A}_\mathcal{U}(t, \mathsf{UK}_t, z)$	$\qquad (d, z) \leftarrow \mathcal{A}_\mathcal{U}(t, \mathsf{UK}_t, z)$
$\quad \mathsf{CS}_{t+1} \leftarrow \mathsf{Update}(t, \mathsf{CS}_t)$	$\quad k_{t+1} \stackrel{R}{\leftarrow} \{0,1\}^\kappa$
$\quad \mathsf{UK}_{t+1} \leftarrow \mathsf{Derive}(t+1, \mathsf{CS}_{t+1})$	$\quad b \leftarrow \mathcal{A}_\mathcal{G}(k_{t+1}, z)$
$\quad k_{t+1} \leftarrow \mathsf{Extract}(t+1, \mathsf{UK}_{t+1}, t+1)$	$\quad \mathsf{return}\ b$
$\quad b \leftarrow \mathcal{A}_\mathcal{G}(k_{t+1}, z)$	
$\quad \mathsf{return}\ b$	

Figure 4.3: Experiments defining the security of key-updating schemes.

Definition 14 (Security of Key-Updating Schemes) *Let* $\mathsf{KU} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Extract})$ *be a key-updating scheme and* $\mathcal{A}$ *an adversary algorithm that participates in one of the two experiments defined in Figure 4.3.*

The advantage of the adversary $\mathcal{A} = (\mathcal{A}_\mathcal{U}, \mathcal{A}_\mathcal{G})$ *for* KU *is defined as*

$$\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU},\mathcal{A}}(T) = \left|\Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}1}_{\mathsf{KU},\mathcal{A}}(T) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}0}_{\mathsf{KU},\mathcal{A}}(T) = 1\right]\right|.$$

We define $\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}}(T, \tau)$ *to be the maximum advantage* $\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU},\mathcal{A}}(T)$ *over all adversary algorithms* $\mathcal{A}$ *running in time at most* τ*.*

Without loss of generality, we can relate the success probability of adversary $\mathcal{A}$ of distinguishing between the two experiments and its advantage as

$$\begin{aligned} \Pr[\mathcal{A} \text{ succeeds}] &= \frac{1}{2}\left[\Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}0}_{\mathsf{KU},\mathcal{A}}(T) = 0\right] + \Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}1}_{\mathsf{KU},\mathcal{A}}(T) = 1\right]\right] \\ &= \frac{1}{2}\left[1 + \mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU},\mathcal{A}}(T)\right]. \end{aligned} \tag{4.2}$$

Remark. The security notion we have defined is equivalent to a seemingly stronger security definition, in which the adversary can choose the challenge time interval t^* with the restriction that t^* is greater than the time interval at which the adversary outputs stop and that t^* is polynomial in the security parameter. This second security definition guarantees, intuitively, that the adversary is not gaining any information about the keys of any future time intervals after it outputs stop.

4.3 Composition of Key-Updating Schemes

Let $\mathsf{KU}_1 = (\mathsf{Init}_1, \mathsf{Update}_1, \mathsf{Derive}_1, \mathsf{Extract}_1)$ and $\mathsf{KU}_2 = (\mathsf{Init}_2, \mathsf{Update}_2, \mathsf{Derive}_2, \mathsf{Extract}_2)$ be two secure key-updating schemes using the same security parameter κ with T_1 and T_2 time intervals, respectively. In this section, we show how to combine the two schemes into a secure key-updating scheme $\mathsf{KU} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Extract})$, which is either the additive or multiplicative composition of the two schemes with $T = T_1 + T_2$ and $T = T_1 \cdot T_2$ time intervals, respectively. Similar generic composition methods have been given previously for forward-secure signature schemes (Malkin et al. [2002]).

For simplicity, we assume the length of the random seed in the Init algorithm of the scheme KU to be κ for both composition methods. Let $G : \{0,1\}^\kappa \rightarrow \{0,1\}^{l_1(\kappa)+l_2(\kappa)}$ be a pseudorandom generator; it can be used to expand a random seed of length κ into two pseudorandom bit strings of length $l_1(\kappa)$ and $l_2(\kappa)$, respectively, as needed for Init_1 and Init_2. We write $G(s) = G_1(s) \| G_2(s)$ with $|G_1(s)| = l_1(\kappa)$ and $|G_2(s)| = l_2(\kappa)$ for $s \in \{0,1\}^\kappa$.

We make several notations for the running times of the algorithms of a key-updating scheme. We denote by $\mathsf{TimeUserKeys}(T, \mathsf{KU})$ the total running time to generate the users keys of key-updating scheme KU for intervals $1, 2, \ldots, T$. This includes the time to initialize the scheme, apply algorithm Update T times and algorithm Derive T times. We denote by $\mathsf{TimeKeys}(T, \mathsf{KU})$ the total running time to generate the keys of key-updating scheme KU for intervals $1, 2, \ldots, T$ obtained by T runs of algorithm $\mathsf{Extract}$. We denote by $\mathsf{TimeMaxExtract}(T, \mathsf{KU})$ the maximum running time of algorithm $\mathsf{Extract}$ of KU.

4.3.1 Additive Composition

The additive composition of two key-updating schemes uses the keys generated by the first scheme for the first T_1 time intervals and the keys generated by the second scheme for the subsequent T_2 time intervals. The user key for the first T_1 intervals in KU is the same as that of scheme KU_1 for the same interval. For an interval t greater than T_1, the user key includes both the user key for interval $t - T_1$ of scheme KU_2, and the user key for interval T_1 of scheme KU_1. The details of the additive composition method are described in Figure 4.4.

The security of the composition operation is analyzed in the following theorem.

Theorem 6 *Suppose that* $\mathsf{KU}_1 = (\mathsf{Init}_1, \mathsf{Update}_1, \mathsf{Derive}_1, \mathsf{Extract}_1)$ *and* $\mathsf{KU}_2 = (\mathsf{Init}_2, \mathsf{Update}_2, \mathsf{Derive}_2, \mathsf{Extract}_2)$ *are two secure key-updating schemes with* T_1 *and* T_2 *time*

$\mathsf{Init}(1^\kappa, T, s)$:	$\mathsf{Derive}(t, (\mathsf{CS}_t^1, \mathsf{CS}_t^2))$:
$\mathsf{CS}_0^1 \leftarrow \mathsf{Init}_1(1^\kappa, T_1, G_1(s))$ $\mathsf{CS}_0^2 \leftarrow \mathsf{Init}_2(1^\kappa, T_2, G_2(s))$ return $(\mathsf{CS}_0^1, \mathsf{CS}_0^2)$	if $t < T_1$ $\mathsf{UK}_t^1 \leftarrow \mathsf{Derive}_1(t, \mathsf{CS}_t^1)$ $\mathsf{UK}_t^2 \leftarrow \perp$ else $\mathsf{UK}_t^1 \leftarrow \mathsf{Derive}_1(T_1, \mathsf{CS}_t^1)$ $\mathsf{UK}_t^2 \leftarrow \mathsf{Derive}_2(t - T_1, \mathsf{CS}_t^2)$ return$(\mathsf{UK}_t^1, \mathsf{UK}_t^2)$
$\mathsf{Update}(t, (\mathsf{CS}_t^1, \mathsf{CS}_t^2))$: if $t < T_1$ $\mathsf{CS}_{t+1}^1 \leftarrow \mathsf{Update}_1(t, \mathsf{CS}_t^1)$ $\mathsf{CS}_{t+1}^2 \leftarrow \mathsf{CS}_t^2$ else $\mathsf{CS}_{t+1}^1 \leftarrow \mathsf{CS}_t^1$ $\mathsf{CS}_{t+1}^2 \leftarrow \mathsf{Update}_2(t - T_1, \mathsf{CS}_t^2)$ return $(\mathsf{CS}_{t+1}^1, \mathsf{CS}_{t+1}^2)$	$\mathsf{Extract}(t, (\mathsf{UK}_t^1, \mathsf{UK}_t^2), i)$: if $i > T_1$ $k_i \leftarrow \mathsf{Extract}_2(t - T_1, \mathsf{UK}_t^2, i - T_1)$ else if $t < T_1$ $k_i \leftarrow \mathsf{Extract}_1(t, \mathsf{UK}_t^1, i)$ else $k_i \leftarrow \mathsf{Extract}_1(T_1, \mathsf{UK}_t^1, i)$ return k_i

Figure 4.4: The additive composition of KU_1 and KU_2.

intervals, respectively, and that G is a pseudorandom generator as above. Then $\mathsf{KU} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Extract})$ *described in Figure 4.4 denoted as* $\mathsf{KU}_1 \oplus \mathsf{KU}_2$ *is a secure key-updating scheme with $T_1 + T_2$ time intervals:*

$$\begin{aligned}\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}}(T_1 + T_2, \tau) &\leq \mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_1}(T_1, \tau) + \mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_2}(T_2, \tau + \mathsf{TimeUserKeys}(T_1, \mathsf{KU}_1)) \\ &+ \mathsf{Adv}^{\mathtt{prg}}_{G}(\tau).\end{aligned}$$

Proof: Let $\mathcal{A} = (\mathcal{A}_{\mathcal{U}}, \mathcal{A}_{\mathcal{G}})$ be an adversary for KU running in time τ. We build two adversary algorithms $\mathcal{A}^1 = (\mathcal{A}^1_{\mathcal{U}}, \mathcal{A}^1_{\mathcal{G}})$ and $\mathcal{A}^2 = (\mathcal{A}^2_{\mathcal{U}}, \mathcal{A}^2_{\mathcal{G}})$ for KU_1 and KU_2, respectively.

Construction of $\mathcal{A}^1$. $\mathcal{A}^1$ simulates the environment for $\mathcal{A}$, by giving to $\mathcal{A}_{\mathcal{U}}$ at each iteration t the user key UK_t^1 that $\mathcal{A}^1_{\mathcal{U}}$ receives from the center. If $\mathcal{A}$ aborts or $\mathcal{A}_{\mathcal{U}}$ does not output stop until time interval $T_1 - 1$, then $\mathcal{A}^1$ outputs $\perp$ and aborts. Otherwise, $\mathcal{A}^1_{\mathcal{U}}$ outputs stop at the same time interval as $\mathcal{A}_{\mathcal{U}}$. In the challenge phase, $\mathcal{A}^1_{\mathcal{G}}$ receives as input a challenge key k_{t+1} and gives that to $\mathcal{A}_{\mathcal{G}}$. $\mathcal{A}^1_{\mathcal{G}}$ outputs the same bit as $\mathcal{A}_{\mathcal{G}}$. $\mathcal{A}^1$ has the same running time as $\mathcal{A}$. The success probability of $\mathcal{A}^1$ for $b \in \{0, 1\}$ is

$$\Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_1, \mathcal{A}^1}(T_1) = b\right] = \Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}, \mathcal{A}}(T_1 + T_2) = b \mid E_1 \cap E_2\right], \tag{4.3}$$

where E_1 is the event that $\mathcal{A}_{\mathcal{U}}$ outputs stop at a time interval strictly less than T_1 and E_2 the event that $\mathcal{A}$ does not distinguish the simulation done by $\mathcal{A}^1$ from the protocol execution.

The only difference between the simulation and the protocol execution is that the initial state for KU_1 is a random seed in the simulation and it is generated using a pseudorandom generator G in the protocol. If $\mathcal{A}$ distinguishes the simulation from the protocol, then a distinguisher algorithm for the pseudorandom generator can be constructed. By the definition of E_2, we have $\Pr[\bar{E_2}] \leq \mathsf{Adv}_G^{\mathtt{prg}}(\tau)$.

Construction of $\mathcal{A}^2$. $\mathcal{A}^2$ simulates the environment for $\mathcal{A}$: it first picks a random seed s of length κ and generates from $G_1(s)$ an instance of the scheme KU_1. For the first T_1 iterations of $\mathcal{A}_{\mathcal{U}}$, $\mathcal{A}^2$ gives to $\mathcal{A}_{\mathcal{U}}$ the user keys generated from s. If $\mathcal{A}$ aborts or $\mathcal{A}_{\mathcal{U}}$ stops at a time interval less than T_1, then $\mathcal{A}^2$ aborts the simulation. For the next T_2 time interval, $\mathcal{A}^2$ feeds $\mathcal{A}_{\mathcal{U}}$ the user keys received from the center. If $\mathcal{A}_{\mathcal{U}}$ outputs stop at a time interval $t \geq T_1$, then $\mathcal{A}^2_{\mathcal{U}}$ outputs stop at time interval $t - T_1$. In the challenge phase, $\mathcal{A}^2_{\mathcal{G}}$ receives a challenge k_{t-T_1+1}, gives this challenge to $\mathcal{A}_{\mathcal{G}}$ and outputs what $\mathcal{A}_{\mathcal{G}}$ outputs. $\mathcal{A}^2$ runs in time $\tau + \mathsf{TimeUserKeys}(T_1, \mathsf{KU}_1)$. The success probability of $\mathcal{A}^2$ for $b \in \{0,1\}$ is

$$\Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_2,\mathcal{A}^2}(T_2) = b\big] = \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1+T_2) = b \mid \bar{E_1}\big]. \tag{4.4}$$

We can relate the success probabilities of $\mathcal{A}$, $\mathcal{A}^1$, and $\mathcal{A}^2$ for $b \in \{0,1\}$ as follows:

$$\begin{aligned}
\Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1+T_2) = b\big] &= \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1+T_2) = b \cap E_1\big] + \\
&\quad \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1+T_2) = b \cap \bar{E_1}\big] \\
&= \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1+T_2) = b \cap E_1 \cap E_2\big] + \\
&\quad \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1+T_2) = b \cap E_1 \cap \bar{E_2}\big] + \\
&\quad \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1+T_2) = b \cap \bar{E_1}\big] \\
&\leq \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1+T_2) = b \mid E_1 \cap E_2\big]\Pr\big[E_1 \cap E_2\big] + \\
&\quad \Pr\big[\bar{E_2}\big] + \\
&\quad \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1+T_2) = b \mid \bar{E_1}\big]\Pr\big[\bar{E_1}\big] \\
&\leq \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_1,\mathcal{A}^1}(T_1) = b\big]\Pr\big[E_1\big] + \\
&\quad \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_2,\mathcal{A}^2}(T_2) = b\big]\Pr\big[\bar{E_1}\big] + \Pr\big[\bar{E_2}\big] \qquad (4.5) \\
&= p\Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_1,\mathcal{A}^1}(T_1) = b\big] + \\
&\quad (1-p)\Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_2,\mathcal{A}^2}(T_2) = b\big] + \Pr\big[\bar{E_2}\big],
\end{aligned}$$

where $p = \Pr[E_1]$ and (4.5) follows from (4.3) and (4.4). Finally, we can infer from (4.2)

$$\Pr[\mathcal{A} \text{ succeeds }] \leq p\Pr[\mathcal{A}^1 \text{ succeeds }] + (1-p)\Pr[\mathcal{A}^2 \text{ succeeds }] + \Pr\big[\bar{E_2}\big]$$

and

$$\begin{aligned}\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU},\mathcal{A}}(T_1+T_2) &\leq p\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_1,\mathcal{A}^1}(T_1)+(1-p)\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_2,\mathcal{A}^2}(T_2)+\mathsf{Adv}^{\mathtt{prg}}_{G}(\tau)\\ &\leq \mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_1,\mathcal{A}^1}(T_1)+\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_2,\mathcal{A}^2}(T_2)+\mathsf{Adv}^{\mathtt{prg}}_{G}(\tau). \qquad (4.6)\end{aligned}$$

The statement of the theorem follows from the last relation. □

Extended Additive Composition. It is immediate to extend the additive composition to construct a new scheme with $T_1 + T_2 + 1$ time intervals. The idea is to use the first scheme for the keys of the first T_1 intervals, the second scheme for the keys of the next T_2 intervals, and the seed s as the key for the $(T_1 + T_2 + 1)$-th interval. By revealing the seed s as the user key at interval $T_1 + T_2 + 1$, all previous keys of KU_1 and KU_2 can be derived. This idea will be useful in our later construction of a binary tree key-updating scheme. We call this composition method *extended additive composition*.

4.3.2 Multiplicative Composition

The idea behind the multiplicative composition operation is to use every key of the first scheme to seed an instance of the second scheme. Thus, for each one of the T_1 time intervals of the first scheme, we generate an instance of the second scheme with T_2 time intervals.

We denote a time interval t for $1 \leq t \leq T_1 \cdot T_2$ of scheme KU as a pair $t = <i, j>$, where i and j are such that $t = (i-1)T_2 + j$ for $1 \leq i \leq T_1$ and $1 \leq j \leq T_2$. The Update algorithm is run initially for time interval $t = 0$, which will be expressed as $<0, 0>$. The user key for a time interval $t = <i, j>$ includes both the user key for time interval $i - 1$ of scheme KU_1 and the user key for time interval j of scheme KU_2. A user receiving $\mathsf{UK}_{<i,j>}$ can extract the key for any time interval $<m, n> \leq <i, j>$ by first extracting the key K for time interval m of KU_1 (this step needs to be performed only if $m < i$), then using K to derive the initial state of the m-th instance of the scheme KU_2, and finally, deriving the key $k_{<m,n>}$. The details of the multiplicative composition method are shown in Figure 4.5.

The security of the multiplicative composition method is analyzed in the following theorem.

Theorem 7 *Suppose that* $\mathsf{KU}_1 = (\mathsf{Init}_1, \mathsf{Update}_1, \mathsf{Derive}_1, \mathsf{Extract}_1)$ *and* $\mathsf{KU}_2 = (\mathsf{Init}_2, \mathsf{Update}_2, \mathsf{Derive}_2, \mathsf{Extract}_2)$ *are two secure key-updating schemes with* T_1 *and* T_2 *time intervals, respectively, and that* G *is a pseudorandom generator as above. Then* $\mathsf{KU} =$

$\mathsf{Init}(1^\kappa, T, s)$: $\quad \mathsf{CS}_0^1 \leftarrow \mathsf{Init}_1(1^\kappa, T_1, G_1(s))$ $\quad \mathsf{CS}_1^1 \leftarrow \mathsf{Update}_1(0, \mathsf{CS}_0^1)$ $\quad k_1^1 \leftarrow \mathsf{Extract}_1(1, \mathsf{Derive}_1(1, \mathsf{CS}_1^1), 1)$ $\quad \mathsf{CS}_0^2 \leftarrow \mathsf{Init}_2(1^\kappa, T_2, G_2(k_1^1))$ $\quad$ return $(\perp, \mathsf{CS}_0^1, \mathsf{CS}_0^2)$	$\mathsf{Derive}(<i,j>, (\mathsf{CS}_{i-1}^1, \mathsf{CS}_i^1, \mathsf{CS}_j^2))$: $\quad$ if $i > 1$ $\quad\quad \mathsf{UK}_{i-1}^1 \leftarrow \mathsf{Derive}_1(i-1, \mathsf{CS}_{i-1}^1)$ $\quad$ else $\quad\quad \mathsf{UK}_{i-1}^1 \leftarrow \perp$ $\quad \mathsf{UK}_j^2 \leftarrow \mathsf{Derive}_2(j, \mathsf{CS}_j^2)$ $\quad$ return $(\mathsf{UK}_{i-1}^1, \mathsf{UK}_j^2)$
$\mathsf{Update}(<i,j>, (\mathsf{CS}_{i-1}^1, \mathsf{CS}_i^1, \mathsf{CS}_j^2))$: $\quad$ if $j = T_2$ $\quad\quad \mathsf{CS}_{i+1}^1 \leftarrow \mathsf{Update}_1(i, \mathsf{CS}_i^1)$ $\quad\quad k_{i+1}^1 \leftarrow \mathsf{Extract}_1(i+1,$ $\quad\quad\quad\quad \mathsf{Derive}_1(i+1, \mathsf{CS}_{i+1}^1), i+1)$ $\quad\quad \mathsf{CS}_0^2 \leftarrow \mathsf{Init}_2(1^\kappa, T_2, G_2(k_{i+1}^1))$ $\quad\quad \mathsf{CS}_1^2 \leftarrow \mathsf{Update}_2(0, \mathsf{CS}_0^2)$ $\quad\quad$ return $(\mathsf{CS}_i^1, \mathsf{CS}_{i+1}^1, \mathsf{CS}_1^2)$ $\quad$ else $\quad\quad \mathsf{CS}_{j+1}^2 \leftarrow \mathsf{Update}_2(j, \mathsf{CS}_j^2)$ $\quad\quad$ return $(\mathsf{CS}_{i-1}^1, \mathsf{CS}_i^1, \mathsf{CS}_{j+1}^2)$	$\mathsf{Extract}(<i,j>, (\mathsf{UK}_{i-1}^1, \mathsf{UK}_j^2), <m,n>)$: $\quad$ if $i = m$ $\quad\quad k_{<m,n>} \leftarrow \mathsf{Extract}_2(j, \mathsf{UK}_j^2, n)$ $\quad$ else $\quad\quad K \leftarrow \mathsf{Extract}_1(i-1, \mathsf{UK}_{i-1}^1, m)$ $\quad\quad \mathsf{CS}_0^2 \leftarrow \mathsf{Init}_2(1^\kappa, T_2, G_2(K))$ $\quad\quad k_{<m,n>} \leftarrow \mathsf{Extract}_2(T_2, \mathsf{CS}_0^2, n)$ $\quad$ return $k_{<m,n>}$

Figure 4.5: The multiplicative composition of KU_1 and KU_2.

(Init, Update, Derive, Extract) *described in Figure 4.5 denoted as* $\mathsf{KU}_1 \otimes \mathsf{KU}_2$ *is a secure key-updating scheme with* $T_1 \cdot T_2$ *time intervals:*

$$\mathsf{Adv}_{\mathsf{KU}}^{\mathtt{sku}}(T_1 \cdot T_2, \tau) \leq \mathsf{Adv}_{\mathsf{KU}_1}^{\mathtt{sku}}(T_1, \tau + \tau_1 + \tau_2 + \tau_3) + q\mathsf{Adv}_{\mathsf{KU}_2}^{\mathtt{sku}}(T_2, \tau + \tau_4) + \mathsf{Adv}_G^{\mathtt{prg}}(\tau),$$

where $\tau_1 = \mathsf{TimeKeys}(T_1, \mathsf{KU}_1)$, $\tau_2 = \mathsf{TimeUserKeys}(T_2, \mathsf{KU})$, $\tau_3 = \mathsf{TimeMaxExtract}(T_2, \mathsf{KU}_2)$ *and* $\tau_4 = \mathsf{TimeUserKeys}(T_1, \mathsf{KU}_1)$.

Proof: Let $\mathcal{A} = (\mathcal{A}_\mathcal{U}, \mathcal{A}_\mathcal{G})$ be an adversary for KU running in time τ. We build two adversary algorithms $\mathcal{A}^1 = (\mathcal{A}_\mathcal{U}^1, \mathcal{A}_\mathcal{G}^1)$ and $\mathcal{A}^2 = (\mathcal{A}_\mathcal{U}^2, \mathcal{A}_\mathcal{G}^2)$ for KU_1 and KU_2, respectively.

Construction of $\mathcal{A}^1$. $\mathcal{A}_\mathcal{U}^1$ gets from the center the user keys UK_i^1 of scheme KU_1 for all time intervals i until it outputs stop. $\mathcal{A}^1$ simulates the environment for $\mathcal{A}$ by sending the following user keys:

1. At interval $<i,1>$, for $1 \leq i \leq T_1$, $\mathcal{A}^1$ runs $k_i \leftarrow \mathsf{Extract}_1(i, \mathsf{UK}_i^1, i)$; $\mathsf{CS}_0^2 \leftarrow \mathsf{Init}_2(1^\kappa, T_2, G_2(k_i))$; $\mathsf{CS}_1^2 \leftarrow \mathsf{Update}_2(0, \mathsf{CS}_0^2)$; $\mathsf{UK}_1^2 \leftarrow \mathsf{Derive}_2(1, \mathsf{CS}_1^2)$ and gives $\mathcal{A}_\mathcal{U}$ the user key $\mathsf{UK}_{<i,1>} = (\mathsf{UK}_{i-1}^1, \mathsf{UK}_1^2)$.

2. At time interval $<i,j>$, for $1 \leq i \leq T_1$ and $1 < j \leq T_2$, $\mathcal{A}_{\mathcal{U}}^1$ computes $\mathsf{CS}_j^2 \leftarrow \mathsf{Update}_2(j-1, \mathsf{CS}_{j-1}^2)$ and $\mathsf{UK}_j^2 \leftarrow \mathsf{Derive}_2(j, \mathsf{CS}_j^2)$ and gives to $\mathcal{A}_{\mathcal{U}}$ the user key $\mathsf{UK}_{<i,j>} = (\mathsf{UK}_{i-1}^1, \mathsf{UK}_j^2)$.

If $\mathcal{A}$ aborts or $\mathcal{A}_{\mathcal{U}}$ outputs stop at a time interval $<i,j>$ with $j \neq T_2$, then $\mathcal{A}_{\mathcal{U}}^1$ aborts the simulation and outputs $\perp$. Otherwise, $\mathcal{A}_{\mathcal{U}}^1$ outputs stop at time interval i. In the challenge interval, $\mathcal{A}_{\mathcal{G}}^1$ is given a challenge key k_{i+1} and it executes $\mathsf{CS}_0^2 \leftarrow \mathsf{Init}_2(1^\kappa, T_2, G_2(k_{i+1}))$; $\mathsf{CS}_1^2 \leftarrow \mathsf{Update}_2(0, \mathsf{CS}_0^2)$; $M \leftarrow \mathsf{Derive}_2(1, \mathsf{CS}_1^2)$; $k_1^2 \leftarrow \mathsf{Extract}_2(1, M, 1)$. It then gives the challenge k_1^2 to $\mathcal{A}_{\mathcal{G}}$. $\mathcal{A}_{\mathcal{G}}^1$ outputs the same bit as $\mathcal{A}_{\mathcal{G}}$. The running time of $\mathcal{A}^1$ is $\tau + \tau_1 + \tau_2 + \tau_3$. The success probability of $\mathcal{A}^1$ for $b \in \{0,1\}$ is

$$\Pr\big[\mathsf{Exp}_{\mathsf{KU}_1,\mathcal{A}^1}^{\mathtt{sku}\text{-}b}(T_1) = b\big] = \Pr\big[\mathsf{Exp}_{\mathsf{KU},\mathcal{A}}^{\mathtt{sku}\text{-}b}(T_1 \cdot T_2) = b \mid E_1 \cap E_2\big], \quad (4.7)$$

where E_1 is the event that $\mathcal{A}_{\mathcal{U}}$ outputs stop at a time interval (i,j) with $j = T_2$ and E_2 the event that $\mathcal{A}$ does not distinguish the simulation done by $\mathcal{A}^1$ from the protocol execution. If $\mathcal{A}$ distinguishes the simulation from the protocol, then a distinguisher algorithm for the pseudorandom generator can be constructed. By the definition of E_2, we have $\Pr[\bar{E}_2] \leq \mathsf{Adv}_G^{\mathtt{prg}}(\tau)$.

Construction of $\mathcal{A}^2$. Assuming that $\mathcal{A}_{\mathcal{U}}$ runs at most q times (and q is polynomial in κ), $\mathcal{A}^2$ makes a guess for the time interval i^* in which $\mathcal{A}_{\mathcal{U}}$ outputs stop. $\mathcal{A}^2$ picks i^* uniformly at random from the set $\{1, \ldots, q\}$. $\mathcal{A}^2$ generates an instance of the scheme KU_1 with i^* time intervals. For any interval $<i,j>$ with $i < i^*$, $\mathcal{A}^2$ generates the user keys using the keys from this instance of KU_1. For time intervals $<i^*,j>$ with $1 \leq j \leq T_2$, $\mathcal{A}^2$ outputs user key $(\mathsf{UK}_{i^*-1}^1, \mathsf{UK}_j^2)$, where $\mathsf{UK}_{i^*-1}^1$ is the user key for time interval i^*-1 of KU_1 that it generated itself and UK_j^2 is the user key for time interval j of KU_2 that it received from the center.

If $\mathcal{A}$ aborts or $\mathcal{A}_{\mathcal{U}}$ outputs stop at a time interval $<i,j>$ with $i \neq i^*$ or with $i = i^*$ and $j = T_2$, then $\mathcal{A}^2$ aborts the simulation and outputs $\perp$. Otherwise, if $\mathcal{A}_{\mathcal{U}}$ outputs stop at a time interval $<i^*,j>$, then $\mathcal{A}_{\mathcal{U}}^2$ outputs stop at time interval j. In the challenge phase, $\mathcal{A}^2$ receives a challenge key k_{j+1} and gives that to $\mathcal{A}_{\mathcal{G}}$. $\mathcal{A}_{\mathcal{G}}^2$ outputs the same bit as $\mathcal{A}_{\mathcal{G}}$. The running time of $\mathcal{A}^2$ is $\tau + \tau_4$. The success probability of $\mathcal{A}^2$ for $b \in \{0,1\}$ is

$$\Pr\big[\mathsf{Exp}_{\mathsf{KU}_2,\mathcal{A}^2}^{\mathtt{sku}\text{-}b}(T_2) = b\big] = \frac{1}{q}\Pr\big[\mathsf{Exp}_{\mathsf{KU},\mathcal{A}}^{\mathtt{sku}\text{-}b}(T_1 \cdot T_2) = b \mid \bar{E}_1 \cap E_2\big]. \quad (4.8)$$

We can infer

$$\Pr\big[\mathsf{Exp}_{\mathsf{KU},\mathcal{A}}^{\mathtt{sku}\text{-}b}(T_1 \cdot T_2) = b\big] = \Pr\big[\mathsf{Exp}_{\mathsf{KU},\mathcal{A}}^{\mathtt{sku}\text{-}b}(T_1 \cdot T_2) = b \cap E_1 \cap E_2\big] + \quad (4.9)$$

$$
\begin{aligned}
& \Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1 \cdot T_2) = b \cap \bar{E}_1 \cap E_2\right] + && (4.10)\\
& \Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1 \cdot T_2) = b \cap \bar{E}_2\right] \\
\leq\ & \Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1 \cdot T_2) = b \mid E_1 \cap E_2\right] \Pr\left[E_1 \cap E_2\right] + \\
& \Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}}(T_1 \cdot T_2) = b \mid \bar{E}_1 \cap E_2\right] \Pr\left[\bar{E}_1 \cap E_2\right] + \\
& \Pr[\bar{E}_2] \\
=\ & \Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_1,\mathcal{A}^1}(T_1) = b\right] \Pr\left[E_1 \cap E_2\right] + \\
& q \Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_2,\mathcal{A}^2}(T_2) = b\right] \Pr\left[\bar{E}_1 \cap E_2\right] + \\
& \Pr[\bar{E}_2] && (4.11)\\
\leq\ & p \Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_1,\mathcal{A}^1}(T_1) = b\right] + \\
& (1-p) q \Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_2,\mathcal{A}^2}(T_2) = b\right] + \Pr[\bar{E}_2],
\end{aligned}
$$

where $p = \Pr[E_1]$ and (4.11) follows from (4.7) and (4.8). Finally we can infer from (4.2) that

$$
\begin{aligned}
\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU},\mathcal{A}}(T_1 \cdot T_2) &\leq p\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_1,\mathcal{A}^1}(T_1) + (1-p)q\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_2,\mathcal{A}^2}(T_2) + \mathsf{Adv}^{\mathtt{prg}}_{G}(\tau) \\
&\leq \mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_1,\mathcal{A}^1}(T_1) + q\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_2,\mathcal{A}^2}(T_2) + \mathsf{Adv}^{\mathtt{prg}}_{G}(\tau). \qquad (4.12)
\end{aligned}
$$

The statement of the theorem follows from the last relation. ∎

4.4 Constructions of Key-Updating Schemes

In this section, we describe three constructions of key-updating schemes with different complexity and communication tradeoffs. The first two constructions are based on previously proposed methods (Kallahalla et al. [2003], Fu et al. [2006]). We give cryptographic proofs that demonstrate the security of the existing constructions after some subtle modifications. Additionally, we propose a third construction that is more efficient than the known schemes. It uses a binary tree to derive the user keys and is also provably secure in our model.

4.4.1 Chaining Construction (CKU)

In this construction, the center generates an initial random seed of length κ and applies a pseudorandom generator iteratively i times to obtain the key for time interval $T-i$, for $1 \leq i \leq T-1$. This construction is inspired by a folklore method using a hash chain

for deriving the keys. A construction based on a hash chain can be proven secure if the hash function h is modeled as a random oracle. To obtain a provably secure scheme in the standard model, we replace the hash function with a pseudorandom generator.

Let $G : \{0,1\}^\kappa \rightarrow \{0,1\}^{2\kappa}$ be a pseudorandom generator. We write $G(s) = G_1(s) \| G_2(s)$ with $|G_1(s)| = |G_2(s)| = \kappa$ for $s \in \{0,1\}^\kappa$. The algorithms of the chaining construction, called CKU, are:

- $\mathsf{Init}(1^\kappa, T, s)$ generates a random seed s_0 of length κ from s and outputs $\mathsf{CS}_0 = s_0$.
- $\mathsf{Update}(t, \mathsf{CS}_t)$ copies the state CS_t into CS_{t+1}.
- $\mathsf{Derive}(t, \mathsf{CS}_t)$ and $\mathsf{Extract}(t, \mathsf{UK}_t, i)$ are given in Figure 4.6.

$\mathsf{Derive}(t, \mathsf{CS}_t)$:	$\mathsf{Extract}(t, \mathsf{UK}_t, i)$:
$B_{T+1} \leftarrow \mathsf{CS}_t$	$(B_t, k_t) \leftarrow \mathsf{UK}_t$
for $i = T$ downto t	for $j = t-1$ downto i
$(B_i, k_i) \leftarrow G(B_{i+1})$	$(B_j, k_j) \leftarrow G(B_{j+1})$
return $\mathsf{UK}_t \leftarrow (B_t, k_t)$	return k_i

Figure 4.6: The $\mathsf{Derive}(t, \mathsf{CS}_t)$ and $\mathsf{Extract}(t, \mathsf{UK}_t, i)$ algorithms of the chaining construction.

This construction has constant center-state size and linear cost for the user-key derivation algorithm. An alternative construction with linear center-state size and constant user-key derivation is to precompute all the keys k_i and user keys UK_i, for $1 \le i \le T$ in the Init algorithm and store all of them in the initial center state CS_0. The security of the chaining construction is given by the following theorem.

Theorem 8 *Given a pseudorandom generator G,* CKU *is a secure key-updating scheme:*

$$\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{CKU}}(T, \tau) \le \mathsf{Adv}^{\mathtt{prg}}_{G}(\tau + \mathsf{TimeRand}(\kappa) + T \cdot \mathsf{TimePRG}(G)) + (T-1)\mathsf{Adv}^{\mathtt{prg}}_{G}(\tau),$$

where $\mathsf{TimePRG}(G)$ *is the time to compute one application of G and* $\mathsf{TimeRand}(k)$ *is the time to generate a uniformly random number of length k bits.*

Proof: Let $\mathcal{A} = (\mathcal{A}_\mathcal{U}, \mathcal{A}_\mathcal{G})$ be an adversary for the chaining key-updating scheme running in time τ. We construct an algorithm $\mathcal{D}$ that distinguishes the output of the pseudorandom generator from a random string of length 2κ with sufficiently large probability.

Algorithm $\mathcal{D}$ has to simulate the environment for $\mathcal{A}$. $\mathcal{D}$ picks B_{T+1} uniformly at random from $\{0,1\}^\kappa$ and computes the user keys for previous time intervals as $(B_i, k_i) = G(B_{i+1})$, for $i = T, \ldots, 1$. $\mathcal{D}$ gives to $\mathcal{A}_\mathcal{U}$ user key $\mathsf{UK}_i = (B_i, k_i)$ at iteration i.

Algorithm $\mathcal{D}$ is given a challenge string $r = r_0 \| r_1$ of length 2κ, which in experiment $\mathsf{Exp}^{\mathtt{prg\text{-}0}}_{G,\mathcal{D}}$ is the output of the pseudorandom generator on input a random seed of length κ, and in experiment $\mathsf{Exp}^{\mathtt{prg\text{-}1}}_{G,\mathcal{D}}$ is a random string of length 2κ (see the experiments defined in Figure 4.1).

If $\mathcal{A}_\mathcal{U}$ outputs stop at time interval t, $\mathcal{D}$ gives to $\mathcal{A}_\mathcal{G}$ the challenge key $k_{t+1} = r_1$ and $\mathcal{D}$ outputs what $\mathcal{A}_\mathcal{G}$ outputs. The running time of $\mathcal{D}$ is $\tau + \mathsf{TimeRand}(\kappa) + T \cdot \mathsf{TimePRG}(G)$. Denote by $p_b = \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{CKU},\mathcal{A}}(T) = b\big]$. It is immediate that

$$\Pr\big[\mathsf{Exp}^{\mathtt{prg\text{-}1}}_{G,\mathcal{D}} = 1\big] = \Pr\big[\mathsf{Exp}^{\mathtt{sku\text{-}1}}_{\mathsf{CKU},\mathcal{A}}(T) = 1\big] = p_1, \tag{4.13}$$

and

$$\Pr\big[\mathsf{Exp}^{\mathtt{prg\text{-}0}}_{G,\mathcal{D}} = 0\big] = p_0', \tag{4.14}$$

where p_0' is the probability that $\mathcal{A}$, given the user keys as in experiment $\mathsf{Exp}^{\mathtt{sku\text{-}0}}_{\mathsf{CKU},\mathcal{A}}(T)$, but challenge key $k_{t+1} = G_2(s)$ for a random seed $s \in \{0,1\}^\kappa$, outputs 0. The challenge key given to $\mathcal{A}$ in experiment $\mathsf{Exp}^{\mathtt{sku\text{-}0}}_{\mathsf{CKU},\mathcal{A}}(T)$ is $G_2(G_1^{T-t-1}(s))$, where $G_1^i(s) = G_1(\ldots G_1(s)\ldots)$ for i applications of G_1. We can bound the absolute difference between p_0 and p_0' as

$$\begin{aligned} |p_0' - p_0| &\le \Pr\big[\mathcal{A} \text{ distinguishes between } G_2(s) \text{ and } G_2(G_1^{T-t-1}(s))\big] \\ &\le (T-t)\Pr\big[\mathcal{A} \text{ distinguishes between } s \xleftarrow{R} \{0,1\}^\kappa \text{ and } G_1(s)\big] \\ &\le (T-t)\mathsf{Adv}^{\mathtt{prg}}_G(\tau). \end{aligned} \tag{4.15}$$

Using (4.13), (4.14) and (4.15), we can relate the success probabilities of $\mathcal{A}$ and $\mathcal{D}$ by

$$\begin{aligned} \Pr\big[\mathcal{D} \text{ succeeds}\big] &= \frac{1}{2}\big(\Pr\big[\mathsf{Exp}^{\mathtt{prg\text{-}0}}_{G,\mathcal{D}} = 0\big] + \Pr\big[\mathsf{Exp}^{\mathtt{prg\text{-}1}}_{G,\mathcal{D}} = 1\big]\big) \\ &= \frac{1}{2}\big(p_0' + p_1\big) \\ &= \frac{1}{2}(p_0 + p_1 + p_0' - p_0) \\ &\ge \Pr\big[\mathcal{A} \text{ succeeds}\big] - \frac{1}{2}(T-t)\mathsf{Adv}^{\mathtt{prg}}_G(\tau). \end{aligned}$$

It follows that

$$\Pr\big[\mathcal{A} \text{ succeeds}\big] \le \Pr\big[\mathcal{D} \text{ succeeds}\big] + \frac{1}{2}(T-t)\mathsf{Adv}^{\mathtt{prg}}_G(\tau),$$

and

$$\begin{aligned}\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{CKU},\mathcal{A}}(T) &\leq \mathsf{Adv}^{\mathtt{prg}}_{G,\mathcal{D}} + (T-t)\mathsf{Adv}^{\mathtt{prg}}_{G}(\tau) \\ &\leq \mathsf{Adv}^{\mathtt{prg}}_{G,\mathcal{D}} + (T-1)\mathsf{Adv}^{\mathtt{prg}}_{G}(\tau).\end{aligned}$$

The statement of the theorem follows from last relation. ∎

4.4.2 Trapdoor Permutation Construction (TDKU)

In this construction, the center picks an initial random state that is updated at each time interval by applying the inverse of a trapdoor permutation. The trapdoor is known only to the center, but a user, given the state at a certain moment, can apply the permutation iteratively to generate all previous states. The key for a time interval is generated by applying a hash function, modeled as a random oracle, to the current state. This idea underlies the key rotation mechanism of the Plutus file system (Kallahalla et al. [2003]), with the difference that Plutus uses the output of an RSA trapdoor permutation directly for the encryption key. We could not prove the security of this scheme in our model for key-updating schemes, even when the trapdoor permutation is not arbitrary, but instantiated with the RSA permutation.

This construction has the advantage that knowledge of the total number of time intervals is not needed in advance; on the other hand, its security can only be proved in the random oracle model. Let a family of trapdoor permutations be given such that the domain size of the permutations with security parameter κ is $l(\kappa)$, for some polynomial l. Let $h : \{0,1\}^{l(\kappa)} \rightarrow \{0,1\}^{\kappa}$ be a hash function modeled as a random oracle. The detailed construction of the trapdoor permutation scheme, called TDKU, is presented below:

- $\mathsf{Init}(1^{\kappa}, T, s)$ generates a random $s_0 \xleftarrow{R} \{0,1\}^{l(\kappa)}$ and a trapdoor permutation $f : \{0,1\}^{l(\kappa)} \rightarrow \{0,1\}^{l(\kappa)}$ with trapdoor τ from seed s using a pseudorandom generator. Then it outputs $\mathsf{CS}_0 = (s_0, f, \tau)$.

- $\mathsf{Update}(t, \mathsf{CS}_t)$ with $\mathsf{CS}_t = (s_t, f, \tau)$ computes $s_{t+1} = f^{-1}(s_t)$ and outputs $\mathsf{CS}_{t+1} = (s_{t+1}, f, \tau)$.

- $\mathsf{Derive}(t, \mathsf{CS}_t)$ outputs $\mathsf{UK}_t \leftarrow (s_t, f)$.

- $\mathsf{Extract}(t, \mathsf{UK}_t, i)$ applies the permutation iteratively $t-i$ times to generate state $s_i = f^{t-i}(\mathsf{UK}_t)$ and then outputs $h(s_i)$.

The security of this construction is given be the following theorem.

Theorem 9 *Given a family of trapdoor permutations and a hash function h modeled as a random oracle,* TDKU *is a secure key-updating scheme in the random oracle model:*

$$\mathsf{Adv}^{\mathsf{sku}}_{\mathsf{TDKU}}(T,\tau) \leq 2T\mathsf{Adv}^{\mathsf{one\text{-}way}}_{\mathcal{F}}(\tau + \tau_1(T) + T \cdot \mathsf{TimeTrap}(f)),$$

where $\tau_1(n)$ is the time to pick a number uniformly at random from $\{1, 2, \ldots, n\}$ and $\mathsf{TimeTrap}(f)$ *is the time to compute an application of f.*

Proof: Let $\mathcal{A} = (\mathcal{A}_{\mathcal{U}}, \mathcal{A}_{\mathcal{G}})$ be an adversary for the trapdoor key-updating scheme running in time τ. Assuming that $\mathcal{A}_{\mathcal{U}}$ runs at most T times, we construct an adversary $\mathcal{A}'$ for the one-wayness of $\mathcal{F}$, which given f and $y \leftarrow f(x)$ with $x \stackrel{R}{\leftarrow} \{0,1\}^{l(\kappa)}$ computes $f^{-1}(y)$ with sufficiently large probability.

Algorithm $\mathcal{A}'$ has to simulate the environment for $\mathcal{A}$. $\mathcal{A}'$ makes a guess at the time interval t^* in which $\mathcal{A}_{\mathcal{U}}$ outputs stop. $\mathcal{A}'$ picks t^* uniformly at random from the set $\{1, \ldots, T\}$. If $\mathcal{A}_{\mathcal{U}}$ does not output stop at time interval t^*, then $\mathcal{A}'$ aborts the simulation. Otherwise, at time interval t less than t^*, $\mathcal{A}'$ gives to $\mathcal{A}_{\mathcal{U}}$ the user key $\mathsf{UK}_t = (f^{t^*-t}(y), f)$. $\mathcal{A}'$ runs in time $\tau + \tau_1(T) + T \cdot \mathsf{TimeTrap}(f)$.

Algorithm Extract is executed by $\mathcal{A}$ as in the description of the scheme, but $\mathcal{A}'$ simulates the random oracle for $\mathcal{A}$. If $\mathcal{A}$ queries x to the random oracle for which $f(x) = y$, then $\mathcal{A}'$ outputs x. Let E be the event that $\mathcal{A}$ asks query $x = f^{-1}(y)$ to the oracle and $\bar{E}$ the negation of this event. Since the adversary has no advantage in distinguishing the properly generated key k_{t+1} from a randomly generated key if it does not query the random oracle at x, it follows that

$$\Pr[\mathcal{A} \text{ succeeds } \mid \bar{E}] \leq \frac{1}{2},$$

from which we can infer

$$\begin{aligned}
\Pr[\mathcal{A} \text{ succeeds }] &= \Pr[\mathcal{A} \text{ succeeds } \mid E]\Pr[E] + \Pr[\mathcal{A} \text{ succeeds } \mid \bar{E}]\Pr[\bar{E}] \\
&\leq \Pr[E] + \frac{1}{2}. \qquad (4.16)
\end{aligned}$$

Equations (4.2) and (4.16) imply that $\Pr[E] \geq \frac{1}{2}\mathsf{Adv}^{\mathsf{sku}}_{\mathsf{TDKU},\mathcal{A}}(T)$. Then the success probability of algorithm $\mathcal{A}'$ is

$$\mathsf{Adv}^{\mathsf{one\text{-}way}}_{\mathcal{F},\mathcal{A}'} = \frac{1}{T}\Pr[E] \geq \frac{1}{2T}\mathsf{Adv}^{\mathsf{sku}}_{\mathsf{TDKU},\mathcal{A}}(T).$$

The statement of the theorem follows from the last relation.

□

4.4.3 Tree Construction (TreeKU)

In the two schemes above, at least one of the algorithms Update, Derive and Extract has worst-case complexity linear in the total number of time intervals. We present a tree construction based on ideas of Canetti at al. (Canetti et al. [2003]) with constant complexity for the Derive algorithm and logarithmic worst-case complexity in the number of time intervals for the Update and Extract algorithms. Moreover, the amortized complexity of the Update algorithm is constant. In this construction, the user key size is increased by at most a logarithmic factor in T compared to the user key size of the two constructions described above.

Our tree-based key-updating scheme, called TreeKU, generates keys using a complete binary tree with T nodes, assuming that $T = 2^d - 1$ for some $d \in \mathbf{Z}$. Each node in the tree is associated with a time interval between 1 and T, a unique label in $\{0,1\}^*$, a *tree-key* in $\{0,1\}^\kappa$ and an *external key* in $\{0,1\}^\kappa$ such that:

1. Time intervals are assigned to tree nodes using post-order tree traversal, i.e., a node corresponds to interval i if it is the i-th node in the post-order traversal of the tree. We refer to the node associated with interval t as node t.

2. We define a function Label that maps node t with $1 \le t \le T$ to its label in $\{0,1\}^*$ as follows. The root of the tree is labeled by the empty string ε, and the left and right children of a node with label ℓ are labeled by $\ell \| 0$ and by $\ell \| 1$, respectively. The parent of a node with label ℓ is denoted by $\mathsf{parent}(\ell)$, thus $\mathsf{parent}(\ell \| 0) = \mathsf{parent}(\ell \| 1) = \ell$. We denote the length of a label ℓ by $|\ell|$.

3. The tree-key for the root node is chosen at random. The tree-keys for the two children of an internal node in the tree are derived from the tree-key of the parent node using a pseudorandom generator $G : \{0,1\}^\kappa \to \{0,1\}^{2\kappa}$. For an input $s \in \{0,1\}^\kappa$, we write $G(s) = G_1(s) \| G_2(s)$ with $|G_1(s)| = |G_2(s)| = \kappa$. If the tree-key for the internal node with label ℓ is denoted u_ℓ, then the tree-keys for its left and right children are $u_{\ell\|0} = G_1(u_\ell)$ and $u_{\ell\|1} = G_2(u_\ell)$, respectively. This implies that once the tree-key for a node is revealed, then the tree-keys of its children can be computed, but knowing the tree-keys of both children of a node does not reveal any information about the tree-key of the node.

4. The external key of a node t is the key k_t output by the scheme to the application for interval t. For a node t with tree-key $u_{\mathsf{Label}(t)}$, the external key k_t is obtained by computing $\mathsf{PRF}_{u_{\mathsf{Label}(t)}}(1)$, where $\mathsf{PRF}_u(b) = \mathsf{PRF}(u, b)$ and $\mathsf{PRF} : \{0,1\}^\kappa \times \{0,1\} \to \{0,1\}^\kappa$ is a pseudorandom function on bits.

We describe the four algorithms of the binary tree key-updating scheme:

```
Update(t, (P_t, L_t))
  if t = 0
    P_1 ← LeftKeys(ε, u_T)                     /* contains the label/tree-key pairs of the left-most nodes */
    L_1 ← ∅                                    /* the set of left siblings is empty */
  else
    ℓ_t ← Label(t)                             /* compute the label of node t */
    u_t ← SearchKey(ℓ_t, P_t)                  /* compute the tree-key of node t */
    if ℓ_t ends in 0                           /* t is the left child of its parent */
      (ℓ_s, u_s) ← RightSib(ℓ_t, P_t)          /* compute the label and tree-key of the right sibling of t */
      P_{t+1} ← P_t \ {(ℓ_t, u_t)} ∪ LeftKeys(ℓ_s, u_s)   /* update the label/tree-key pair in P_{t+1} */
      L_{t+1} ← L_t ∪ {(ℓ_t, u_t)}             /* add the label and tree-key of t to the left siblings of t + 1 */
    else                                       /* t is the right child of its parent */
      (ℓ_s, u_s) ← LeftSib(ℓ_t, L_t)           /* compute the label and tree-key of the left sibling of t */
      P_{t+1} ← P_t \ {(ℓ_t, u_t)}             /* remove label/tree-key pair of t from P_{t+1} */
      L_{t+1} ← L_t \ {(ℓ_s, u_s)}             /* remove label/tree-key pair of left sibling of t from L_{t+1} */
  return (P_{t+1}, L_{t+1})

LeftKeys(ℓ, u)
  A ← ∅                                        /* initialize set A with the empty set */
  while |ℓ| ≤ d                                /* advance to the left until we reach a leaf */
    A ← A ∪ {(ℓ, u)}                           /* add the label and tree-key of the current node in A */
    ℓ ← ℓ||0                                   /* move to left child of the node with label p */
    u ← G_1(u)                                 /* compute the tree-key of the left child */
  return A
```

Figure 4.7: The $\mathsf{Update}(t, (P_t, L_t))$ algorithm.

- $\mathsf{Init}(1^\kappa, T, s)$ generates the tree-key for the root node randomly, $u_T \xleftarrow{R} \{0,1\}^\kappa$, using seed s, and outputs $\mathsf{CS}_0 = (\{(\varepsilon, u_T)\}, \emptyset)$.

- $\mathsf{Update}(t, \mathsf{CS}_t)$ updates the state $\mathsf{CS}_t = (P_t, L_t)$ to the next center state $\mathsf{CS}_{t+1} = (P_{t+1},$ $L_{t+1})$. The center state for interval t consists of two sets: P_t that contains pairs of (label, tree-key) for all nodes on the path from the root to node t (including node t), and L_t that contains label/tree-key pairs for all left siblings of the nodes in P_t that are not in P_t.

  We use several functions in the description of the Update algorithm. For a label ℓ and a set A of label/tree-key pairs, we define a function $\mathsf{SearchKey}(\ell, A)$ that outputs a tree-key u for which $(\ell, u) \in A$, if the label exists in the set, and $\perp$ otherwise. Given

a label ℓ and a set of label/tree-key pairs A, function $\mathsf{RightSib}(\ell, A)$ returns the label and the tree-key of the right sibling of the node with label ℓ, and, similarly, function $\mathsf{LeftSib}(\ell, A)$ returns the label and the tree-key of the left sibling of the node with label ℓ (assuming the labels and tree-keys of the siblings are in A). The function LeftKeys is given as input a label/tree-key pair of a node and returns all label/tree-key pairs of the left-most nodes in the subtree rooted at the input node, including label and tree-key of the input node.

The code for the Update and LeftKeys algorithms is given in Figure 4.7. We omit the details of functions SearchKey, RightSib and LeftSib. The Update algorithm distinguishes three cases:

1. If $t = 0$, the Update algorithm computes the label/tree-key pairs of all left-most nodes in the complete tree using function LeftKeys and stores them in P_1. The set L_1 is empty in this case, as nodes in P_1 do not have left siblings.
2. If t is the left child of its parent, the successor of node t in post-order traversal is the left-most node in the subtree rooted at the right sibling t' of node t. P_{t+1} contains all label/tree-key pairs in P_t except the tuple for node t, and, in addition, all label/tree-key pairs for the left-most nodes in the subtree rooted at t', which are computed by LeftKeys. The set of left siblings L_{t+1} contains all label/tree-key pairs from L_t and, in addition, the label/tree-key pair for node t.
3. If t is the right child of its parent, node $t + 1$ is its parent, so P_{t+1} contains all label/tree-key pairs from P_t except the tuple for node t, and L_{t+1} contains all the label/tree-key pairs in L_t except the pair for the left sibling of node t.

- Algorithm $\mathsf{Derive}(t, (P_t, L_t))$ outputs the user tree-key UK_t, which is the minimum information needed to generate the set of tree-keys $\{u_i : i \leq t\}$. Since the tree-key of any node reveals the tree-keys for all nodes in the subtree rooted at that node, UK_t consists of the label/tree-key pairs for the left siblings (if any) of all nodes on the path from the root to the parent of node t and the label/tree-key pair of node t. This information has already been pre-computed such that one can set $\mathsf{UK}_t \leftarrow \{(\mathsf{Label}(t), u_t)\} \cup L_t$.

- Algorithm $\mathsf{Extract}(t, \mathsf{UK}_t, i)$ first finds the maximum predecessor of node i in post-order traversal whose label/tree-key pair is included in the user tree-key UK_t. Then it computes the tree-keys for all nodes on the path from that predecessor to node i. The external key k_i is derived from the tree-key u_i as $k_i = \mathsf{PRF}_{u_i}(1)$ using the pseudorandom function. The algorithm is in Figure 4.8.

$\mathsf{Extract}(t, \mathsf{UK}_t, i)$
$\ell_1 \ldots \ell_s \leftarrow \mathsf{Label}(i)$ /* the label of i has length s */
$v \leftarrow s$
$\ell \leftarrow \ell_1 \ldots \ell_v$
while $v > 0$ and $\mathsf{SearchKey}(\ell, \mathsf{UK}_t) = \perp$ /* find a predecessor of i that is in UK_t */
$v \leftarrow v - 1$
$\ell \leftarrow \ell_1 \ldots \ell_v$
for $j = v + 1$ to s /* compute tree-keys of all nodes on path from predecessor to i */
$u_{\ell_1 \ldots \ell_j} \leftarrow G_{\ell_j}(u_{\ell_1 \ldots \ell_{j-1}})$
$k_{\ell_1 \ldots \ell_s} \leftarrow \mathsf{PRF}_{u_{\ell_1 \ldots \ell_s}}(1)$ /* return external key of node i */
return $k_{\ell_1 \ldots \ell_s}$

Figure 4.8: The $\mathsf{Extract}(t, \mathsf{UK}_t, i)$ algorithm.

Analysis of Complexity. The worst-case complexity of the cryptographic operations used in the Update and Extract algorithms is logarithmic in the number of time intervals, and that of Derive is constant. However, it is easy to see that the key for each node is computed exactly once if T updates are executed. This implies that the total cost of all update operations is T pseudorandom-function applications, so the amortized cost per update is constant. The size of the center state and the user key is proportional to the height of the binary tree, so the worst-case space complexity is $\mathcal{O}(\kappa \log_2 T)$ bits.

The security of the tree construction is given be the following theorem.

Theorem 10 *Given a pseudorandom generator G and a pseudorandom function* PRF*,* TreeKU *is a secure key-updating scheme:*

$$\mathsf{Adv}^{\mathsf{KU}}_{\mathsf{TreeKU}}(T, \tau) \leq T\mathsf{Adv}^{\mathtt{prf}}_{\mathsf{PRF}}(\tau + \mathsf{TimePRF}(\mathsf{PRF}), 1) + (T-1)\mathsf{Adv}^{\mathtt{prg}}_{G}(\tau),$$

where $\mathsf{TimePRF}(\mathsf{PRF})$ *is the time to compute one application of* PRF*.*

Proof: Scheme TreeKU with $T = 2^d - 1$ time intervals can be obtained from $2^{d-1} - 1$ extended additive compositions of a trivial key-updating scheme TrivKU with one time interval, defined as follows:

- $\mathsf{Init}(1^\kappa, T, s)$ generates a random user key $M \xleftarrow{R} \{0,1\}^\kappa$ from the seed s and outputs $\mathsf{CS}_0 = M$.
- $\mathsf{Update}(t, \mathsf{CS}_t)$ outputs $\mathsf{CS}_{t+1} \leftarrow \mathsf{CS}_t$ only for $t = 0$.
- $\mathsf{Derive}(t, \mathsf{CS}_t)$ outputs $\mathsf{UK}_t \leftarrow M$ for $t = 1$.

- $\mathsf{Extract}(t, \mathsf{UK}_t, i)$ returns $k = \mathsf{PRF}_M(1)$ for $t = i = 1$.

Given that PRF is a pseudorandom function, it is easy to see that TrivKU is a secure key-updating scheme. Consider an adversary $\mathcal{A}$ for TrivKU. Since the scheme has one time interval, $\mathcal{A}$ is not given any user keys and it has to output stop at time interval 0. We build a distinguisher algorithm $\mathcal{D}$ for the pseudorandom function. $\mathcal{D}$ is given access to an oracle $G : \{0,1\} \rightarrow \{0,1\}^\kappa$, which is either $\mathsf{PRF}(k, \cdot)$ with $k \xleftarrow{R} \{0,1\}^\kappa$, or a random function $g \xleftarrow{R} \{f : \{0,1\} \rightarrow \{0,1\}^\kappa\}$. $\mathcal{D}$ gives to $\mathcal{A}$ the challenge $k_1 = G(1)$ and outputs the same bit as $\mathcal{A}$. It is immediate that the advantage of $\mathcal{D}$ in distinguishing the pseudorandom function from random functions is the same as the advantage of adversary $\mathcal{A}$ for TrivKU and this implies that

$$\mathsf{Adv}^{\mathsf{KU}}_{\mathsf{TrivKU}}(1, \tau) \leq \mathsf{Adv}^{\mathtt{prf}}_{\mathsf{PRF}}(\tau + \mathsf{TimePRF}(\mathsf{PRF}), 1).$$

The tree scheme with T time intervals can be constructed as follows: generate 2^{d-1} instances of TrivKU and make them leaves in the tree; build the tree bottom-up by additively composing (using the extended method) two adjacent nodes at the same level in the tree. The security of the binary tree scheme obtained by additive composition as described above follows from Theorem 6. $\square$

An incremental tree construction. We can construct an incremental tree scheme using ideas from the generic forward-secure signature scheme of Malkin et al. (Malkin et al. [2002]). The incremental scheme does not require the total number of time intervals to be known in advance.

Let $\mathsf{TreeKU}(i)$ be the binary tree construction with $2^i - 1$ nodes. Then the incremental tree scheme is obtained by additively composing binary tree schemes with increasing number of intervals: $\mathsf{TreeKU}(1) \oplus \mathsf{TreeKU}(2) \oplus \mathsf{TreeKU}(3) \oplus \ldots$. The keys generated by the tree scheme $\mathsf{TreeKU}(i)$ correspond to the time intervals between $2^i - i$ and $2^{i+1} - i - 2$ in the incremental scheme. Once the intervals of the tree scheme $\mathsf{TreeKU}(i)$ are exhausted, an instance of $\mathsf{TreeKU}(i+1)$ is generated, if needed.

In addition to allowing a practically unbounded number of time intervals, this construction has the property that the complexity of the Update, Derive and $\mathsf{Extract}$ algorithms and the size of the center state and user key depend on the number of past time intervals. Below we perform a detailed analysis of the cost of the scheme for an interval t that belongs to $\mathsf{TreeKU}(i)$ with $2^i - i \leq t \leq 2^{i+1} - i - 2$:

1. The center state includes all the root keys of the previous $i-1$ trees and the center state for node t in $\mathsf{TreeKU}(i)$. In the worst-case, this equals $(i-1) + (2i-1) =$

$3i-2 = 3\lceil\log_2(t)\rceil - 2$ tree-keys. Similarly, the user key for interval t includes the user key of node t as in scheme $\mathsf{TreeKU}(i)$ and the root keys of the previous $i-1$ trees, in total $(i-1)+(i-1) = 2i-2 = 2\lceil\log_2(t)\rceil - 2$ tree-keys. It follows that the space complexity of the center state and the user key for interval t is $\mathcal{O}(\kappa \log_2(t))$ bits.

2. The cost of both Update and Extract algorithms is at most $i = \lceil\log_2(t)\rceil$ applications of the pseudorandom generator. The cost of Derive is constant, as in the tree construction.

4.5 Performance of the Constructions

In this section we analyze the time complexity of the cryptographic operations and the space complexity of the center state and the user key for the three proposed constructions. Recall that all schemes generate keys of length κ. In analyzing the time complexity of the algorithms, we specify what kind of operations we measure and distinguish public-key operations (PK op.) from pseudorandom generator applications (PRG op.) because PK operations are typically much more expensive than PRG applications. We omit the time complexity of the Init algorithm, as it involves only the pseudorandom generator for all schemes except for the trapdoor permutation scheme, in which Init also generates the trapdoor permutation. The space complexities are measured in bits. Table 4.1 shows the details for a given number T of time intervals.

	CKU	TDKU	TreeKU
$\mathsf{Update}(t, \mathsf{CS}_t)$ time	0	1 PK op.	$\mathcal{O}(\log_2 T)$ PRG op.*
$\mathsf{Derive}(t, \mathsf{CS}_t)$ time	$T-t$ PRG op.	const.	$O(\log_2 T)$
$\mathsf{Extract}(t, \mathsf{UK}_t, i)$ time	$t-i$ PRG op.	$t-i$ PK op.	$\mathcal{O}(\log_2 T)$ PRG op.
Center state size	κ	poly(κ)	$\mathcal{O}(\kappa \log_2 T)$
User key size	κ	κ	$\mathcal{O}(\kappa \log_2 T)$

Table 4.1: Worst-case time and space complexities of the constructions for T time intervals. *Note: the amortized complexity of $\mathsf{Update}(t, \mathsf{CS}_t)$ in the binary tree scheme is constant.

In the chaining scheme CKU, the Update algorithm takes no work, but the Extract and the Derive algorithms take linear work in the number of time intervals. On the other hand, the trapdoor permutation scheme TDKU has efficient user-key derivation, which involves only a copy operation, but the complexity of the Update algorithm is one application of

the trapdoor permutation inverse and that of the $\mathsf{Extract}(t, \mathsf{UK}_t, i)$ algorithm is $t - i$ applications of the trapdoor permutation. The tree-based scheme TreeKU balances the tradeoffs between the complexity of the three algorithms, taking logarithmic work in the number of time intervals for all three algorithms in the worst-case. The Derive algorithm involves only $\mathcal{O}(\log_2 T)$ copy operations, and Update and Extract algorithms involve $\mathcal{O}(\log_2 T)$ PRG operations. This comes at the cost of increasing the center-state and user-key sizes to $\mathcal{O}(\kappa \log_2 T)$. Note that the amortized cost of the Update algorithm in the binary tree construction is constant.

As the chaining and the trapdoor permutation schemes have worst-case complexities linear in T for at least one algorithm, both of them require the number of time intervals to be rather small. In contrast, the binary tree construction can be used for a practically unbounded number of time intervals.

In an application in which the number of time intervals in not known in advance, the incremental tree scheme can be used. Its space and time complexities only depend on the number of past revocations and not on the total number of revocations supported. The incremental tree construction is an interesting example of an additive composition of tree constructions with different number of intervals. Furthermore, our additive and multiplicative composition methods allow the construction of new schemes starting from the basic three constructions described in Section 4.4.

4.6 Experimental Evaluation of the Three Constructions

We have implemented the chaining, trapdoor, and tree constructions for 128-bit keys. We have used the 128-bit AES block cipher to implement the pseudorandom generator G as $G(s) = AES_s(0^{128})||AES_s(1^{128})$ with $|s| = 128$ for the CKU and TreeKU constructions of Sections 4.4.1 and 4.4.3. In construction TDKU from Section 4.4.2, we have used the RSA permutation with a bit length of 1024 and public exponent 3 and the SHA-1 hash function as the random oracle h.

We performed the following experiment. For a fixed total number of revocations T, the center first initializes the key-updating scheme. Then, the steps below are repeated for $t = 1, \ldots, T$:

- The center runs the Update and Derive algorithms to simulate one revocation.
- Given the user key for interval t, the user runs the Extract algorithm to obtain the key k_1 for the *first* time interval.

Note that the time to extract the key for the first interval is larger than the extraction time for any other interval between 1 and t in all three constructions. Hence, the extraction time for the first interval represents a worst-case measure. We measured the performance

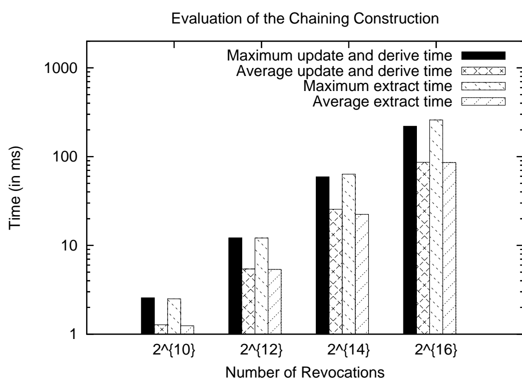

Figure 4.9: Evaluation of the chaining scheme.

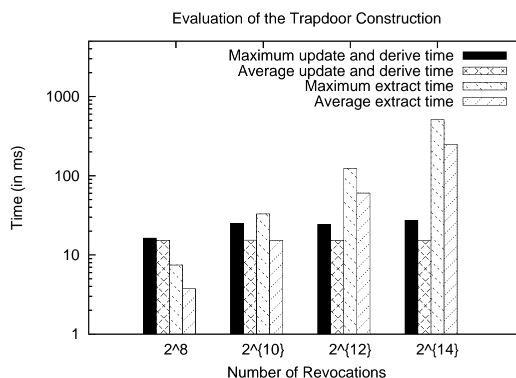

Figure 4.10: Evaluation of the trapdoor scheme.

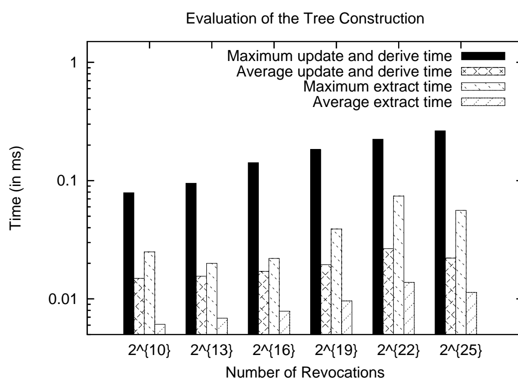

Figure 4.11: Evaluation of the tree scheme.

using four metrics: the maximum and average Update and Derive time for the center (over the T revocations), and the maximum and average Extract time for clients to compute the key for the first time interval (from one of the T time intervals). We ran our experiments on a 2.4 GHz Intel Xeon processor machine, running Linux 2.6. Our unoptimized implementation was written in C++ using `gcc` 3.2.1.

The results are presented in Figures 4.9, 4.10, and 4.11, respectively. The graphs show the measured time as a function of the total number of revocations T, which ranges from 2^8 to 2^{25} depending on the scheme. Note that both axis are logarithmic and that the ver-

tical axis differs for the three constructions. In the chaining construction, the cost of both the center and client computation increases linearly with the total number of revocations, as expected. In the trapdoor permutation construction, the center time is always constant, but the extraction time grows linearly with the total number of revocations. In the tree construction, all four metrics have a logarithmic dependence on the total number of revocations. We observe that the tree construction performs several orders of magnitude better than the other schemes.

Table 4.2 gives a direct comparison of the constructions in an experiment with 1024 revocations as above. It contains also the timing measurements for the first 1024 revocations in the tree construction where the upper bound T on number of revocations was set to a much larger value. This makes it possible to relate the tree construction to the trapdoor permutation scheme, which has no fixed upper bound on the number of revocations. It is evident that the tree scheme performs much better than the other schemes, even with a bound on the number of revocations that allows a practically unlimited number of them.

Scheme	T	Maximum Time Update+Derive (ms)	Average Time Update+Derive (ms)	Maximum Time Extract (ms)	Average Time Extract (ms)
Chaining	1024	2.57	1.28	2.5	1.24
Trapdoor	1024	25.07	15.36	32.96	15.25
Tree	1024	0.079	0.015	0.025	0.006
Tree	2^{16}	0.142	0.015	0.018	0.0076
Tree	2^{25}	0.199	0.015	0.02	0.01

Table 4.2: Evaluation of the three constructions for 1024 revocations.

The space usage for $T = 1024$ is as follows. The center state is 16 bytes for the chaining construction, 384 bytes for the trapdoor construction, and at most 328 bytes for the tree scheme. The size of the user key is 32 bytes for the chaining construction, 128 bytes for the trapdoor construction, and at most 172 bytes for the tree scheme. In general, for the tree scheme with depth d, the center state takes at most $(2d - 1)(16 + d/8)$ bytes, containing $2d - 1$ key value/key label pairs, assuming 16-byte keys and d-bit labels. The user key size is at most d key/label pairs, which take $d(16 + d/8)$ bytes.

In summary, we note that the performance of the tree scheme is superior to the others. The chaining construction has the smallest space requirements, but its computation cost becomes prohibitive for large T. The trapdoor construction has slightly smaller space requirements than the tree scheme, but these savings are very small compared to the additional computational overhead.

4.7 Cryptographic Primitives in the Lazy Revocation Model

We have described the abstraction of key-updating schemes, a method to manage the cryptographic keys of any symmetric-key cryptographic algorithm used in cryptographic file systems for lazy revocation. In this section, we provide definitions of the cryptographic primitives that can be used in a file system adopting the lazy revocation model. We start by giving rigorous definitions for symmetric encryption schemes in Section 4.7.1 and message authentication codes for lazy revocation in Section 4.7.2 and provide generic constructions starting from any secure key-updating scheme and a secure symmetric-key encryption scheme or message authentication code, respectively.

There exists cryptographic file system implementations in which signature schemes are used for providing file integrity (e.g., (Fu [1999], Fu et al. [2002], Adya et al. [2002], Kallahalla et al. [2003], Goh et al. [2003], Li et al. [2004])). To simplify key distribution, we choose to preserve the public verification key of the digital signature scheme constant across user revocations. We define security for signature schemes for lazy revocations that evolve the signing key at each revocation and keep the public key constant in Section 4.7.3. We also provide a generic transformation from identity-based signatures to signature schemes with lazy revocation.

Finally, we show how our constructions of cryptographic primitives for lazy revocation can be used to improve the key management scheme of the Plutus file system (Kallahalla et al. [2003]) in two ways: first, the extraction of encryption keys for previous time intervals can be done more efficiently than key rotation in Plutus, using only symmetric-key operations, and, secondly, using signature schemes with lazy revocation, the storage space taken by the signature verification keys can be reduced from linear in the number of revocations to a constant.

4.7.1 Symmetric Encryption Schemes with Lazy Revocation ($\mathsf{SE\text{-}LR}$)

In a cryptographic file system adopting lazy revocation, the encryption key for a file must be updated by the trusted entity (e.g., the owner of the file) when a user is revoked access to the file. Users might need to encrypt file blocks using the encryption key of the current time interval or to decrypt file blocks using *any* key of a previous time interval. Upon sending a corresponding request to the trusted entity, authorized users receive the *user key* of the current time interval from the trusted entity. Both the encryption and decryption algorithms take as input the user key, and the decryption algorithm additionally takes as

input the index of the time interval for which decryption is performed.

Symmetric encryption schemes with lazy revocation include Init, Update and Derive algorithms for key generation that are similar to the corresponding algorithms of key-updating schemes, and secret-key encryption and decryption algorithms.

Definition 15 (Symmetric Encryption with Lazy Revocation) *A symmetric encryption scheme with lazy revocation consists of a tuple of five polynomial-time algorithms* (Init, Update, Derive, E, D) *with the following properties:*

- *The* Init*,* Update *and* Derive *deterministic algorithms have the same specification as the corresponding algorithms of a key-updating scheme.*

- *The probabilistic encryption algorithm,* E*, takes as input a* time interval t*, the* user key UK_t *of the current time interval and a* message M*, and outputs a* ciphertext C.

- *The deterministic decryption algorithm,* D*, takes as input a* time interval t*, the* user key UK_t *of the current time interval, the* time interval $i \leq t$ *for which decryption is performed, and a* ciphertext C*, and outputs a* plaintext M.

Correctness of $\mathsf{SE\text{-}LR}$. Suppose that $\mathsf{CS}_0 \leftarrow \mathsf{Init}(1^\kappa, T, s)$ is the initial trusted state computed from a random seed s, $\mathsf{CS}_i \leftarrow \mathsf{Update}(i, \mathsf{Update}(i-1, \ldots, \mathsf{Update}(0, \mathsf{CS}_0)\ldots))$ is the trusted state for time interval $i \leq T$ and $\mathsf{UK}_i \leftarrow \mathsf{Derive}(i, \mathsf{CS}_i)$ is the user key for time interval i. The correctness property requires that $\mathsf{D}(t, \mathsf{UK}_t, i, \mathsf{E}(i, \mathsf{UK}_i, M)) = M$, for all messages M from the encryption domain and all i, t with $i \leq t \leq T$.

CPA-security of $\mathsf{SE\text{-}LR}$. The definition of CPA-security for $\mathsf{SE\text{-}LR}$ schemes requires that any polynomial-time adversary with access to the user key for a time interval t that it may choose adaptively (and, thus, with knowledge of all keys for time intervals prior to t), and with access to an encryption oracle for time interval $t+1$ is not able to distinguish encryptions of two messages of its choice for time interval $t+1$.

Formally, consider an adversary $\mathcal{A}$ that participates in the following experiment:

Initialization: Given a random seed, the initial trusted state CS_0 is generated with the Init algorithm.

Key compromise: The adversary adaptively picks a time interval t such that $0 \leq t < T$ as follows. Starting with $t = 0, 1, \ldots$, the adversary is given the user keys UK_t for

all consecutive time intervals until $\mathcal{A}$ decides to output stop or t becomes equal to $T-1$.

Challenge: When $\mathcal{A}$ outputs stop, it also outputs two messages, M_0 and M_1. A random bit b is selected and $\mathcal{A}$ is given a challenge $C = \mathsf{E}(t+1, \mathsf{UK}_{t+1}, M_b)$, where UK_{t+1} is the user key for time interval $t+1$ generated with the Init, Update and Derive algorithms.

Guess: $\mathcal{A}$ has access to an encryption oracle $\mathsf{E}(t+1, \mathsf{UK}_{t+1}, \cdot)$ for time interval $t+1$. At the end of this phase, $\mathcal{A}$ outputs a bit b' and succeeds if $b = b'$.

For an adversary $\mathcal{A}$ and a SE-LR scheme $\mathcal{E}^{\mathtt{lr}}$ with T time intervals we denote $\mathsf{Adv}^{\mathtt{cpa\text{-}lr}}_{\mathcal{E}^{\mathtt{lr}},\mathcal{A}}(T)$ its CPA advantage. W.l.o.g., we can relate the success probability of $\mathcal{A}$ and its advantage as

$$\Pr\big[\mathcal{A} \text{ succeeds}\big] = \frac{1}{2}\big[1 + \mathsf{Adv}^{\mathtt{cpa\text{-}lr}}_{\mathcal{E}^{\mathtt{lr}},\mathcal{A}}(T)\big]. \tag{4.17}$$

We denote by $\mathsf{Adv}^{\mathtt{cpa\text{-}lr}}_{\mathcal{E}^{\mathtt{lr}}}(T, \tau, q)$ the maximum advantage $\mathsf{Adv}^{\mathtt{cpa\text{-}lr}}_{\mathcal{E}^{\mathtt{lr}},\mathcal{A}}(T)$ over all adversary algorithms running in time at most τ and making q queries to the encryption oracle.

Tweakable ciphers (see Section 2.1.1). Tweakable ciphers with lazy revocation can be defined and implemented in a similar way as symmetric encryption schemes with lazy revocation. We omit here the details.

Generic construction. Let $\mathsf{KU} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Extract})$ be a secure key-updating scheme and $\mathcal{E} = (\mathsf{Gen}, \mathsf{E}, \mathsf{D})$ a CPA-secure symmetric encryption scheme such that the keys generated by KU have the same length as those generated by $\mathcal{E}$. We construct a symmetric encryption scheme with lazy revocation $\mathcal{E}^{\mathtt{lr}} = (\mathsf{Init}^{\mathtt{lr}}, \mathsf{Update}^{\mathtt{lr}}, \mathsf{Derive}^{\mathtt{lr}}, \mathsf{E}^{\mathtt{lr}}, \mathsf{D}^{\mathtt{lr}})$ as follows:

- The $\mathsf{Init}^{\mathtt{lr}}$, $\mathsf{Update}^{\mathtt{lr}}$, and $\mathsf{Derive}^{\mathtt{lr}}$ algorithms of $\mathcal{E}^{\mathtt{lr}}$ are the same as the corresponding algorithms of KU.
- The $\mathsf{E}^{\mathtt{lr}}(t, \mathsf{UK}_t, M)$ algorithm runs $k_t \leftarrow \mathsf{Extract}(t, \mathsf{UK}_t, t)$ and outputs $C \leftarrow \mathsf{E}_{k_t}(M)$.
- The $\mathsf{D}^{\mathtt{lr}}(t, \mathsf{UK}_t, i, M)$ algorithm runs $k_i \leftarrow \mathsf{Extract}(t, \mathsf{UK}_t, i)$ and outputs $M \leftarrow \mathsf{D}_{k_i}(C)$.

Theorem 11 *Suppose that* KU *is a secure key-updating scheme for lazy revocation with* T *time intervals and* $\mathcal{E}$ *is a CPA-secure symmetric encryption scheme with security parameter* κ*. Then* $\mathcal{E}^{\mathtt{lr}}$ *is a CPA-secure symmetric encryption scheme with lazy revocation:*

$$\begin{aligned}\mathsf{Adv}^{\mathtt{cpa\text{-}lr}}_{\mathcal{E}^{\mathtt{lr}}}(T,\tau,q) &\leq \mathsf{Adv}^{\mathsf{cpa}}_{\mathcal{E}}(\tau + \mathsf{TimeUserKeys}(T,\mathsf{KU}), q) \\ &+ 2\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}}(T,\tau).\end{aligned}$$

Proof: Correctness is easy to see. To prove CPA-security of $\mathcal{E}^{\mathtt{lr}}$, let $\mathcal{A}^{\mathtt{lr}}$ be an adversary algorithm for scheme $\mathcal{E}^{\mathtt{lr}}$ running in time τ. We construct an adversary $\mathcal{A}$ for the CPA-security of $\mathcal{E}$:

- $\mathcal{A}$ is given access to an encryption oracle $\mathsf{E}_k(\cdot)$.
- $\mathcal{A}$ generates a random seed s and uses this to generate an instance of the scheme KU.
- $\mathcal{A}$ gives to $\mathcal{A}^{\mathtt{lr}}$ the user keys UK_t from the instance of scheme KU generated in the step above.
- When $\mathcal{A}^{\mathtt{lr}}$ outputs stop at time interval t and two messages, M_0 and M_1, $\mathcal{A}$ also outputs M_0 and M_1.
- $\mathcal{A}$ is given challenge C and it gives this challenge to $\mathcal{A}^{\mathtt{lr}}$.
- When $\mathcal{A}^{\mathtt{lr}}$ makes a query to the encryption oracle for time interval $t+1$, $\mathcal{A}$ replies to this query using the encryption oracle $\mathsf{E}_k(\cdot)$.
- $\mathcal{A}$ outputs the same bit as $\mathcal{A}^{\mathtt{lr}}$.

The running time of $\mathcal{A}$ is $\tau + \mathsf{TimeUserKeys}(T,\mathsf{KU})$. From the construction of the simulation it follows that

$$\Pr\big[\mathcal{A} \text{ succeeds}\big] = \Pr[\mathcal{A}^{\mathtt{lr}} \text{ succeeds } | E\big],$$

where E is the event that $\mathcal{A}^{\mathtt{lr}}$ does not distinguish the simulation done by $\mathcal{A}$ from the CPA game defined in Section 4.7.1. The only difference between the simulation and the CPA game is that $\mathcal{A}$ uses in the simulation the encryption oracle with a randomly generated key to reply to encryption queries for time interval $t+1$, whereas in the CPA game the

encryption is done with key k_{t+1} generated with the Update, Derive and $\mathsf{Extract}$ algorithms of scheme KU. By the definition of E, we have $\Pr[\bar{E}] \leq \mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}}(T,\tau)$.

We can bound the probability of success of $\mathcal{A}^{\mathtt{lr}}$ as:

$$\begin{aligned}
\Pr\big[\mathcal{A}^{\mathtt{lr}} \text{ succeeds}\big] &= \Pr\big[\mathcal{A}^{\mathtt{lr}} \text{ succeeds } \mid E\big]\Pr\big[E\big]+ \\
&\quad \Pr\big[\mathcal{A}^{\mathtt{lr}} \text{ succeeds } \mid \bar{E}\big]\Pr\big[\bar{E}\big] \\
&\leq \Pr\big[\mathcal{A}^{\mathtt{lr}} \text{ succeeds } \mid E\big]+\Pr\big[\bar{E}\big] \\
&\leq \Pr\big[\mathcal{A} \text{ succeeds }\big]+\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}}(T,\tau). \qquad (4.18)
\end{aligned}$$

Using (4.1), (4.17), and (4.18) we obtain

$$\mathsf{Adv}^{\mathtt{cpa\text{-}lr}}_{\mathcal{E}^{\mathtt{lr}},\mathcal{A}^{\mathtt{lr}}}(T) \leq \mathsf{Adv}^{\mathtt{cpa}}_{\mathcal{E},\mathcal{A}} + 2\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}}(T,\tau).$$

The last relation proves the statement of the theorem. □

Implementation. In practice, we can instantiate the CPA-secure symmetric-encryption scheme with a block cipher (such as AES) in one of the CPA-secure modes of operation (FIPS81 [1980]) (e.g., cipher-block chaining) and the key-updating scheme with the efficient binary tree construction from Section 4.4.3, which only performs symmetric-key operations (more specifically, pseudo-random function applications implemented again by a block cipher).

Suppose that AES with 128-bit key size is used for the derivation of the keys. In a system that supports up to 1000 revocations, at most 10 AES computations need to be done for the Update, Derive and $\mathsf{Extract}$ algorithms. The center state and user keys consist of up to 10 AES keys or 160 bytes each. This adds a very small overhead to the cost of file data encryption.

4.7.2 Message-Authentication Codes with Lazy Revocation (MAC-LR)

If message-authentication codes are used for providing integrity in a cryptographic file system, then a secret key for computing and verifying authentication tags needs to be distributed to all authorized users. The users generate a tag using the key of the current time interval and may verify tags for any of the previous time intervals with the corresponding keys. Similar to symmetric-key encryption with lazy revocation, both the tagging and verification algorithms need to take as input the current user key, and the verification algorithm additionally takes as input the index of the time interval at which the tag was generated.

Message-authentication codes with lazy revocation include Init, Update and Derive algorithms for key generation that are similar to the corresponding algorithms of key-updating schemes, and secret-key tagging and verification algorithms.

Definition 16 (Message-Authentication Codes with Lazy Revocation) *A message-authentication code with lazy revocation consists of a tuple of five polynomial-time algorithms* $\mathsf{MA}^{\mathtt{lr}} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Tag}, \mathsf{Ver})$ *with the following properties:*

- *The* Init*,* Update *and* Derive *deterministic algorithms have the same specification as the corresponding algorithms of a key-updating scheme.*

- *The probabilistic tagging algorithm,* Tag*, takes as input a* time interval t*, the* user key UK_t *of the current time interval and a* message M*, and outputs an authentication* tag v.

- *The deterministic verification algorithm,* Ver*, takes as input a* time interval t*, the* user key UK_t *of the current time interval, the* time interval i *for which verification is performed, a* message M*, and a* tag v*, and outputs a* bit*. A tag* v *computed at time interval* i *is said to be* valid *on message* M *if* $\mathsf{Ver}(t, \mathsf{UK}_t, i, M, \mathsf{Tag}(i, \mathsf{UK}_i, M)) = 1$ *for some* $t \geq i$*.*

Correctness of $\mathsf{MAC\text{-}LR}$**.** Suppose that $\mathsf{CS}_0 \leftarrow \mathsf{Init}(1^\kappa, T, s)$ is the initial trusted state computed from a random seed s, $\mathsf{CS}_i \leftarrow \mathsf{Update}(i, \mathsf{Update}(i-1, \ldots, \mathsf{Update}(0, \mathsf{CS}_0)\ldots))$ is the trusted state for time interval $i \leq T$ and $\mathsf{UK}_i \leftarrow \mathsf{Derive}(i, \mathsf{CS}_i)$ is the user key for time interval i. The correctness property requires that $\mathsf{Ver}(t, \mathsf{UK}_t, i, M, \mathsf{Tag}(i, \mathsf{UK}_i, M)) = 1$, for all messages M from the message space and all i, t with $i \leq t \leq T$.

CMA-security of $\mathsf{MAC\text{-}LR}$**.** The definition of security for $\mathsf{MAC\text{-}LR}$ schemes requires that any adversary with access to the user key for a time interval t that it may choose adaptively (and, thus, with knowledge of all keys for time intervals prior to t), and with access to tagging and verification oracles for time interval $t+1$ is not able to create a valid tag on a message not queried to the tagging oracle.

Formally, consider an adversary $\mathcal{A}$ that participates in the following experiment:

Initialization: Given a random seed, the initial trusted state CS_0 is generated with the Init algorithm.

Key compromise: The adversary adaptively picks a time interval t such that $0 \leq t < T$ as follows. Starting with $t = 0, 1, \ldots$, the adversary is given the user keys UK_t for all consecutive time intervals until $\mathcal{A}$ decides to output stop or t becomes equal to $T-1$.

Tag generation: $\mathcal{A}$ has access to a tagging oracle $\mathsf{Tag}(t+1, \mathsf{UK}_{t+1}, \cdot)$ and a verification oracle $\mathsf{Ver}(t+1, \mathsf{UK}_{t+1}, \cdot, \cdot, \cdot)$ for time interval $t+1$ and outputs a message M and a tag v.

The adversary is successful if M was not a query to the tagging oracle and v is a valid tag on M for interval $t+1$. For a scheme with T time intervals, we denote by $\mathsf{Adv}^{\texttt{cma-lr}}_{\mathsf{MA}^{\texttt{lr}},\mathcal{A}}(T)$ the probability of success of $\mathcal{A}$ and by $\mathsf{Adv}^{\texttt{cma-lr}}_{\mathsf{MA}^{\texttt{lr}}}(T, \tau, q_1, q_2)$ the maximum advantage $\mathsf{Adv}^{\texttt{cma-lr}}_{\mathsf{MA}^{\texttt{lr}},\mathcal{A}}(T)$ over all adversary algorithms $\mathcal{A}$ running in time at most τ and making q_1 queries to the tagging oracle and q_2 queries to the verification oracle.

Generic construction. Let $\mathsf{KU} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Extract})$ be a secure key-updating scheme and $\mathsf{MA} = (\mathsf{Gen}, \mathsf{Tag}, \mathsf{Ver})$ a CMA-secure message-authentication code such that the keys generated by KU have the same length as those generated by MA. We construct a message-authentication code with lazy revocation $\mathsf{MA}^{\texttt{lr}} = (\mathsf{Init}^{\texttt{lr}}, \mathsf{Update}^{\texttt{lr}}, \mathsf{Derive}^{\texttt{lr}}, \mathsf{Tag}^{\texttt{lr}}, \mathsf{Ver}^{\texttt{lr}})$ as follows:

- The $\mathsf{Init}^{\texttt{lr}}$, $\mathsf{Update}^{\texttt{lr}}$, and $\mathsf{Derive}^{\texttt{lr}}$ algorithms of scheme $\mathsf{MA}^{\texttt{lr}}$ are the same as the corresponding algorithms of KU.
- The $\mathsf{Tag}^{\texttt{lr}}(t, \mathsf{UK}_t, M)$ algorithm runs $k_t \leftarrow \mathsf{Extract}(t, \mathsf{UK}_t, t)$ and outputs $C \leftarrow \mathsf{Tag}_{k_t}($
$M)$.
- The $\mathsf{Ver}^{\texttt{lr}}(t, \mathsf{UK}_t, i, M, v)$ algorithm runs $k_i \leftarrow \mathsf{Extract}(t, \mathsf{UK}_t, i)$ and outputs the value returned by $\mathsf{Ver}_{k_i}(M, v)$.

Theorem 12 *Suppose that* KU *is a secure key-updating scheme for lazy revocation with* T *time intervals and* MA *is a CMA-secure message-authentication code. Then* $\mathsf{MA}^{\texttt{lr}}$ *is a secure message-authentication code with lazy revocation:*

$$\begin{aligned}\mathsf{Adv}^{\texttt{cma-lr}}_{\mathsf{MA}^{\texttt{lr}}}(T, \tau, q_1, q_2) &\leq \mathsf{Adv}^{\mathsf{cma}}_{\mathsf{MA}}(\tau + \mathsf{TimeUserKeys}(T, \mathsf{KU}), q_1, q_2) \\ &+ 2\mathsf{Adv}^{\texttt{sku}}_{\mathsf{KU}}(T, \tau).\end{aligned}$$

Proof: Correctness is easy to see. To prove CMA-security for $\mathsf{MA}^{\mathtt{lr}}$, let $\mathcal{A}^{\mathtt{lr}}$ be an adversary algorithm for $\mathsf{MA}^{\mathtt{lr}}$ running in time τ. We construct an adversary $\mathcal{A}$ for MA:

- $\mathcal{A}$ is given access to a tagging oracle $\mathsf{Tag}(\cdot)$ and a verification oracle $\mathsf{Ver}(\cdot,\cdot)$.
- $\mathcal{A}$ generates a random seed s and uses this to generate an instance of scheme KU.
- $\mathcal{A}$ gives to $\mathcal{A}^{\mathtt{lr}}$ the user keys UK_t from the instance of scheme KU generated in the step above.
- When $\mathcal{A}^{\mathtt{lr}}$ makes a query to the tagging or verification oracle for time interval $t+1$, $\mathcal{A}$ replies to this query using the tagging oracle $\mathsf{Tag}(\cdot)$ and verification oracle $\mathsf{Ver}(\cdot,\cdot)$, respectively.
- $\mathcal{A}$ outputs the same message and tag pair as $\mathcal{A}^{\mathtt{lr}}$.

The running time of $\mathcal{A}$ is $\tau + \mathsf{TimeUserKeys}(T, \mathsf{KU})$. From the construction of the simulation it follows that

$$\Pr\big[\mathcal{A} \text{ succeeds}\big] = \Pr[\mathcal{A}^{\mathtt{lr}} \text{ succeeds } \mid E\big],$$

where E is the event that $\mathcal{A}^{\mathtt{lr}}$ does not distinguish between the simulation done by $\mathcal{A}$ and the MAC game from Section 4.7.2. Using a similar argument as in the proof of Theorem 11, we can bound $\Pr[\bar{E}] \leq \mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}}(T,\tau)$. It is immediate, as in the proof of Theorem 11 that

$$\Pr\big[\mathcal{A}^{\mathtt{lr}} \text{ succeeds }\big] \leq \Pr\big[\mathcal{A} \text{ succeeds }\big] + \mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}}(T,\tau)$$

and the conclusion of the theorem follows. ▯

Implementation. In practice, there are many efficient MAC schemes, such as CBC-MAC (Menezes et al. [1997]) or HMAC (Bellare et al. [1996]). They can be combined with key-updating schemes and achieve the same complexities as the implementation of symmetric encryption schemes with lazy revocation.

4.7.3 Signature Schemes with Lazy Revocation (SS-LR)

Signature schemes can be used for providing integrity of files. When differentiation of readers and writers is desired, a MAC is not sufficient because it is a symmetric primitive, and an asymmetric signature scheme is needed. The group signing key is distributed only

to writers, but the group verification key is given to all readers for the filegroup. Writers may modify files and recompute signatures using the signing key of the current time interval. Readers may check signatures on files generated at previous time intervals. We consider a model for signature schemes with lazy revocation in which the public key remains constant over time and only the signing keys change at the beginning of every time interval.

Signature schemes with lazy revocation include Init, Update and Derive algorithms similar to those of key-updating schemes, but with the following differences: the Init outputs also the public key of the signature scheme, and the Derive algorithm outputs directly the signing key for the time interval given as input. User keys in this case are the same as signing keys, as users perform operations only with the signing keys of the current time interval. SS-LR schemes also include signing and verification algorithms.

Definition 17 (Signature Schemes with Lazy Revocation) *A signature scheme with lazy revocation consists of a tuple of five polynomial-time algorithms* $\mathcal{S}^{\mathtt{lr}} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Sign}, \mathsf{Ver})$ *with the following properties:*

- *The deterministic initialization algorithm,* Init*, takes as input the* security parameter 1^κ*, the* number of time intervals T*, and a random seed* s*, and outputs an initial* trusted state CS_0 *and the* public key PK.
- *The deterministic key update algorithm,* Update*, takes as input the current* time interval t *and the current* trusted state CS_t*, and outputs a* trusted state CS_{t+1} *for the next time interval.*
- *The deterministic key derivation algorithm,* Derive*, takes as input a* time interval t *and the* trusted state CS_t*, and outputs a* signing key SK_t *for time interval* t.
- *The probabilistic signing algorithm,* Sign*, takes as input the* secret key SK_t *for time interval* t *and a* message M*, and outputs a* signature σ.
- *The deterministic verification algorithm,* Ver*, takes as input the* public key PK*, a* time interval t*, a* message M *and a* signature σ *and outputs a* bit*. A signature* σ *generated at time* t *is said to be* valid *on a message* M *if* $\mathsf{Ver}(\mathsf{PK}, t, M, \sigma) = 1$.

Correctness of SS-LR. Suppose that $(\mathsf{CS}_0, \mathsf{PK}) \leftarrow \mathsf{Init}(1^\kappa, T, s)$ are the public key and the initial trusted state computed from a random seed s, $\mathsf{CS}_i \leftarrow \mathsf{Update}(i, \mathsf{Update}(i - 1, \dots, \mathsf{Update}(0, \mathsf{CS}_0) \dots))$ is the trusted state for interval $i \leq T$ and $\mathsf{SK}_i \leftarrow \mathsf{Derive}(i, \mathsf{CS}_i)$ is the signing key for interval i. The correctness property requires that $\mathsf{Ver}(\mathsf{PK}, t, M, \mathsf{Sign}(\mathsf{SK}_t, M)) = 1$, for all messages M and all intervals $t \leq T$.

Security of $\mathsf{SS\text{-}LR}$**.** The definition of security for $\mathsf{SS\text{-}LR}$ requires that any adversary with access to the signing keys SK_i for $1 \leq i \leq t$, with t adaptively chosen, and a signing oracle for time interval $t+1$ is not able to generate a message and a valid signature for time interval $t+1$ that was not obtained from the signing oracle.

Formally, consider an adversary $\mathcal{A}$ that participates in the following experiment:

Initialization: Given a random seed, the initial trusted state CS_0 and the public key PK are generated with the Init algorithm. PK is given to $\mathcal{A}$.

Key compromise: The adversary adaptively picks a time interval t such that $0 \leq t < T$ as follows. Starting with $t = 0, 1, \ldots$, the adversary is given the signing keys SK_t for all consecutive time intervals until $\mathcal{A}$ decides to output stop or t becomes equal to $T - 1$.

Signature generation: $\mathcal{A}$ is given access to a signing oracle $\mathsf{Sign}(\mathsf{SK}_{t+1}, \cdot)$ for time interval $t+1$ and outputs a message M and signature σ.

The adversary is successful if M was not a query to the signing oracle and σ is a valid signature on M for time interval $t+1$. For a scheme $\mathcal{S}^{\mathtt{lr}}$ with T time intervals, we denote by $\mathsf{Adv}^{\mathtt{cma\text{-}lr}}_{\mathcal{S}^{\mathtt{lr}},\mathcal{A}}(T)$ the probability of success of $\mathcal{A}$ and by $\mathsf{Adv}^{\mathtt{cma\text{-}lr}}_{\mathcal{S}^{\mathtt{lr}}}(T, \tau, q)$ the maximum advantage $\mathsf{Adv}^{\mathtt{cma\text{-}lr}}_{\mathcal{S}^{\mathtt{lr}},\mathcal{A}}(T)$ over all adversary algorithms $\mathcal{A}$ running in time at most τ and making q queries to the signing oracle.

Generic construction from identity-based signatures. We construct a signature scheme with lazy revocation from an identity-based signature scheme by letting every time interval define a different identity. Let $\mathcal{S} = (\mathsf{MKGen}, \mathsf{UKGen}, \mathsf{Sign}, \mathsf{Ver})$ be a secure identity-based signature scheme. We construct a signature scheme with lazy revocation $\mathcal{S}^{\mathtt{lr}} = (\mathsf{Init}^{\mathtt{lr}}, \mathsf{Derive}^{\mathtt{lr}}, \mathsf{Update}^{\mathtt{lr}}, \mathsf{Sign}^{\mathtt{lr}}, \mathsf{Ver}^{\mathtt{lr}})$ as follows:

- $\mathsf{Init}^{\mathtt{lr}}(1^\kappa, T)$ runs $(\mathsf{MSK}, \mathsf{MPK}) \leftarrow \mathsf{MKGen}(1^\kappa)$ and outputs the initial trusted state $\mathsf{CS}_0 = \mathsf{MSK}$ and the public key MPK for the signature scheme.
- $\mathsf{Update}^{\mathtt{lr}}(t, \mathsf{CS}_t)$ outputs $\mathsf{CS}_{t+1} \leftarrow \mathsf{CS}_t$.
- $\mathsf{Derive}^{\mathtt{lr}}(t, \mathsf{CS}_t)$ runs $\mathsf{SK}_t \leftarrow \mathsf{UKGen}(\mathsf{CS}_0, t)$ and outputs SK_t.
- $\mathsf{Sign}^{\mathtt{lr}}(\mathsf{SK}_t, M)$ runs $\sigma \leftarrow \mathsf{Sign}(\mathsf{SK}_t, M)$ and outputs σ.
- $\mathsf{Ver}^{\mathtt{lr}}(\mathsf{MPK}, t, M, \sigma)$ outputs the same as $\mathsf{Ver}(\mathsf{MPK}, t, M, \sigma)$.

Theorem 13 *Suppose that $\mathcal{S}$ is a secure identity-based signature scheme. Then $\mathcal{S}^{\texttt{lr}}$ is a secure signature scheme with lazy revocation:*

$$\mathsf{Adv}^{\texttt{cma-lr}}_{\mathcal{S}^{\texttt{lr}}}(T, \tau, q) \leq \mathsf{Adv}^{\mathsf{ibs}}_{\mathcal{S}}(\tau + T \cdot \mathsf{TimeCorrupt}, 0, T, q),$$

where $\mathsf{TimeCorrupt}$ *is the time to run* $\mathsf{Corrupt}(\cdot)$.

Proof: Correctness is easy to see. To prove security of $\mathcal{S}^{\texttt{lr}}$, let $\mathcal{A}^{\texttt{lr}}$ be an adversary algorithm for $\mathcal{S}^{\texttt{lr}}$ running in time τ. We construct an adversary $\mathcal{A}$ for $\mathcal{S}$:

- $\mathcal{A}$ is given the public key MPK of scheme $\mathcal{S}$. $\mathcal{A}$ gives MPK to $\mathcal{A}^{\texttt{lr}}$.
- When $\mathcal{A}^{\texttt{lr}}$ requests the secret key UK_t, $\mathcal{A}$ runs $\mathsf{SK}_t \leftarrow \mathsf{Corrupt}(t)$ and gives SK_t to $\mathcal{A}^{\texttt{lr}}$.
- When $\mathcal{A}^{\texttt{lr}}$ makes a query M to the signing oracle for interval $t+1$, $\mathcal{A}$ runs $\sigma \leftarrow \mathsf{Sign}(t+1, M)$ and returns σ to $\mathcal{A}^{\texttt{lr}}$.
- Finally, $\mathcal{A}^{\texttt{lr}}$ outputs a message M and a signature σ for time interval $t+1$. Then, $\mathcal{A}$ outputs $(t+1, M, \sigma)$.

It is immediate that the probability of success of $\mathcal{A}$ is the same as the probability of success of $\mathcal{A}^{\texttt{lr}}$. The running time of $\mathcal{A}$ is $\tau + T \cdot \mathsf{TimeCorrupt}$. The conclusion of the theorem follows immediately. ∎

Implementation. Generic constructions of identity-based schemes from a certain class of standard identification schemes, called *convertible*, are given by Bellare et al.(Bellare et al. [2004]). The most efficient construction of an IBS scheme is the Guillou-Quisquater scheme (Guillou and Quisquater [1988]) that needs two exponentiations modulo an RSA modulus N for both generating and verifying a signature. The size of a signature is two elements of Z_N^*.

Relation to key-insulated signature schemes. A signature scheme with lazy revocation that has T time intervals can be used to construct a perfect $(T-1, T)$ key-insulated signature scheme, as defined by Dodis et al. (Dodis et al. [2003b]). However, the two notions are not equivalent since the attack model for key-insulated signatures is stronger. An adversary for a $(T-1, T)$ key-insulated signature scheme is allowed to compromise the signing keys for any $T-1$ intervals out of the total T intervals. Further differences

between key-insulated signatures and SS-LR are that both the trusted entity and the user update their internal state at the beginning of every interval and that both parties jointly generate the signing keys for each interval.

4.7.4 Applications to Cryptographic File Systems

In this section, we show how our cryptographic algorithms with lazy revocation can be applied to distributed cryptographic file systems, using the Plutus file system as an example. This also leads to an efficiency improvement for the revocation mechanism in Plutus.

Plutus (Kallahalla et al. [2003]) is a secure file system that uses an innovative decentralized key management scheme. In Plutus, files are divided into filegroups, each of them managed by the owner of its files. Blocks in a file are each encrypted with a different symmetric *file-block key*. The encryptions of the file-block keys for all blocks in a file are stored in a *lockbox*, which is encrypted with a *file-lockbox key*. The hash of the file is signed with a *file-signing key* for integrity protection and the signature can be verified with a *file-verification key*. The file-lockbox, file-signing and file-verification keys are the same for all files in a filegroup. Differentiation of readers and writers is done by distributing the appropriate keys to the users. In particular, the group owner distributes the file-lockbox and file-verification keys only to readers, and the file-lockbox and file-signing keys only to writers.

Plutus uses lazy revocation and a mechanism called *key rotation* for efficient key management. The file-lockbox and file-verification keys for previous time intervals can be derived from the most recent keys. Our cryptographic primitives with lazy revocation generalize the key rotation mechanism because we allow previous keys to be derived from our user key, which may be different from the actual key used for cryptographic operations at the current time interval. This allows more flexibility in constructing key-updating schemes.

We now recall the Plutus key rotation mechanisms for encryption and signing keys and demonstrate in both cases how our cryptographic primitives with lazy revocation lead to more efficient solutions.

For *encryption*, the group manager as the trusted entity uses the inverse of the RSA trapdoor permutation to update the file-lockbox encryption key after every user revocation. Users derive file-lockbox keys of previous time intervals using the public RSA trapdoor permutation. The construction does not have a cryptographic security proof and cannot be generalized to arbitrary trapdoor permutations because the output of the trapdoor permutation is not necessarily uniformly distributed. But it could be fixed by applying a hash

function to the output of the trapdoor permutation for deriving the key, which makes the construction provably secure in the random oracle model. Indeed, this is our *trapdoor permutation* key-updating scheme from Section 4.4.

However, the binary-tree key-updating scheme is more efficient because it uses only symmetric-key operations (e.g., a block cipher). Used in a symmetric encryption scheme with lazy revocation according to Section 4.7.1, it improves the time for updating and deriving file-lockbox keys by several orders of magnitude.

For *signatures*, Plutus uses RSA in a slightly different method than for encryption. A different public-key/secret-key pair is generated by the group owner after every revocation, and hence the RSA moduli differ for all time intervals and need to be stored with the file meta-data. The public verification exponent can be derived from the file-lockbox key by readers. An alternative solution based on our signature schemes with lazy revocation according to Section 4.7.3 uses only one verification key and achieves two distinct advantages: first, the storage space for the public keys is reduced to a constant from linear in the number of revocations and, secondly, the expensive operation of deriving the public verification exponent in Plutus does not need to be performed. For example, using the Guillou-Quisquater IBS scheme, deriving the public key of a time interval during verification takes only a few hash function applications.

4.8 Related Work

Time-evolving cryptography. The notion of secure key-updating schemes is closely related to forward- and backward-secure cryptographic primitives. Indeed, a secure key-updating scheme is forward-secure as defined originally by Anderson (Anderson [2002]), in the sense that it maintains security in the time intervals following a key exposure. However, this is the opposite of the forward security notion formalized by Bellare and Miner (Bellare and Miner [1999]) and used in subsequent work. Here we use the term forward security to refer to the latter notion.

Time-evolving cryptography protects a cryptographic primitive against key exposure by dividing the time into intervals and using a different secret key for every time interval. Forward-secure primitives protect past uses of the secret key: if a device holding all keys is compromised, the attacker cannot have access to past keys. In the case of forward-secure signatures, the attacker cannot generate past signatures on behalf of the user, and in the case of forward-secure encryption, the attacker cannot decrypt old ciphertexts. There exist many efficient constructions of forward-secure signatures (Bellare and Miner [1999], Abdalla and Reyzin [2000], Itkis and Reyzin [2001]) and several generic

constructions (Krawczyk [2000], Malkin et al. [2002]). Bellare and Yee (Bellare and Yee [2003]) analyze forward-secure private-key cryptographic primitives (forward-secure pseudorandom generators, message authentication codes and symmetric encryption) and Canetti et al. (Canetti et al. [2003]) construct the first forward-secure public-key encryption scheme.

Forward security has been combined with backward security in models that protect both the past and future time intervals, called key-insulated (Dodis et al. [2002, 2003b]) and intrusion-resilient models (Itkis and Reyzin [2002], Dodis et al. [2003a]). In both models, there is a center that interacts with the user in the key update protocol. The basic key insulation model assumes that the center is trusted and the user is compromised in at most t time intervals and guarantees that the adversary does not gain information about the keys for the intervals the user is not compromised. A variant of this model, called strong key insulation, allows the compromise of the center as well. Intrusion-resilience tolerates arbitrarily many break-ins into both the center and the user, as long as the break-ins do not occur in the same time interval. The relation between forward-secure, key-insulated and intrusion-resilient signatures has been analyzed by Malkin et al. (Malkin et al. [2004]). A survey of forward-secure cryptography is given by Itkis (Itkis).

Re-keying, i.e., deriving new secret keys periodically from a master secret key, is a standard method used by many applications. It has been formalized by Abdalla and Bellare (Abdalla and Bellare [2000]). The notion of key-updating schemes that we define is closely related to re-keying schemes, with the difference that in our model, we have the additional requirement of being able to derive past keys efficiently.

Multicast key distribution. In key distribution schemes for multicast, a group controller distributes a group encryption key to all users in a multicast group. The group of users is dynamic and each join or leave event requires the change of the encryption key. The goal is to achieve both forward and backward security. In contrast, in our model of key-updating schemes users should be able to derive past encryption keys efficiently.

A common key distribution model for multicast is that of *key graphs*, introduced by Wong et al. (Wong et al. [2000]) and used subsequently in many constructions (Sherman and McGrew [2003], Rodeh et al. [2001], Goshi and Ladner [2003],Goodrich et al. [2004]). In these schemes, each user knows its own secret key and, in addition, a subset of secret keys used to generate the group encryption key and to perform fast update operations. The relation between users and keys is modeled in a directed acyclic graphs, in which the source nodes are the users, intermediary nodes are keys and the unique sink node is the group encryption key. A path from a user node to the group key contains all the keys known to that user. The complexity and communication cost of key update operations

is optimal for tree structures (Tamassia and Triandopoulos [2005]), and in this case it is logarithmic in the number of users in the multicast group. We also use trees for generating keys, but our approach is different in considering the nodes of the tree to be only keys, and not users. We obtain logarithmic update cost in the number of revocations, not in the number of users in the group.

Key Management in Cryptographic Storage Systems. Early cryptographic file systems (Blaze [1993], Cattaneo et al. [2001]) did not address key management. Cepheus (Fu [1999]) is the first cryptographic file system that considers sharing of files and introduces the idea of lazy revocation for improving performance. However, key management in Cepheus is centralized by using a trusted key server for key distribution. More recent cryptographic file systems, such as Oceanstore (Kubiatowicz et al. [2000]) and Plutus (Kallahalla et al. [2003]), acknowledge the benefit of decentralized key distribution and propose that key management is handled by file owners themselves. For efficient operation, Plutus adopts a lazy revocation model and uses a key-updating scheme based on RSA, as described in Section 4.4.2.

Farsite (Adya et al. [2002]), SNAD (Miller et al. [2002]), SiRiUS (Goh et al. [2003]) and Windows EFS (Russinovich [1999]) use public-key cryptography for key management. The file encryption key is encrypted by the file owner with the public keys of all the users that are authorized to access the file. This approach simplifies key management, but the key storage per group is proportional to the number of users in the group. Our key-updating schemes for lazy revocation can be incorporated in such systems to provide efficient user revocation. In addition, they can be used in conjunction with the recently proposed collusion resistant broadcast encryption system by Boneh et al. (Boneh et al. [2005]) in order to reduce the storage space needed for the encryption keys to an amount independent of the number of users accessing the file. In more detail, the file owner could initialize a state that is updated after each revocation using for instance the binary tree key-updating scheme. From the state at a certain interval, a user key can be derived. The setup algorithm of the broadcast encryption scheme is run to generate a public key and secret keys for all authorized users. The user key of each interval could be stored encrypted under the public key of the broadcast encryption scheme of Boneh et al. (Boneh et al. [2005]) in the file header. The encrypted user key could be retrieved by an authorized user on demand and it could be decrypted by only knowing the appropriate secret key and the set of authorized users at a certain interval. An authorized user could efficiently extract the file encryption key from the decrypted user key. To add a new user in the system, the encrypted user key could be updated by performing a single exponentiation. To revoke a user's access, the state of the key-updating scheme needs to be updated and the new user

key has to be encrypted with the broadcast encryption scheme for the new set of users that have access to the file.

Independently and concurrently to our work Fu et al. (Fu et al. [2006]) have proposed a cryptographic definition for key-updating schemes, which they call *key regression schemes*. Key regression schemes are, in principle, equivalent to key-updating schemes. Their work formalizes three key regression schemes: two constructions, one using a hash function and one using a pseudo-random permutation, are essentially equivalent to our chaining construction, and an RSA-based construction originating in Plutus, which is equivalent to our trapdoor-permutation construction. Our composition methods and the tree-based construction are novel contributions that go beyond their work.

Chapter 5

On Consistency of Encrypted Files

In this chapter we address the problem of consistency for cryptographic file systems in which encrypted files are shared by clients. 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. Our goal is to find necessary and sufficient conditions for the consistency of an encrypted file object, knowing the consistency of the file access protocol and the key distribution protocol. To our knowledge, our work (Oprea and Reiter [2006a,b]) is the first to provide a framework for analyzing the consistency of encrypted files for generic consistency conditions.

We define in Section 5.2 two generic classes of consistency conditions that extend and generalize existing consistency conditions, after introducing some preliminary material in Section 5.1. We then formally define consistency for encrypted file objects in a generic way in Section 5.3: 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 provide, in our main result in Section 5.4, 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. We describe in Section 5.5 a consistent encrypted file built from from a sequentially consistent key object and a fork consistent file object using the protocol by (Mazieres and Shasha [2002]). A related work description is given in Section 5.6.

5.1 Preliminaries

5.1.1 Basic Definitions and System Model

Most of our definitions are taken from Herlihy and Wing (Herlihy and Wing [1990]). 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.

5.1.2 Eventual Propagation

A history satisfies *eventual propagation* (Friedman et al. [2002]) 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 18 (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 *(Friedman et al. [2002]) 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.

5.1.3 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 19 (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 $\xrightarrow{S}$.

5.2 Classes of Consistency Conditions

Our goal 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., Friedman et al. [2002]), 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 tree (denoted *forking tree*). This class includes fork consistency (Mazieres and Shasha [2002]), and extends that definition to other new, unexplored consistency conditions. The two classes do not cover all the existing (or possible) consistency conditions.

5.2.1 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* (Ahamad et al. [1995]) 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 20 (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:*

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

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 (Herlihy and Wing [1990]) and sequential consistency (Lamport [1979]) belong to $\mathcal{C}_{\mathcal{O}}^+$, and PRAM (Lipton and Sandberg [1988]) and causal consistency (Ahamad et al. [1995]) 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 5.2.3.

5.2.2 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 21 (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 22 (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 $\mathsf{FORK\text{-}C}_2$.

Remark. Fork consistency, as defined by Mazieres and Shasha (Mazieres and Shasha [2002]), belongs to class $\mathcal{C}_{\mathcal{F}}$ and is equivalent to $\mathsf{FORK\text{-}Linearizability}$.

5.2.3 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 5.1.

Examples of a linearizable and a causally consistent history are given in Figures 5.2 and 5.3, respectively. The history from Figure 5.2 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 Figure 5.3 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 $X.\mathsf{write}(2)$; $X.\mathsf{write}(1)$; $1 \leftarrow X.\mathsf{read}()$.

Class	Type of Serialization	Example of Condition	Partial Order
$\mathcal{C}_{\mathcal{O}}^{+}$	Serialization	Linearizability (Herlihy and Wing [1990])	$<_H$
		Sequential Consistency (Lamport [1979])	$\xrightarrow{lo}$
$\mathcal{C}_{\mathcal{O}} \setminus \mathcal{C}_{\mathcal{O}}^{+}$	Serialization set	Causal Consistency (Ahamad et al. [1995])	$\xrightarrow{*}$
		PRAM Consistency (Lipton and Sandberg [1988])	$\xrightarrow{lo}$
$\mathcal{C}_{\mathcal{F}}$	Forking Serialization Set	FORK-Linearizability	$<_H$
		FORK-Sequential Consistency	$\xrightarrow{lo}$

Figure 5.1: Classes of consistency conditions.

p_1 : X.write(1) $3 \leftarrow X$.read()

p_2 : X.write(2) X.write(3)

Figure 5.2: Linearizable history.

p_1 : X.write(1) $2 \leftarrow X$.read()

p_2 : X.write(2) $1 \leftarrow X$.read()

Figure 5.3: Causal consistent history.

We give an example of a history in Figure 5.4 that is not linearizable, but that accepts a forking tree shown in Figure 5.5 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.

p_1 : X.write(1) $2 \leftarrow X$.read() Y.write(1) $1 \leftarrow Y$.read()

p_2 : X.write(2) Y.write(2) $2 \leftarrow Y$.read()

p_3 : X.write(3) $3 \leftarrow X$.read()

Figure 5.4: FORK-Linearizable history.

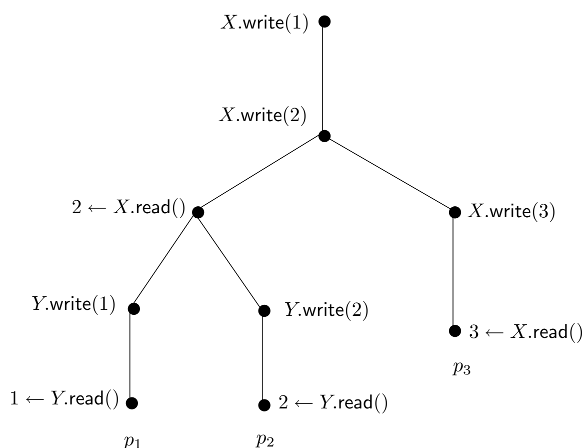

Figure 5.5: A forking tree for the history.

5.3 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 chapter, 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 (Fu [1999], Kallahalla et al. [2003], Backes et al. [2006]), 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 5.5:

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 23 *(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.

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

Definition 24 *(Consistency of Encrypted File Systems) Let H be a history of completed high-level operations on encrypted file objects $\mathsf{ENCF}_1, \dots, \mathsf{ENCF}_n$ 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 $\mathsf{KEY}_1, \dots, \mathsf{KEY}_n$ and file objects $\mathsf{FILE}_1, \dots, \mathsf{FILE}_n$ 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.

5.4 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 15, 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.

5.4.1 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 $w = \mathsf{KEY}.\mathsf{write}(k)$ 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 $r = (k \leftarrow \mathsf{KEY}.\mathsf{read}())$ 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.

5.4.2 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 5.6 of a history that is not key-monotonic for sequentially 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. $\mathsf{FILE}.\mathsf{write}(C_1)$ is after $\mathsf{FILE}.\mathsf{write}(C_2)$ with respect to the real-time ordering, and, thus, in any linearizable serialization of file operations, c_2 is written before C_1.

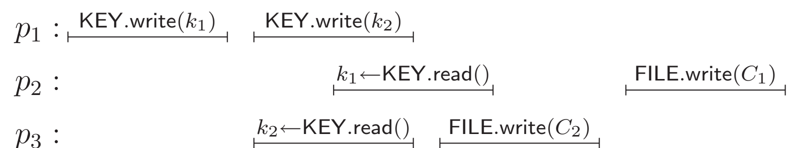

Figure 5.6: A history that is not key-monotonic.

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 25 (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* $F_1 \overset{S}{\longrightarrow} F_2$ *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* $k_2 \overset{\mathsf{C}_1,H|\mathsf{KEY}}{\longrightarrow} k_1$*;*
- (KM_2) *for any three file write operations* $F_1 \overset{S}{\longrightarrow} F_2 \overset{S}{\longrightarrow} F_3$*, 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 5.6 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 for a history H when the key write operations are uniquely ordered by the ordering relation $\overset{\mathsf{C}_1,H|\mathsf{KEY}}{\longrightarrow}$. To handle concurrent key writes with respect to $\overset{\mathsf{C}_1,H|\mathsf{KEY}}{\longrightarrow}$, 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 5.7 that is not key-monotonic for sequentially 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.

5.4.3 Simpler Conditions for Single-Writer Key Object

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

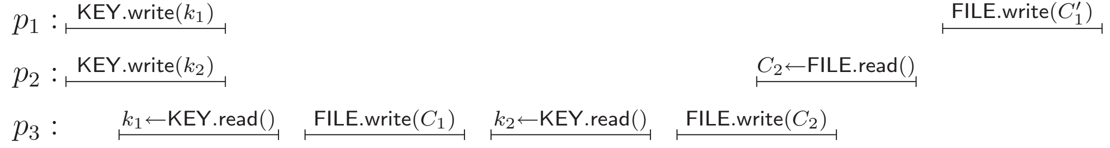

Figure 5.7: A history that does not satisfy condition (KM_2).

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 sequentially consistent, the key-monotonicity conditions given in Definition 25 can be simplified. We present below the simplified condition.

Proposition 14 *Let H be a history of operations on the single-writer key object* KEY *and file object* FILE *such that $H|\mathsf{KEY}$ is sequentially 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 $k_1 = k_2$.*

Proof: Suppose first that the conditions from Definition 25 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 25 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 25. 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.

5.4.4 Obtaining Consistency for Encrypted File Objects

We give here the main result of our chapter, a necessary and sufficient condition for implementing consistent encrypted file objects, as defined in Section 5.3. 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.

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 23). 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 23, 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 15 *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 23, 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 23 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 25 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 $\xrightarrow{\mathsf{C}_1,H_l|\mathsf{KEY}}$ 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:

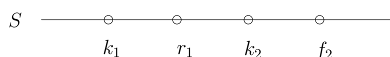

Figure 5.8: First case in which read r_1 cannot be inserted into S.

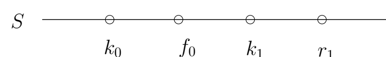

Figure 5.9: Second case in which read r_1 cannot be inserted into S.

- 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., $k_1 \xrightarrow{S} r_1 \xrightarrow{S} k_2 \xrightarrow{S} F_2$ (see Figure 5.8). 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 $k_1 \xrightarrow{S} r_1 \xrightarrow{S} k_2 \xrightarrow{S} F_2' \xrightarrow{S} F_2$ 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 $k_2 \xrightarrow{lo} F_2$ or there exists a key read operation r_2 that returns the value written by k_2 and $r_2 \xrightarrow{lo} F_2$.

In the first case, $k_2 \xrightarrow{lo} F_2$ and $F_2 \xrightarrow{lo} r_1$ imply $k_2 \xrightarrow{lo} r_1$. 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 $k_1 \xrightarrow{S} r_1 \xrightarrow{S} k_2$ and this represents a contradiction.

In the second case, $r_2 \xrightarrow{lo} F_2$ and $F_2 \xrightarrow{lo} r_1$ imply $r_2 \xrightarrow{lo} r_1$. For the same reason as above, r_2 should be ordered before r_1 in S. But $k_1 \xrightarrow{S} r_1 \xrightarrow{S} k_2 \xrightarrow{S} r_2$ 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., $k_0 \xrightarrow{S} F_0 \xrightarrow{S} k_1 \xrightarrow{S} r_1$ (see Figure 5.9). 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 23 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 25, 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. 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 (Miller et al. [2002]) and SiRiUS (Goh et al. [2003]), 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 (Mazieres and Shasha [2002]) in Section 5.5.

Moreover, the framework offered by Theorem 15 simplifies complex proofs for showing consistency of encrypted files. In order to apply Definition 23 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 15, 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 Section 5.5 demonstrates the usability of our framework.

5.5 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 (Mazieres and Shasha [2002]) that it is possible to construct a fork consistent file system even when the file server is potentially Byzantine. We use the SUNDR protocol (Mazieres and Shasha [2002]) together with our main result, Theorem 15, 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).

In our example application, a symmetric encryption scheme $\mathcal{E}$ is used to encrypt files (consisting of three algorithms Gen, E and D as defined in Section 4.1.3) and a signature scheme (consisting of three algorithms Gen, Sign and Ver as defined in Section 4.1.4) is used to protect the integrity of files. 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 sequentially consistent shared object can be used (e.g., Attiya and Welch [1994]), and we leave here the protocol unspecified. Finally, we prove the consistency of the protocol using our main result.

5.5.1 Implementation of High-Level Operations

The storage interface consists of four high-level operations similar to those presented in Section 5.3. Their implementation is detailed in Figure 5.10.

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 re-encrypted 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.

5.5.2 The File Access Protocol

We first describe the original SUNDR protocol (Mazieres and Shasha [2002]) 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.

```
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 ← E_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 ← E_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 ← D_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 5.10: The encrypted file protocol for client u.

The SUNDR protocol. In the SUNDR protocol, the storage server can be split into two components: a block store (denoted S_{BS}) on which clients can invoke read and write operations on blocks and a consistency server $\mathsf{S}_{\mathsf{CONS}}$ 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. $\mathsf{S}_{\mathsf{CONS}}$ 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 and only if $v[i] \leq w[i]$ for all $i = 1, \ldots, U$.

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 Merkle [1989] 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: $\mathsf{check_int}(C, v)$ that given an encrypted file C and a version structure v checks the integrity of C using the integrity information in v, and $\mathsf{update_int}(C, v)$ 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 5.11. 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, but we skipped this for clarity of presentation. The code for the block store and the consistency server for the "bare-bones" SUNDR protocol is in Figures 5.12 and 5.13, 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 5.13) to protect against faulty clients.

We refer the reader to the paper of Mazieres and Shasha (Mazieres and Shasha [2002]) for more details and a proof of fork consistency of this protocol.

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

```
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                               /* store locally the type of the operation */
9.      v_max ← check_cons()                         /* execute the version update protocol with S_CONS*/
10.     C ← S_BS.read()                              /* read the encrypted file from the block store */
11.     if not check_int(C, v_max)
12.         abort                                    /* check the integrity of the encrypted file */
13.     return (C, v_max.seq)

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

Figure 5.11: The file access protocol for client 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 23 in Figure 5.11). The key sequence numbers in the version structures guarantee that the file operations are serialized according to increasing key sequence numbers.

```
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 5.12: 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]              /* check that x is greater than all version structures */
6.       abort
7.    else v[u] ← x                             /* update the version structure for u */
```

Figure 5.13: The code for the consistency server $\mathsf{S_{CONS}}$.

5.5.3 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 24 in Figure 5.11 and line 5 in Figure 5.13). 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 16 *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 5.10. If $H_l|\mathsf{KEY}$ is sequentially consistent and the file access protocol is implemented with the protocol in Figure 5.11, then the execution of the protocol is* enc-*consistent.*

Proof: Conditions (1), (2), (3) and (4) from Theorem 15 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 $H_l|\mathsf{FILE}$ 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 19-23 of the protocol in Figure 5.11.

The protocol guarantees that $v^1_{max} \leq v^2_{max}$, which implies that $v^1_{max}.seq \leq v^2_{max}.seq$. Line 23 in the protocol from Figure 5.11 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 15, 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. ∎

5.6 Related Work

SUNDR (Li et al. [2004]) is the first file system that provides consistency guarantees (i.e., fork consistency (Mazieres and Shasha [2002])) in a model with a Byzantine storage server and benign clients. 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. We use the SUNDR protocol in our example implementation from Section 5.5 and show how to obtain a fork consistent encrypted file object with sequentially consistent key distribution.

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 (Castro and Liskov [1999]), SINTRA (Cachin and Poritz [2002]) and PASIS (Abd-El-Malek et al. [2005]). An example of a distributed encrypting file system that provides strong consistency guarantees for both file data and meta-data is FARSITE (Adya et al. [2002]). 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. (Lakshmanan et al. [2001]) 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 (Goh et al. [2003]) and Plutus (Kallahalla et al. [2003]), 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 (Mazieres et al. [1999]) separates key management from file system security and gives multiple schemes for key management; Cepheus (Fu [1999]) relies on a trusted server for key distribution; and SNAD (Miller et al. [2002]) uses separate key and file objects to secure network attached storage. However, none of these systems addresses consistency.

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 (Herlihy and Wing [1990]), sequential consistency (Lamport [1979]), and timed consistency (Torres-Rojas et al. [1999]) to loose consistency guarantees such as causal consistency (Ahamad et al. [1995]), PRAM consistency (Lipton and Sandberg [1988]), coherence (Goodman [1989], Gharachorloo et al. [1990]), processor consistency (Goodman [1989], Gharachorloo et al. [1990], Ahamad et al. [1992]), weak consistency (Dubois et al. [1988]), entry consistency (Bershad et al. [1993]), and release consistency (Lenoski et al. [1992]). A generic, continuous consistency model for wide-area replication that generalizes the notion of serializability (Bernstein et al. [1987]) for transactions on replicated objects has been introduced by Yu and Vahdat (Yu and Vahdat [2002]). 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* (Vitenberg and Friedman [2003]) and *composability* (Friedman et al. [2002]). 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 (Friedman et al. [2002]). 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.

Chapter 6

Conclusions

In this thesis, we have proposed three different mechanisms that could be used in securing networked storage architectures. The new proposed algorithms could be implemented at different layers (e.g., the object layer in object-based storage or the file layer in cryptographic file systems) when securing a storage systems.

This dissertation has presented as a first contribution novel algorithms for integrity checking in both block-level storage systems and cryptographic file systems. The algorithms are efficient by exploiting the low entropy of a large majority of block contents in real workloads. Several observations regarding the distribution of block and file accesses give further opportunities for optimization in the algorithms. The distribution of the block accesses in the disk traces we collected follow a power-law curve, with the majority of blocks (around 65%) being written only once. The NFS file traces collected at Harvard University exhibit a high sequentiality in file write accesses. A central idea in designing the new integrity algorithms is to use block write counters to protect against replay attacks. The particular observed distributions of block and file accesses allows a large reduction in the amount of trusted storage needed to represent these counters. While the performance and storage costs of the integrity algorithms for block-level storage are evaluated through simulation, the integrity algorithms for file integrity are integrated in the EncFS cryptographic file systems. A thorough evaluation on their performance relative to the Merkle tree standard is given, showing the impact of both the file contents and the file access patterns on the performance and storage costs of the new integrity algorithms.

A second contribution of this dissertation is to define a model of key management schemes for cryptographic file systems using lazy revocation, called key-updating schemes. Key-updating schemes can be used with any symmetric-key cryptographic algorithm to evolve the cryptographic keys of a group after each user revocation. They have the prop-

erty that older versions of the cryptographic keys of the group can be efficiently derived from some current user-key distributed to all users having access right to the group at the current time. The key-updating scheme based on a binary tree for deriving keys has logarithmic trusted storage and user key size in the total number of revocations, and worst-case logarithmic cost to update the keys after every revocation, and to extract any previous key. Our simulation results show that it improves several orders of magnitude upon previously proposed key-rotation schemes. Three characteristics make the new integrity and key management algorithms very practical: first, they do not require a modification of the storage interface; secondly, they minimize the amount of trusted storage; and, thirdly, they add a reasonable performance overhead.

Finally, this dissertation analyzes theoretically the consistency of an encrypted file object, for some generic classes of consistency conditions that generalize and extend well-known conditions for shared objects. In a model in which the encryption key for a file is viewed as a shared object, the consistency of both the file and the key object that encrypts the file contribute to the consistency of the encrypted file. The thesis presents necessary and sufficient conditions for the file access and key distribution protocols that guarantee the consistency of an encrypted file object, assuming some initial conditions are met. General conditions are given for the case in which the encryption key for a file could be written by all processes. In typical cryptographic file systems, the encryption key for a file is modified by a single user (e.g., the file owner or a trusted key distribution server), and we give simpler conditions for this particular case. The framework simplifies complex proofs for showing the consistency of encrypted file objects, as demonstrated by our example implementation of a fork consistent encrypted file object.

There are a number of open problems and future work directions suggested by this dissertation. In the area of integrity checking (not necessarily in file systems), are there any algorithms more efficient than Merkle trees independent of the data characteristics that reduce the amount of storage for integrity and the integrity bandwidth? Is there some provable lower bound on the amount of storage for integrity used by any integrity algorithm and an algorithm that meets this bound? Our new integrity algorithms for data encrypted with a tweakable cipher distinguish the blocks written to disks from uniformly random blocks by computing their entropy and comparing it with a given threshold. We have also experimented with a randomness test based on compression levels of block contents (i.e., a block is considered random if it cannot be compressed enough), but it was extremely inefficient and we did not include the results here. Another interesting question raised by our work is if there exists other efficient randomness tests with a provable bound on the false negative rate.

In the area of consistency for encrypted objects, we make an initial contribution in

defining consistency for two generic classes of consistency conditions and giving some conditions under which they can be realized. An important question is what are the right notions of consistency for file systems (encrypted and not encrypted). In particular, if the storage servers are Byzantine, what is the highest level of consistency that can be achieved? Is that fork consistency? Are there any more efficient implementations of forking conditions (i.e., fork consistency and weaker conditions from class $\mathcal{C}_{\mathcal{F}}$) than the existing ones?

Bibliography

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)*, pages 59–74. ACM, 2005. 5.6

M. Abdalla and M. Bellare. Increasing the lifetime of a key: A comparitive analysis of the security of rekeying techniques. In *Proc. Asiacrypt 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 546–559. Springer-Verlag, 2000. 4.8

M. Abdalla and L. Reyzin. A new forward-secure digital signature scheme. In *Proc. Asiacrypt 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 116–129. Springer-Verlag, 2000. 4.8

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. 3.4, 3.8, 4.7, 4.8, 5.6

M. K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. A. Thekkath. Block-level security for network-attached disks. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003. 3.8

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. 5.6

M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 1(9):37–49, 1995. 1.1, 1.2.3, 5.2.1, 5.2.1, 5.2.3, 5.6

R. Anderson. Two remarks on public-key cryptology. Technical Report UCAM-CL-TR-549, University of Cambridge, 2002. 4.8

J. H. Ann and M. Bellare. Does encryption with redundancy provide authenticity? In *Proc. Eurocrypt 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 512–528. Springer-Verlag, 2001. 2.9

H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994. 5.5

A. Azagury, R. Canetti, M. Factor, S. Halevi, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, and J. Satran. A two layer approach for securing an object store network. In *Proc. First Intl. IEEE Security in Storage Workhsop (SISW)*, 2002. 3.8

A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi. Towards an object store. pages 165–177, 2003. 3.8

M. Backes, C. Cachin, and A. Oprea. Secure key-updating for lazy revocation. Technical Report RZ 3627, IBM Research, August 2005a. 1.2.2, 4

M. Backes, C. Cachin, and A. Oprea. Lazy revocation in cryptographic file systems. In *Proc. 3rd Intl. IEEE Security in Storage Workhsop (SISW)*, 2005b. 1.2.2, 4

M. Backes, C. Cachin, and A. Oprea. Secure key-updating for lazy revocation. In *Proc. 11th European Symposium on Research in Computer Security*, volume 4189 of *Lecture Notes in Computer Science*, pages 327–346. Springer-Verlag, 2006. 1.2.2, 4, 5.3

M. Bellare and S. Miner. A forward-secure digital signature scheme. In *Proc. Crypto 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–448. Springer-Verlag, 1999. 4.8

M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Proc. Asiacrypt 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer-Verlag, 2000. 1.2.1, 2.5, 2.9

M. Bellare and P. Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. In *Proc. Asiacrypt 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 317–330. Springer-Verlag, 2000. 2.9

M. Bellare and B. Yee. Forward-security in private-key cryptography. In *Proc. The RSA conference - Cryptographer's track 2003(RSA-CT)*, volume 2612 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2003. 4.2.2, 4.8

M. Bellare, J. Kilian, and P. Rogaway. The security of the cipher block chaining message authentication code. In *Proc. Crypto 1994*, volume 839 of *Lecture Notes in Computer Science*, pages 341–358. Springer-Verlag, 1994. 2.1.4

M. Bellare, R. Canetti, and H. Krawczyk. Keyed hash functions for message authentication. In *Proc. Crypto 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1996. 4.7.2

M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer-Verlag, 1998. 2.9

M. Bellare, T. Kohno, and C. Namprempre. Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol. In *Proc. 9th ACM Conference on Computer and Communication Security (CCS)*, pages 1–11, 2002. 2.9

M. Bellare, C. Namprempre, and G. Neven. Security proofs for identity-based identification and signature schemes. In *Proc. Eurocrypt 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 268–286. Springer-Verlag, 2004. 4.7.3

P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. 5.6

B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared-memory system. In *Proc. IEEE COMPCON Conference*, pages 528–537. IEEE, 1993. 5.6

J. Black and H. Urtubia. Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption. In *Proc. 11th USENIX Security Symposium*, pages 327–338, 2002. 2.6.3

M. Blaze. A cryptographic file system for Unix. In *Proc. First ACM Conference on Computer and Communication Security (CCS)*, pages 9–16, 1993. 1.1, 4.8

M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 1994. 3.7, 3.8

D. Boneh, C. Gentry, and B. Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. In *Proc. Crypto 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 258–275. Springer-Verlag, 2005. 4.8

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. 5.6

R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. In *Proc. Eurocrypt 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 255–271. Springer-Verlag, 2003. 4.4.3, 4.8

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. 5.6

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. 1.1, 3.8, 4.8

D. E. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Incremental multiset hash functions and their application to memory integrity checking. In *Proc. Asiacrypt 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 188–207. Springer-Verlag, 2003. 3.8

D. E. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. van Dijk, and S. Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *Proc. 26th IEEE Symposium on Security and Privacy*, pages 139–153, 2005. 3.7, 3.8

Y. Dodis, J. Katz, S. Xu, and M. Yung. Key insulated public-key cryptosystems. In *Proc. Eurocrypt 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 65–82. Springer-Verlag, 2002. 4.8

Y. Dodis, M. Franklin, J. Katz, A. Miyaji, and M. Yung. Intrusion-resilient public-key encryption. In *Proc. The RSA conference - Cryptographer's track 2003(RSA-CT)*, volume 2612 of *Lecture Notes in Computer Science*, pages 19–32. Springer-Verlag, 2003a. 4.8

Y. Dodis, J. Katz, S. Xu, and M. Yung. Strong key-insulated signature schemes. In *Proc. 6th International Workshop on Theory and Practice in Public Key Cryptography (PKC)*, volume 2567 of *Lecture Notes in Computer Science*, pages 130–144. Springer-Verlag, 2003b. 4.7.3, 4.8

M. Dubois, C. Scheurich, and F.A. Briggs. Synchronization, coherence and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, 1988. 5.6

D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive nfs tracing of email and research workloads. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST)*, pages 203–216, 2003. 3, 3.3, 3.3, 3.6, 3.6.2

M. Factor, D. Nagle, D. Naor, E. Riedel, and J. Satran. The OSD security protocol. In *Proc. 3rd Intl. IEEE Security in Storage Workhsop (SISW)*, 2005. 1.1

FIPS197. Advanced encryption standard. Federal Information Processing Standards Publication 197, U.S. Department of Commerce/National Institute of Standards and Technology, National Technical Information Service, Springfield, Virginia, November 2001. 1.1

FIPS81. DES modes of operation. Federal Information Processing Standards Publication 81, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, 1980. 1.1, 4.7.1

R. Friedman, R. Vitenberg, and G. Chockler. On the composability of consistency conditions. *Information Processing Letters*, 86:169–176, 2002. 5.1.2, 18, 5.2, 5.6

K. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, 1999. 1.1, 1.1, 1.2.2, 3.4, 3.8, 4.7, 4.8, 5.3, 5.6

K. Fu, F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20:1–24, 2002. 1.1, 3.8, 4.7

Kevin Fu, Seny Kamaram, and Tadayoshi Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Proc. Network and Distributed Systems Security Symposium (NDSS 2006)*, 2006. 4.4, 4.8

FUSE. FUSE: filesystem in userspace. `http://fuse.sourceforge.net`. 1.2.1, 3.5

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 (ISCA)*, pages 15–26, 1990. 5.6

G. Gibson, D. Nagle, K. Amiri, J. Butler, F. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages –, 1998. 3.8

H. Gobioff, G. Gibson, and D. Tygar. Security of network-attached storage devices. Technical Report CMU-CS-97-118, CMU, 1997. 1.1, 3.8

H. Gobioff, D. Nagle, and G. Gibson. Integrity and performance in network-attached storage. Technical Report CMU-CS-98-182, CMU, 1998. 3.8

E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. Network and Distributed Systems Security Symposium (NDSS 2003)*, pages 131–145, 2003. 1.1, 3.8, 4.7, 4.8, 5.4.4, 5.6

S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, 1988. 4.1.4

J. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, 1989. 5.6

M. T. Goodrich, J. Z. Sun, and R. Tamassia. Efficient tree-based revocation in groups of low-state devices. In *Proc. Crypto 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 511–522. Springer-Verlag, 2004. 4.8

J. Goshi and R. E. Ladner. Algorithms for dynamic multicast key distribution trees. In *Proc. 22nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 243–251. ACM, 2003. 4.8

V. Gough. EncFS encrypted filesystem. `http://arg0.net/wiki/encfs`, 2003. 1.1, 1.2.1, 3.5

L. Guillou and J.J. Quisquater. A “paradoxical” identity-based signature scheme resulting from zero-knowledge. In *Proc. Crypto 1988*, volume 403 of *Lecture Notes in Computer Science*, pages 216–231. Springer-Verlag, 1988. 4.7.3

M. Halcrow. eCryptFS: An enterprise-class encrypted filesystem for linux. In *Proc. The Linux Symposium*, pages 201–218, 2005. 3.8

S. Halevi and P. Rogaway. A tweakable enciphering mode. In *Proc. Crypto 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 482–499. Springer-Verlag, 2003. 1.1, 2.1.1, 2.9, 3.5

S. Halevi and P. Rogaway. A parallelizable enciphering mode. In *Proc. The RSA conference - Cryptographer’s track 2004(RSA-CT)*, volume 2964 of *Lecture Notes in Computer Science*, pages 292–304. Springer-Verlag, 2004. 1.1, 2.9

J. S. Heidemann and G. J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, 1994. 3.8

M. Herlihy and J. Wing. Linearizability: A corretness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990. 1.1, 1.2.3, 5.1.1, 5.2.1, 5.2.3, 5.6

G. Itkis. Forward security, adaptive cryptography: Time evolution. Survey, available from `http://www.cs.bu.edu/fac/itkis/pap/forward-secure-survey.pdf`. 4.8

G. Itkis and L. Reyzin. Forward-secure signatures with optimal signing and verifying. In *Proc. Crypto 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 332–354. Springer-Verlag, 2001. 4.8

G. Itkis and L. Reyzin. SiBIR: Signer-base intrusion-resilient signatures. In *Proc. Crypto 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 499–514. Springer-Verlag, 2002. 4.8

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. 1.1, 1.1, 1.2.2, 3.4, 3.8, 4.4, 4.4.2, 4.7, 4.7.4, 4.8, 5.3, 5.6

J. Katz and M. Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In *Proc. 7th International Workshop on Fast Software Encryption (FSE 2000)*, volume 198 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2001. 2.9

V. Kher and Y. Kim. Securing distributed storage: Challenges, techniques, and systems. In *Proc. First ACM International Workshop on Storage Security and Survivability (StorageSS 2005)*, 2005. 3.8

G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A filesystem integrity checker. In *Proc. Second ACM Conference on Computer and Communication Security (CCS)*, pages 18–29, 1994. 3.8

T. Kohno, A. Palacio, and J. Black. Building secure cryptographic transforms, or how to encrypt and MAC, 2003. Cryptology ePrint Archive Report 2005/177. 2.9

H. Krawczyk. Simple forward-secure signatures from any signature scheme. In *Proc. 7th ACM Conference on Computer and Communication Security (CCS)*, pages 108–115, 2000. 4.8

H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL? In *Proc. Crypto 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. Springer-Verlag, 2001. 2.9

J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201. ACM, 2000. 1.1, 4.8

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. 5.6

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Computer*, 28(9):690–691, 1979. 1.1, 5.2.1, 5.2.3, 5.6

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. 5.6

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. 1.1, 3.8, 4.7, 5.6

R. Lipton and J. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, 1988. 1.1, 1.2.3, 5.2.1, 5.2.3, 5.6

M. Liskov, R. Rivest, and D. Wagner. Tweakable block ciphers. In *Proc. Crypto 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, 2002. 2.9

T. Malkin, D. Micciancio, and S. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *Proc. Eurocrypt 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 400–417. Springer-Verlag, 2002. 4.3, 4.4.3, 4.8

T. Malkin, S. Obana, and M. Yung. The hierarchy of key evolving signatures and a characterization of proxy signatures. In *Proc. Eurocrypt 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 306–322. Springer-Verlag, 2004. 4.8

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. 1.2.3, 5, 5.2, 5.2.2, 5.4.4, 5.5, 5.5.2, 5.5.2, 5.6

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. 3.8, 5.6

D. McGrew and S. Fluhrer. The extended codebook (XCB) mode of operation, 2004. Cryptology ePrint Archive Report 2004/278. 1.1, 2.9

Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, 1997. 4.7.2

J. Menon, D.A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM storage tank - a heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2), 2003. 3.8

R. Merkle. A certified digital signature. In *Proc. Crypto 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer-Verlag, 1989. 1.1, 3.1, 5.5.2

E. Miller, D. Long, W. Freeman, and B. Reed. Strong security for distributed file systems. In *Proc. 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002. 1.1, 3.8, 4.8, 5.4.4, 5.6

R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995. 2.1.5

M. Naor and O. Reingold. On the construction of pseudorandom permutations: Luby-rackoff revisited. In *Proc. ACM Symposium on Theory of Computing*, pages 189–199, 1997. 2.1.1

A. Oprea and M. K. Reiter. On consistency of encrypted files. Technical Report CMU-CS-06-113, Carnegie Mellon University, 2006a. Available from `http://reports-archive.adm.cs.cmu.edu/anon/2006/CMU-CS-06-113.pdf`. 1.2.3, 5

A. Oprea and M. K. Reiter. On consistency of encrypted files. In *Proc. 20th International Symposium on Distributed Computing (DISC)*, 2006b. 1.2.3, 5

A. Oprea and M. K. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. Technical Report CMU-CS-06-167, Carnegie Mellon University, 2006c. Available from `http://reports-archive.adm.cs.cmu.edu/anon/2006/CMU-CS-06-167.pdf`. 1.2.1, 3

A. Oprea, M. K. Reiter, and K. Yang. Space-efficient block storage integrity. In *Proc. Network and Distributed Systems Security Symposium (NDSS 2005)*. ISOC, 2005. 1.2.1

Osiris. Osiris: Host integrity management tool, 2004. `http://www.osiris.com`. 3.8

R. Pletka and C. Cachin. Cryptographic security for a high-performance distributed file system. Technical Report RZ 3661, IBM Research, September 2006. 1, 3.8

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. 3.8

O. Rodeh, K.P. Birman, and D. Dolev. Using AVL trees for fault tolerant group key management. *International Journal on Information Security*, 1(2):84–99, 2001. 4.8

P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Proc. 11th International Workshop on Fast Software Encryption (FSE 2004)*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer-Verlag, 2004. 2.1.2

M. Russinovich. Inside encrypting file system. *Windows and .NET Magazine*, June-July, 1999. 1.1, 4.8

Samhain. Samhain: File system integrity checker, 2004. `http://samhain.sourceforge.net`. 3.8

A. Shamir. Identity-based cryptosystems and signature schemes. In *Proc. Crypto 1984*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer-Verlag, 1985. 4.1.4

A. T. Sherman and D. A. McGrew. Key establishment in large dynamic groups using one-way function trees. *IEEE Transactions on Software Engineering*, 29(5):444–458, 2003. 4.8

G. Sivathanu, C. P. Wright, and E. Zadok. Enhancing file system integrity through checksums. Technical Report FSL-04-04, Stony Brook University, 2004. 3.8, 3.8

C. A. Stein, J.H. Howard, and M. I. Seltzer. Unifying file system protection. In *Proc. USENIX Annual Technical Conference 2001*, pages 79–90, 2001. 3.8

T10. SCSI object-based storage device commands (OSD). December 2004. 1.1

R. Tamassia and N. Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, 2005. 4.8

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. 5.6

R. Vitenberg and R. Friedman. On the locality of consistency conditions. In *Proc. 17th International Symposium on Distributed Computing (DISC)*, pages 92–105, 2003. 5.6

C. K. Wong, M. Gouda, and S. S. Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30, 2000. 4.8

C. P. Wright, J. Dave, and E. Zadok. Cryptographic file systems performance: What you don't know can hurt you. In *Proc. Second Intl. IEEE Security in Storage Workhsop (SISW)*, 2003a. 3.8

C. P. Wright, M. Martino, and E. Zadok. Ncryptfs: A secure and convenient cryptographic file system. In *Proc. USENIX Annual Technical Conference 2003*, pages 197–210, 2003b. 1.1, 3.8

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. 5.6

E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Columbia University, 1998. 3.8

Zlib. The zlib compression library. `http://www.zlib.net`. 3.5